FOM University of Applied Sciences for Economics and Management

university location Cologne

Bachelor Thesis

in the study course Business Informatics

to obtain the degree of

Bachelor of Science (B.Sc.)

on the subject

Implementation of an AI-Supported Centralized Procurement Interface to Enhance Procurement Processes in a Large-Scale Enterprise

by

Dominik Fey

Advisor:	Prof. Dr. Bernd Ulmann
Matriculation Number:	601373
Submission:	October 20, 2024

Contents

List of Figures

List of Tables

List of Codes

List of Abbreviations

AI	Artificial Intelligence
API	Application Programming Interface
CSS	Cascading Style Sheets
GenAI	Generative AI
HTML	Hypertext Markup Language
IaC	Infrastructure-as-Code
ID	Identification
IEEE	Institute of Electrical and Electronics Engineers
MVP	Minimum Viable Product
NLP	Natural Language Processing
NLU	Natural Language Understanding
PO	Product Owner
PSA	Privacy Security Assessment
RE	Requirements Engineering
SCSS	Sassy Cascading Style Sheets
SSO	Single Sign-On
TS	TypeScript

List of Symbols

Sperrvermerk

Die vorliegende Abschlussarbeit mit dem Titel ‘Implementation of an AI-Supported Centralized Procurement Interface to Enhance Procurement Processes in a Large-Scale Enterprise’ enthält unternehmensinterne Daten der Firma Deutsche Telekom AG. Daher ist sie nur zur Vorlage bei der FOM sowie den Begutachtern der Arbeit bestimmt. Für die Öffentlichkeit und dritte Personen darf sie nicht zugänglich sein.

Cologne, October 20, 2024 [illegible]

(Ort, Datum) (Eigenhändige Unterschrift)

1 Relevance

Dies soll eine LaTeX-Vorlage für den persönlichen Gebrauch werden. Sie hat weder einen Anspruch auf Richtigkeit, noch auf Vollständigkeit. Die Quellen liegen auf Github zur allgemeinen Verwendung. Verbesserungen sind jederzeit willkommen.

1.1 Objective

Kleiner Reminder für mich in Bezug auf die Dinge, die wir bei der Thesis beachten sollten und LaTeX -Vorlage für die Thesis.

1.2 Structure of the Paper

Kapitel 2 enthält die Inhalte des Thesis-Days und alles, was zum inhaltlichen erstellen der Thesis relevant sein könnte. In Kapitel 3 Methodology findet ihr wichtige Anmerkungen zu LaTeX , wobei die wirklich wichtigen Dinge im Quelltext dieses Dokumentes stehen (siehe auch die Verzeichnisstruktur in Abbildung.

1.3 Technical Stack Relevance

The selection of a robust and integrated technical stack is pivotal for the successful implementation of sophisticated software solutions, particularly when developing a chatbot designed to understand and fulfill complex customer needs. The project presented in this bachelor's thesis involves the creation of a chatbot that leverages Natural Language Processing (NLP) to interpret customer inquiries and match them with relevant items from a supplier catalog. To achieve this, the backend relies on Python and Haystack, while the frontend utilizes Vue, and the deployment is managed through containerization and orchestration technologies like Docker and Kubernetes.

1.3.1 Backend Technologies

Python serves as the backbone of the application due to its extensive ecosystem and its ability to seamlessly integrate various libraries and frameworks that facilitate rapid prototyping and the development of complex data-driven functionalities.[1,2] Its prominence in the fields of machine learning and data science makes it an ideal choice for implementing a chatbot that requires advanced Natural Language Understanding (NLU) capabilities.[3,4] Specifically, Python's compatibility with NLP frameworks like Haystack and many libraries ensures that the chatbot can parse user inputs and perform context-aware semantic searches.[5] This capability is crucial for accurately interpreting customer needs and mapping them to appropriate products or services in the supplier catalog.

The integration of Haystack within this project serves as a cornerstone for the development of an intelligent, search-driven chatbot, which is designed to address the intricate nature of customer inquiries. Haystack, a robust open-source NLP framework, employs a sophisticated Reader-Retriever architecture that harmonizes the capabilities of both information retrieval and deep semantic understanding. This dual approach capitalizes on advanced NLP methodologies to enhance the chatbot's performance in extracting pertinent information from extensive datasets.[6]

Notably, rather than utilizing the standard BERT model, this implementation leverages OpenAI's GPT-4o model within the Haystack framework. This allows the system to engage in nuanced contextual interpretation, thereby significantly improving the precision of semantic searches. The choice of GPT-4o is particularly advantageous in question-answering scenarios, as it allows the chatbot to comprehend the subtleties of customer queries and generate responses that are not only contextually relevant but also demonstrate a high degree of language understanding.[7]

Moreover, Haystack's modular architecture and extensible Application Programming Interfaces (APIs) offer a high degree of flexibility, facilitating seamless integration within the chatbot's overall architecture. This ensures that the processes of searching and retrieving supplier catalog data are executed with optimal accuracy and efficiency. Consequently, Haystack's inclusion in the technical stack is not merely contributory to the current system's capabilities but also establishes a solid foundation for prospective advancements and refinements in the chatbot's functional repertoire.

The backend system also incorporates PostgreSQL as its database solution. PostgreSQL's support for complex queries and its capability to handle structured data are essential for

[1] cf. *Shrivastava*, *S.*, 2024, p. 12.
[2] cf. *Christensen*, *S.* et al., 2022, pp. 240–241.
[3] cf. *Lortie*, *C. J.*, 2022, p. 1.
[4] cf. *Joshi*, *A.*, *Tiwari*, *H.*, 2024, p. 85.
[5] cf. *Fareez*, *M. M. M.*, *Thangarajah*, *V.*, *Saabith*, *S.*, 2020, p. 21.
[6] cf. *Krishnamoorthy*, *V.*, 2021, p. 236.
[7] cf. *Syed*, *Z. H.* et al., 2021, pp. 943–944.

managing and accessing the supplier information and product details stored within the system.[8] The integration of PostgreSQL ensures that the chatbot can quickly and efficiently retrieve the necessary data, thereby reducing latency and enhancing the overall user experience.

For monitoring and observability, the project employs Langfuse and OpenTelemetry, which provide comprehensive tracing and metrics collection across the microservices architecture.[9] This is particularly relevant given the experimental nature of the prototype, where understanding system performance and identifying potential bottlenecks are crucial for iterative development and refinement. By utilizing these tools, the project gains valuable insights into the behavior of the chatbot, allowing for continuous improvement and optimization.

FastAPI serves as the web framework for the backend, offering a high-performance environment that supports asynchronous programming.[10] This choice is particularly relevant for the chatbot, as it enables handling multiple concurrent requests with minimal overhead, ensuring that the application remains responsive even under heavy loads.

The project also leverages the `uv` package for dependency management and deployment configuration. `uv` simplifies the process of configuring Python dependencies and allows for a smoother deployment process by ensuring compatibility and consistency between various package versions.[11]

1.3.2 Frontend Technologie

On the frontend, Vue.js and Vuetify are utilized to create an intuitive and responsive user interface. The decision to use Vue.js stems from its reactive nature and modular architecture, which align with the need for a maintainable and easily extensible codebase.[12,13] It is effective for developing a chatbot interface that needs to present complex data in an accessible manner, while also allowing for dynamic updates based on user interactions.[14]

[8] cf. *Abbasi*, *M.* et al., 2024, pp. 23–24.
[9] cf. *Thakur*, *A.*, *Chandak*, *M. B.*, 2022, p. 15014.
[10] cf. *Chen*, *J.*, 2023, p. 9.
[11] cf. *Cano Rodríguez*, *J. L.*, 2024, URL last accessed on 2024-10-05.
[12] cf. *Kaluža*, *M.*, *Vukelic*, *B.*, 2018, p. 268.
[13] cf. *Li*, *N.*, *Zhang*, *B.*, 2021, pp. 1–2.
[14] cf. *Mokoginta*, *D.*, *Putri*, *D. E.*, *Wattimena*, *F. Y.*, 2024, p. 493.

1.3.3 Deployment and Infrastructure Management

Deployment is managed through a combination of Docker, Kubernetes, and Terraform. Docker's role in containerizing the application ensures that the entire software stack can be encapsulated and deployed consistently across various environments.[15] This is essential for a project like this, where different iterations of the prototype may need to be tested in different setups. Kubernetes, in turn, provides the orchestration needed to manage these containers, allowing for automated scaling and high availability.[16] The use of Terraform as an Infrastructure-as-Code (IaC) tool ensures that cloud resources can be provisioned and managed efficiently, providing a stable and reproducible deployment environment.[17]

In summary, each component of the technical stack has been carefully selected to meet the unique requirements of the chatbot project. The combination of Python and Haystack provides robust NLP capabilities for understanding and processing user inputs, while FastAPI support real-time interactions. PostgreSQL ensures efficient data management, and Langfuse and OpenTelemetry offer the necessary monitoring tools. On the frontend, Vue.js and Vuetify deliver a responsive and interactive user interface, and the deployment stack, comprised of Docker, Kubernetes, and Terraform, guarantees scalability and reliability. This cohesive selection of technologies forms a solid foundation for the development of a chatbot that not only meets the functional requirements but also adheres to best practices in software engineering.

[15] cf. *Openja*, *M.* et al., 2022, p. 191.
[16] cf. *Carrión*, *C.*, 2022, pp. 2, 7–8.
[17] cf. *N.*, *N.*, *Pub*, *I.*, 2023, p. 24.

2 Fundamentals

3 Methodology

3.1 Requirements Engineering

3.1.1 Introduction to Requirements Engineering According to Sommerville

Requirements Engineering (RE), as outlined by Ian Sommerville, is a multifaceted process aimed at systematically defining a system's specifications.[18] It comprises five interconnected activities: Requirements Documentation, Requirements Elicitation, Requirements Analysis and Negotiation, Requirements Validation, and Requirements Management.[19] Each activity has distinct objectives but collectively ensures the system satisfies both functional and non-functional requirements.[20]

Requirements Documentation serves as the foundation, formalizing needs, expectations, and constraints into a structured format, such as a Software Requirements Specification. This document bridges communication between stakeholders and developers and evolves with the project, supporting traceability and consistency. It is not merely a list of requirements but provides contextual rationale for each, enhancing clarity and stakeholder alignment.[21]

The process begins with Requirements Elicitation, which involves proactively engaging with stakeholders, including end-users and domain experts, to extract comprehensive requirements. Sommerville highlights this phase as an active process of understanding the problem domain and reconciling conflicting needs through techniques like interviews, workshops, and prototyping. This activity is vital for capturing all relevant requirements and establishing a shared understanding of the system's objectives.[22]

Next, Requirements Analysis and Negotiation transforms these inputs into a coherent and conflict-free set of requirements. Techniques such as modeling and prioritization help refine requirements, while negotiation resolves trade-offs between competing stakeholder priorities. This ensures a balanced and agreed-upon set of requirements, addressing concerns like performance versus cost or user flexibility versus security.[23]

Requirements Validation, a critical phase, ensures requirements reflect stakeholder needs and are feasible within constraints. Using techniques such as formal inspections, peer reviews, and prototyping, validation identifies ambiguities and contradictions early,

[18] cf. *Sommerville*, *I.*, *Sawyer*, *P.*, 1997, p.5.
[19] cf. *Sommerville*, *I.*, *Sawyer*, *P.*, 1997, p.11.
[20] cf. *Sommerville*, *I.*, *Sawyer*, *P.*, 1997, p.7–8.
[21] cf. *Sommerville*, *I.*, *Sawyer*, *P.*, 1997, pp. 38–40.
[22] cf. *Sommerville*, *I.*, *Sawyer*, *P.*, 1997, p.64–65.
[23] cf. *Sommerville*, *I.*, *Sawyer*, *P.*, 1997, p.112–113.

minimizing costly changes later and ensuring the system aligns with stakeholder expectations.[24]

Requirements Management, the final activity, involves tracking and controlling changes as the project evolves. Requirements are inherently dynamic, shifting with stakeholder understanding, new regulations, or market conditions. Sommerville's approach emphasizes maintaining traceability and documenting changes to ensure all modifications are transparent and approved, reducing risks and maintaining requirement integrity.[25]

3.1.2 Selection of Requirements Engineering Approach

The selection of the RE approach by Sommerville for this project was based on tailoring each RE activity to systematically uncover and refine requirements, thus providing a robust methodological foundation.

For Requirements Documentation, Sommerville's recommendations to "Define a Standard Document Structure" and "Make the Document Easy to Change" were implemented. The standardized structure ensured clarity and traceability, while the flexibility to modify the document supported iterative updates.[26,27] This approach facilitated rigorous traceability, deemed more critical for a prototype-focused project than alternatives like summarizing requirements, which prioritize readability over the detailed documentation needed to manage evolving requirements.[28]

In Requirements Elicitation, the strategies "Use Scenarios to Elicit Requirements," "Define Operational Processes," and "Prototyping Poorly Understood Requirements" were selected due to their effectiveness in refining complex and abstract requirements through iterative development and visualization. "Use Scenarios to Elicit Requirements" enabled precise visualization of user interactions, crucial for capturing the chatbot's nuanced behavior.[29] "Define Operational Processes" provided a structured method for visualizing workflows, ensuring accurate capture of procedural requirements, while "Prototype Poorly U nderstood Requirements" translated ambiguous requirements into tangible artifacts for stakeholder discussions.[30,31] These strategies were preferred over approaches like

[24] cf. *Sommerville*, *I.*, *Sawyer*, *P.*, 1997, p.190–191.
[25] cf. *Sommerville*, *I.*, *Sawyer*, *P.*, 1997, p.216–217.
[26] cf. *Sommerville*, *I.*, *Sawyer*, *P.*, 1997, p.41.
[27] cf. *Sommerville*, *I.*, *Sawyer*, *P.*, 1997, p.60.
[28] cf. *Sommerville*, *I.*, *Sawyer*, *P.*, 1997, p.47.
[29] cf. *Sommerville*, *I.*, *Sawyer*, *P.*, 1997, p.99.
[30] cf. *Sommerville*, *I.*, *Sawyer*, *P.*, 1997, p.102–103.
[31] cf. *Sommerville*, *I.*, *Sawyer*, *P.*, 1997, p.94–96.

"Collect Requirements From Multiple Viewpoints," as the prototype's limited scope did not necessitate broad stakeholder involvement from the outset.[32]

For Requirements Analysis and Negotiation, the project adopted "Plan for Conflicts and Conflict Resolution" due to its structured approach to identifying and resolving conflicting requirements.[33] This systematic method facilitated discussions on competing priorities and consensus-building, making it more suitable than less structured alternatives like "Define System Boundaries" or "Provide Software to Support Negotiations," which do not support the iterative, dialogic nature of ongoing negotiations in prototype development.[34,35]

The Requirements Validation phase applied "Organize Formal Requirements Inspections" and "Use Prototyping to Verify Requirements." The former enabled comprehensive identification of inconsistencies through structured reviews, while the latter facilitated visual validation and iterative feedback.[36,37] The decision to forego "Define Validation Checklists" was based on its less adaptable structure, which is not as effective for validating the evolving visual and interactive components of the project.[38]

Finally, in Requirements Management, "Define Policies for Requirements Management" was implemented to maintain traceability and manage changes systematically. Establishing clear policies ensured that all requirements could be efficiently tracked and modified in response to evolving project needs.[39] The project opted not to use a database for managing requirements, focusing instead on robust change management policies to ensure a controlled and flexible environment for reviewing and integrating changes.[40] This emphasis on policies over tools supported the dynamic nature of the prototype's development while preserving project coherence and stakeholder alignment.

3.1.3 Application of Requirements Engineering in the Project

The application of RE in this project adhered to an iterative and collaborative methodology, incorporating structured documentation, continuous stakeholder engagement, and interactive design tools to refine and validate requirements throughout the development

[32] cf. *Sommerville*, *I.*, *Sawyer*, *P.*, 1997, p.90.
[33] cf. *Sommerville*, *I.*, *Sawyer*, *P.*, 1997, p.125–127.
[34] cf. *Sommerville*, *I.*, *Sawyer*, *P.*, 1997, p.114.
[35] cf. *Sommerville*, *I.*, *Sawyer*, *P.*, 1997, p.121.
[36] cf. *Sommerville*, *I.*, *Sawyer*, *P.*, 1997, p.195–196.
[37] cf. *Sommerville*, *I.*, *Sawyer*, *P.*, 1997, p.203–204.
[38] cf. *Sommerville*, *I.*, *Sawyer*, *P.*, 1997, p.200.
[39] cf. *Sommerville*, *I.*, *Sawyer*, *P.*, 1997, p.221–222.
[40] cf. *Sommerville*, *I.*, *Sawyer*, *P.*, 1997, p.236.

lifecycle. This approach, guided by Sommerville's principles, enabled effective communication, alignment, and feedback across all parties.

The foundation, the Requirements Documentation, began with a structured Word document containing a requirements table, which served as the central repository for capturing, organizing, and updating requirements. This document ensured systematic traceability and provided a consistent foundation for ongoing modifications as new insights emerged. The example from Institute of Electrical and Electronics Engineers (IEEE), mentioned in Sommerville's book, is not used, because for a Prototype it is to much.[41] A simpler version was created.

During Requirements Elicitation, workshops with domain experts were conducted to deepen understanding of the procurement process and to capture core requirements. These sessions were complemented by ad-hoc meetings throughout development to address emerging needs and clarify ambiguities. The elicitation phase leveraged iterative prototyping, progressing from initial mockups in Miro to refined designs in Figma, and ultimately to functional screens in Vue. This incremental approach allowed stakeholders to engage with evolving system representations, offering targeted feedback and further refining requirements at each stage.

The Requirements Analysis and Negotiation phase synthesized feedback gathered during workshops and prototyping sessions. Weekly meetings provided a forum to discuss and resolve conflicting priorities among stakeholders, while bi-weekly management meetings ensured strategic alignment and secured high-level approval for changes. This continuous engagement facilitated a dynamic negotiation process, allowing for re-prioritization and refinement of requirements based on stakeholder feedback and evolving project constraints.

For Requirements Validation, a combination of systematic reviews and interactive prototyping was used. The prototypes, evolving from initial sketches in Miro to detailed designs in Figma and implemented screens in Vue, enabled stakeholders to visualize how their requirements would be realized, aiding in early identification of inconsistencies or misalignments. Regular reviews ensured alignment between the documented requirements and the evolving system, minimizing the risk of discrepancies.

Throughout Requirements Management, change control policies were strictly followed, ensuring that all requirement modifications were systematically documented, reviewed, and approved. The structured requirements table was continuously updated to reflect these changes, maintaining traceability and transparency. This disciplined approach to requirements management preserved control over evolving requirements and ensured that stakeholders remained informed of any adjustments.

[41] cf. *Sommerville, I., Sawyer, P.*, 1997, p.42–43.

3.2 Prototyping

3.2.1 Introduction to the Prototyping Methodology According to Floyd

The prototyping approach developed by Christiane Floyd represents a structured methodology used primarily in software development to improve communication between developers and users, reduce misunderstandings, and ultimately enhance the quality of the final product.[42] This methodology provides an alternative to the traditional linear, phase-oriented development process by introducing a dynamic element of iteration and feedback.[43] As a result, it facilitates a more interactive and user-centered development process.[44]

Floyd's prototyping process is structured as a cyclical sequence of four distinct steps: functional selection, construction, evaluation, and further use.
The first step, Functional Selection, involves identifying the specific functions that the prototype should demonstrate. The selected functionalities are derived from the relevant work tasks to ensure meaningful demonstrations, while acknowledging that the prototype does not need to represent the final product comprehensively. This allows for a degree of flexibility in the selection and prioritization of features to be included in the prototype.[45]

The second step, Construction, entails building the prototype using techniques and tools that enable rapid development and easy adjustments. At this stage, the focus is not on developing a fully functional system, but rather on demonstrating and assessing specific aspects of the final product. This approach enables the prototype to act as a tool for exploring different solutions and gathering feedback.[46]

Evaluation, the third step in the process, serves as the cornerstone of the prototyping methodology. In this step, feedback from all relevant stakeholders—including potential users—is collected and analyzed to refine and guide subsequent development stages. This iterative process ensures that the prototype evolves in alignment with user expectations and needs.[47]

The final step, Further Use, determines the prototype's role after the evaluation phase. Depending on its effectiveness and the degree to which it meets requirements, the prototype can be discarded, modified for continued use, or serve as a foundation for the final product.

[42] cf. *Floyd*, *C.*, 1984, p.2–3.
[43] cf. *Floyd*, *C.*, 1984, p.3.
[44] cf. *Floyd*, *C.*, 1984, p.3–4.
[45] cf. *Floyd*, *C.*, 1984, p.4.
[46] cf. *Floyd*, *C.*, 1984, p.4.
[47] cf. *Floyd*, *C.*, 1984, p.4–5.

This flexibility is critical in accommodating evolving requirements and objectives, making the prototyping approach particularly valuable in contexts where specifications are expected to change frequently.[48]

Floyd's methodology also categorizes prototyping into three primary approaches, each based on the goals and context of development: exploratory prototyping, experimental prototyping, and evolutionary prototyping.[49]
Exploratory Prototyping is primarily used to clarify requirements and foster creative cooperation between developers and users during the early stages of development. It is particularly useful when there is a lack of clarity on the system's final objectives, as it allows for broad experimentation and refinement of ideas before committing to a specific solution.[50]

In contrast, Experimental Prototyping focuses on testing proposed solutions to validate specific hypotheses, such as user interface design, system performance, or algorithm feasibility. This approach might involve techniques such as full functional simulation or human interface simulation to verify that the proposed design meets the intended objectives.[51]

Finally, Evolutionary Prototyping treats the prototype as a system that continuously evolves to adapt to changing requirements over time. Each version of the prototype serves as a basis for the next iteration, incorporating new insights and user feedback. This approach is especially beneficial in scenarios where requirements are expected to change frequently, rendering a static set of requirements impractical.[52]

To support these prototyping processes, various techniques and tools can be utilized.[53]
Modular Design, for instance, encourages the use of small, independent modules that can be replaced or refined as needed, thereby facilitating iterative development and easing integration into the final product.[54]
Additionally, Dialogue Design plays a crucial role in ensuring that the user interface is adaptable and transparent, which enables effective evaluation and modification of the user experience.[55]
Furthermore, Simulation techniques allow for simulating aspects of the final system that

[48] cf. *Floyd*, *C.*, 1984, p.5.
[49] cf. *Floyd*, *C.*, 1984, p.6.
[50] cf. *Floyd*, *C.*, 1984, p.6–7.
[51] cf. *Floyd*, *C.*, 1984, p.8–10.
[52] cf. *Floyd*, *C.*, 1984, p.10–12.
[53] cf. *Floyd*, *C.*, 1984, p.12.
[54] cf. *Floyd*, *C.*, 1984, p.12.
[55] cf. *Floyd*, *C.*, 1984, p.12.

are not yet fully implemented, enabling the assessment of system performance and user interaction without the need for a complete implementation.[56]

3.2.2 Selection of Prototyping Approach

Given the context and objectives of this project, the decision was made to adopt the experimental prototyping approach. This choice is rooted in the fact that the project requirements have already been well-defined through a comprehensive requirements engineering process, thus eliminating the need for exploratory prototyping. The clear specification of functionalities and user expectations ensures that the focus can shift from understanding requirements to validating and testing specific design choices.

Furthermore, the experimental approach is particularly well-suited for scenarios where a prototype is intended to serve as a preliminary proof of concept rather than a foundation for incremental development. This aligns perfectly with the anticipated lifecycle of the prototype in this project, which is expected to be discarded once the Minimum Viable Product (MVP) phase begins. As the project moves towards the MVP stage, a fresh start will be made, incorporating only validated concepts and findings from the experimental prototype. Therefore, evolutionary prototyping is not applicable, as it is primarily designed for projects that involve iterative refinement and continuous evolution of the same prototype.

The experimental prototyping approach allows for focused experimentation with various design elements, interface interactions, and technical implementations, all within a controlled environment that does not necessitate long-term integration into the final product.

3.2.3 Application of Experimental Prototyping in the Project

The experimental prototyping approach will be implemented in this project with a focus on validating user interface designs, interactions, and core functionalities against predefined requirements. The development of the prototype will leverage a set of carefully selected techniques that facilitate rapid iteration and feedback.

A key technique employed is simulation, which is used to mimic certain system behaviors without integrating the prototype into live production environments. This decision is motivated by the limited scope of the prototype and the intention to avoid disruptions to existing systems. By relying on test data instead of actual production data, the prototype

[56] cf. *Floyd*, *C.*, 1984, p.13.

can simulate real-world scenarios and provide valuable insights into its performance and user experience.

Moreover, the prototype will not attempt to implement every function in its final depth and breadth. Instead, certain features will be simulated to convey the look and feel of the system, providing a realistic representation of how the final product would function. This is where Modular Design plays a crucial role. By leveraging modular design principles , the prototype can separate the user interface from the underlying logic and backend functionalities. This allows specific modules to be developed exclusively for the UI, simulating the presence of certain features without the need for fully developed backend logic. For instance, UI elements such as buttons, forms, and interactive components can be displayed and interacted with as if they were functional, even though the backend processing is either simulated or entirely absent. This approach enables the developer to receive early feedback on key aspects of the design and functionality without committing extensive resources to full-scale implementation.

By focusing on these aspects, the prototype will serve as a learning tool, guiding the refinement of requirements and design choices before moving into the more resource-intensive MVP phase.

4 Requirements Engineering

4.1 Requirement Document

The document is structured to facilitate the collection of requirements from stakeholders in a clear and systematic manner. At its core, the document consists of a table where stakeholders are required to fill out specific fields related to the system requirements. The table is designed to capture both functional and non-functional requirements, providing the foundation for further analysis and prioritization by the Product Owner (PO). The empty template that was circulated to stakeholders for completion can also be found in Appendix 1 of the paper, providing a reference for how the requirements were initially collected.

Table 1: Requirement Collection Table [57]

Requirement ID (to be filled by PO)	Description (to be filled by Stake-holder)	Functional/Non-Functional (to be filled by Stakeholder)	Stakeholder (to be filled by Stakeholder)	Priority (to be filled by PO)	Status (to be filled by PO)	Comments/Notes (to be filled by PO)	Last Updated (to be filled by PO)

Each row in the table corresponds to a unique requirement. The first field, the "Requirement ID," is reserved for the PO to assign a unique identifier to each requirement once it has been submitted. This ensures proper traceability and management of requirements across various phases of the project. Following this, the "Description" field is intended for stakeholders to provide a detailed explanation of what they expect the system to do. It emphasizes the need for precision, encouraging stakeholders to specify whether the requirement pertains to a particular system behavior or performance characteristic.

The table also includes a "Functional/Non-Functional" column, where stakeholders are asked to categorize their requirement. If the requirement pertains to what the system should do, it is considered functional. Non-functional requirements, on the other hand, describe how the system performs in terms of attributes such as speed, reliability, or security. To accommodate any uncertainties, the stakeholders are informed that they can leave this classification to be determined during subsequent discussions with the PO, should they find it difficult to distinguish between the two.

Another key component is the "Stakeholder" field, where individuals or teams responsible for submitting the requirement are identified. This enables clear accountability and provides a way to trace the origins of each requirement back to its respective department or stakeholder.

[57] Own illustration

To ensure that stakeholders understand how to complete their portion of the table, a comprehensive guidance section follows the table. This guidance elaborates on each field that needs to be filled, providing examples and clarifications. It emphasizes the importance of clarity in the description, guiding stakeholders to avoid ambiguity, and includes explanations of both functional and non-functional requirements. Additionally, the guidance outlines how to identify themselves as stakeholders, ensuring that their input is properly documented.

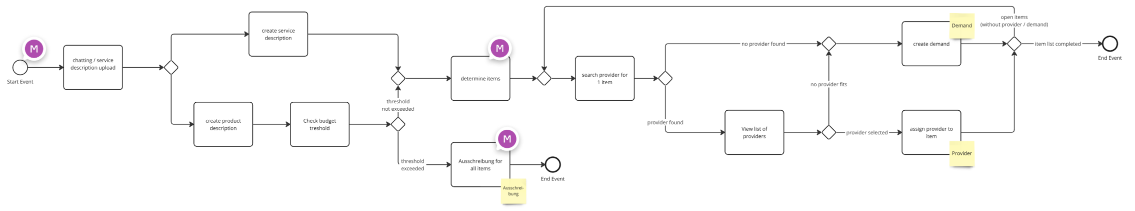
The remainder of the table, including fields such as "Priority," "Status," "Comments/Notes," and "Last Updated," is reserved for the PO to complete after the requirements have been collected. These fields will be used to prioritize, track the progress, and document any changes or updates to each requirement, ensuring that the project remains aligned with both stakeholder expectations and project constraints as it evolves.

This structure ensures a well-organized process for gathering, documenting, and managing system requirements, fostering collaboration between the stakeholders and the PO.

4.2 Requirement Elicitation

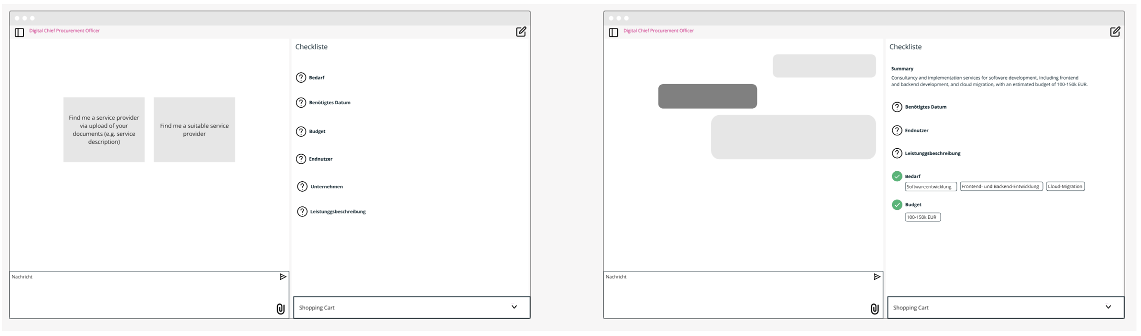
During the Requirement Elicitation phase, a combination of structured workshops and ad-hoc meetings via Microsoft Teams played a central role in defining both the system requirements and the associated business processes. These sessions provided a platform for stakeholders, domain experts, and the PO to collaborate closely, ensuring that the chatbot's functionalities would align with the operational needs of the procurement process. Throughout these meetings, key requirements were identified and discussed in detail, laying the groundwork for the chatbot's behavior, such as understanding user needs, searching through supplier catalogs, and facilitating procurement requests. A critical outcome of these discussions was the definition of core business processes, which were visualized in Miro to provide a clear understanding of how the chatbot would integrate into existing workflows. One example business process can be found in figure 1. By mapping out these processes collaboratively, the stakeholder and PO was able to capture the procedural steps the chatbot would follow, ensuring that both functional and non-functional requirements were well understood and aligned with the organizational objectives.

Figure 1: Defined Business Process in Miro [58]



As these discussions progressed, certain requirements emerged as more complex or abstract, making them difficult to fully articulate through conversation alone. In these cases, prototyping became an essential tool for refining and simplifying these requirements. The Miro mockups, 2 examples shown in figure 2, provided an early-stage visualization of how users would engage with the system, focusing on key interaction points, such as the user inputting a request, the chatbot interpreting the need, and the subsequent search of supplier catalogs for relevant offerings. All Miro Screen Mock-Ups can be found in Appendix 2.1. These early designs enabled stakeholders to visualize the flow of operations and provided a concrete foundation for discussions, allowing for immediate feedback and refinement. The outcome of these discussions clarified the scope of the chatbot's tasks and helped prioritize functionalities, ensuring that the system would meet the diverse requirements of both end-users and the internal procurement processes.

Figure 2: Screen Mock-Ups in Miro [59]

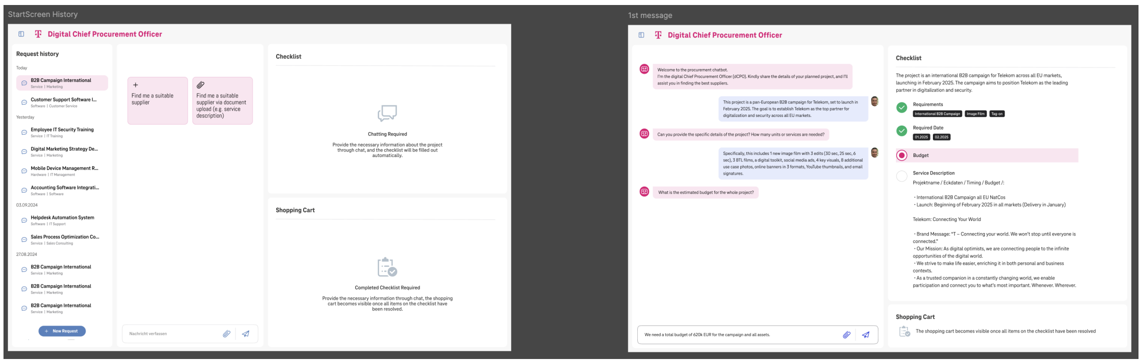

Digital Chief Procurement Officer
Find me a service provider via upload of your documents (e.g. service description)
Find me a suitable service provider
Checkliste
Bedarf
Benötigtes Datum
Budget
Endnutzer
Unternehmen
Leistunggsbeschreibung
Nachricht
Shopping Cart
Digital Chief Procurement Officer
Checkliste
Summary
Consultancy and implementation services for software development, including frontend and backend development, and cloud migration, with an estimated budget of 100-150k EUR.
Benötigtes Datum
Endnutzer
Leistunggsbeschreibung
Bedarf
Softwareentwicklung
Frontend- und Backend-Entwicklung
Cloud-Migration
Budget
100-150k EUR
Nachricht
Shopping Cart

Building on these initial mockups, the project transitioned to more detailed prototyping in Figma. The initial concepts developed in Miro were translated into high-fidelity interactive designs, offering a more precise depiction of the user interface and interaction flows. The Figma prototypes, examples shown in figure 3 and figure 4, allowed stakeholders to experience a more realistic representation of the chatbot, including how it would handle user queries, present search results from supplier catalogs, and guide the user through the

[58] Own illustration
[59] Own illustration

selection and checkout process. All Figma Screen Prototypes can be found in Appendix 2.2. This phase proved critical in refining the system's requirements, as the visual and interactive nature of the Figma screens enabled stakeholders to provide targeted feedback on specific elements of the interface, including the chatbot's responses, the organization of data, and the overall user experience. As the designs became more detailed, the stakeholder was able to refine aspects such as response times, data handling, and catalog integration, ensuring that the chatbot's functionalities aligned with both the technical capabilities of the system and the users' expectations.

Figure 3: Screen Prototype in Figma (Lightmode) [60]


StartScreen History
Digital Chief Procurement Officer
Request history
Today
B2B Campaign International
Service | Marketing
Customer Support Software I...
Software | Customer Service
Yesterday
Employee IT Security Training
Service | IT Training
Digital Marketing Strategy De...
Service | Marketing
Mobile Device Management R...
Hardware | IT Management
Accounting Software Integrati...
Software | Software
03.09.2024
Helpdesk Automation System
Software | IT Support
Sales Process Optimization Co...
Service | Sales Consulting
27.08.2024
B2B Campaign International
Service | Marketing
B2B Campaign International
Service | Marketing
B2B Campaign International
Service | Marketing
New Request
Find me a suitable supplier
Find me a suitable supplier via document upload (e.g. service description)
Nachricht verfassen
Checklist
Chatting Required
Shopping Cart
Completed Checklist Required
1st message
Digital Chief Procurement Officer
Checklist
Requirements
International B2B Campaign
Image Film
Tag-on
Required Date
01.2025
02.2025
Budget
Service Description
Shopping Cart

Provide the necessary information about the project through chat, and the checklist will be filled out automatically.

Provide the necessary information through chat, the shopping cart becomes visible once all items on the checklist have been resolved.

Welcome to the procurement chatbot.
I'm the digital Chief Procurement Officer (dCPO). Kindly share the details of your planned project, and I'll assist you in finding the best suppliers.

This project is a pan-European B2B campaign for Telekom, set to launch in February 2025. The goal is to establish Telekom as the top partner for digitalization and security across all EU markets.

Can you provide the specific details of the project? How many units or services are needed?

Specifically, this includes 1 new image film with 3 edits (30 sec, 25 sec, 6 sec), 3 BTL films, a digital toolkit, social media ads, 4 key visuals, 8 additional use case photos, online banners in 3 formats, YouTube thumbnails, and email signatures.

What is the estimated budget for the whole project?

We need a total budget of 620k EUR for the campaign and all assets.

The project is an international B2B campaign for Telekom across all EU markets, launching in February 2025. The campaign aims to position Telekom as the leading partner in digitalization and security.

Projektname / Eckdaten / Timing / Budget /:

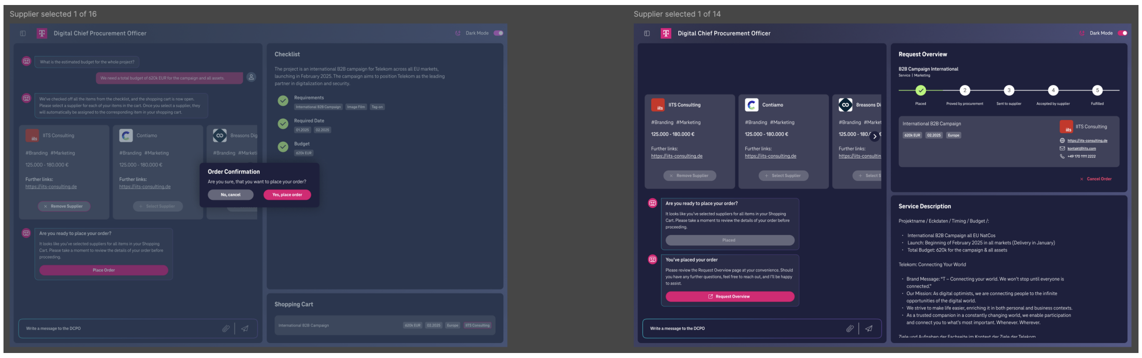
- International B2B Campaign all EU NatCos
- Launch: Beginning of February 2025 in all markets (Delivery in January)

Telekom: Connecting Your World

- Brand Message: "T – Connecting your world. We won't stop until everyone is connected."
- Our Mission: As digital optimists, we are connecting people to the infinite opportunities of the digital world.
- We strive to make life easier, enriching it in both personal and business contexts.
- As a trusted companion in a constantly changing world, we enable participation and connect you to what's most important. Whenever. Wherever.

The shopping cart becomes visible once all items on the checklist have been resolved

Figure 4: Screen Prototype in Figma (Darkmode) [61]



What is the estimated budget for the whole project?

We need a total budget of 620k EUR for the campaign and all assets.

We've checked off all the items from the checklist, and the shopping cart is now open. Please select a supplier for each of your items in the cart. Once you select a supplier, they will automatically be assigned to the corresponding item in your shopping cart.

Are you ready to place your order?
It looks like you've selected suppliers for all items in your Shopping Cart. Please take a moment to review the details of your order before proceeding.

Are you sure, that you want to place your order?

The project is an international B2B campaign for Telekom across all EU markets, launching in February 2025. The campaign aims to position Telekom as the leading partner in digitalization and security.

You've placed your order
Please review the Request Overview page at your convenience. Should you have any further questions, feel free to reach out, and I'll be happy to assist.

Projektname / Eckdaten / Timing / Budget /:

- International B2B Campaign all EU NatCos
- Launch: Beginning of February 2025 in all markets (Delivery in January)
- Total Budget: 620k for the campaign & all assets

Telekom: Connecting Your World

- Brand Message: "T – Connecting your world. We won't stop until everyone is connected."
- Our Mission: As digital optimists, we are connecting people to the infinite opportunities of the digital world.
- We strive to make life easier, enriching it in both personal and business contexts.
- As a trusted companion in a constantly changing world, we enable participation and connect you to what's most important. Whenever. Wherever.

Ziele und Aufgaben der Fachseite im Kontext der Ziele der Telekom

Once the Figma designs were validated, the project moved to the development of functional builds using Vue. These builds marked the first instance of the system in a live environment, where the chatbot could be tested in real-time against actual user inputs. The Vue builds, examples shown in figure provided the opportunity to verify that the system met the documented requirements in practice, with users engaging directly with the chatbot interface to simulate procurement scenarios. During these tests, it became evident whether the chatbot's decision-making processes were functioning as expected and whether its

[60] Own illustration
[61] Own illustration

performance, particularly in searching supplier catalogs and handling user requests, met the defined non-functional requirements. The feedback gathered during this phase led to further adjustments, with particular focus on optimizing the chatbot's responsiveness and ensuring the smooth integration of supplier data into the system's workflow.

The culmination of the Requirement Elicitation phase was the consolidation of all gathered requirements into a structured document that served as the definitive guide for the project's development. This document captured the full scope of the system's functional and non-functional needs, including the precise behavior of the chatbot when interacting with users, the expected performance standards, and the overall user experience considerations. This requirements document evolved alongside the project, incorporating feedback from stakeholders at each stage—from the Miro mockups to the Figma prototypes and finally the Vue builds—ensuring that the requirements were fully traceable throughout the entire development process. The result of this phase was a clear, comprehensive set of requirements that not only reflected the needs of the users but also aligned with the technical constraints and objectives of the system, providing a solid foundation for the subsequent stages of development.

4.3 Requirement Analysis and Negotiation

The Requirement Analysis and Negotiation phase focused on refining the gathered requirements and resolving any conflicts or ambiguities that arose during the elicitation phase. This stage was essential for ensuring that the system would meet both the technical constraints and the diverse needs of the various stakeholders involved in the project. The process of analysis and negotiation was conducted primarily through regular weekly meetings with stakeholders and bi-weekly sessions with senior management, ensuring that the project stayed aligned with both operational expectations and strategic objectives.

The weekly meetings with stakeholders formed the backbone of the Requirement Analysis process. These sessions, which brought together representatives from the procurement teams, technical staff, and other key user groups, provided a structured environment for ongoing discussions about the system's requirements. During these meetings, the feedback gathered from the prototyping phases was reviewed in detail, with particular attention given to how the system's functionalities were evolving in response to the requirements initially defined in the workshops.

A key aspect of these weekly sessions was the continuous prioritization of requirements. The stakeholders provided input on which features needed to be refined or developed first, helping the PO maintain focus on core functionalities, such as the chatbot's ability to interpret and act on user requests. These discussions also enabled the PO to identify any gaps in the requirements or areas where further clarification was needed, ensuring that the chatbot's functionality would align with the intended procurement processes.

In addition to prioritization, the weekly meetings were also used to resolve conflicts between competing requirements. As different stakeholders had varying expectations of how the chatbot should function, these sessions served as a forum for negotiation. For example, there were often trade-offs between modern design and user accessibility, or between performance and cost. By bringing these issues to the forefront in a collaborative setting, the PO was able to reach consensus on the best approach to take. These discussions were particularly valuable in clarifying the chatbot's core functionalities, such as determining the appropriate level of detail the chatbot should provide when responding to user queries and how it should handle requests that were not well-defined.

Through these weekly discussions, the PO was able to continuously refine the system's requirements, ensuring that they remained both feasible and aligned with the overall project goals. As new insights emerged, they were integrated into the requirements document, ensuring that the evolving needs of the stakeholders were captured and could be tracked throughout the development process.

In parallel with the weekly stakeholder sessions, bi-weekly meetings with senior management were held to ensure that the project remained aligned with the broader strategic objectives of the organization. These meetings provided a higher-level perspective on the system's development, focusing on ensuring that the chatbot's functionalities would not only meet the immediate operational needs but also support the company's long-term goals for procurement efficiency and digital transformation.

The bi-weekly management meetings were critical in securing approvals for key decisions made during the Requirement Analysis phase. For instance, as the system's development progressed and the trade-offs between performance and cost became more apparent, it was essential to have senior management's input to determine the acceptable levels of investment in system performance. These meetings also provided an opportunity to discuss the prioritization of high-impact features, ensuring that the project remained focused on delivering value in line with the company's strategic initiatives.

One important aspect of the bi-weekly management meetings was the focus on risk mitigation. As potential risks were identified during the stakeholder meetings, such as delays in catalog integration or challenges related to system scalability, these issues were brought to the attention of senior management. This allowed the PO to make informed decisions about how to allocate resources and manage risks effectively, ensuring that the system development remained on track and that any potential bottlenecks could be addressed early.

The management meetings also served as a platform to reflect the progress made in the weekly stakeholder sessions. By reviewing the decisions made and the requirements refined during those discussions, senior management was able to provide feedback and ensure that the system's development remained consistent with the organization's vision. This continuous feedback loop between the weekly stakeholder sessions and the bi-weekly management reviews ensured that the system's development remained agile and responsive to both operational and strategic needs.

The iterative nature of the Requirement Analysis and Negotiation phase allowed for continuous refinement of the chatbot's requirements. The weekly stakeholder meetings provided detailed input on the system's functionality and performance, while the bi-weekly management meetings ensured that these decisions aligned with the organization's broader goals. Throughout this phase, the requirements document was continuously updated, reflecting the outcomes of each discussion and ensuring that every decision was documented and traceable.

4.4 Requirement Validation

The Requirement Validation phase was critical in ensuring that the requirements gathered and refined in previous phases were accurate, complete, and feasible. The goal was to verify that the functional and non-functional needs of the system were well understood and aligned with stakeholder expectations. This phase relied on interactive tools such as Figma prototypes to visualize and simulate the chatbot's core functionalities. In addition, initial backend conversations were tested during this phase to verify key technical aspects of the chatbot's processing logic.

The Figma prototypes played a crucial role in validating the system's design and behavior. These prototypes allowed stakeholders to interact with a detailed and high-fidelity representation of how the chatbot would operate in practice. They offered a clear view of the user interface, navigation, and interaction flows, simulating the chatbot's procurement process without requiring full backend integration or live data.

Stakeholders were invited to explore these prototypes, providing feedback on whether the user experience and functionality met the documented requirements. The prototypes simulated key features such as the chatbot's response to user inputs, search functionality within supplier catalogs, and the overall flow from request initiation to the presentation of recommendations. This validation helped identify any areas where the chatbot's design needed to be adjusted to better meet business objectives. Based on stakeholder feedback, improvements were made to enhance usability and ensure the chatbot's interaction with users was intuitive and aligned with the intended procurement processes.

In addition to validating the user interface through Figma prototypes, early backend conversations were also validated in this phase. This validation focused on ensuring that the backend could interpret user input and return appropriate responses, even though the system was not yet fully integrated or using live data. By simulating conversations between the chatbot and users, the PO was able to validate that the backend logic was functioning correctly and that the chatbot could handle simple dialogues.

These initial backend tests provided valuable insights into how the chatbot's NLP would work in practice, allowing the PO to refine the backend's ability to interpret requests and provide relevant responses. Testing these conversations during the validation phase ensured that both the frontend prototypes and backend logic were progressing in alignment with the documented requirements.

Throughout the Requirement Validation phase, regular review sessions were conducted with stakeholders to gather ongoing feedback. These sessions were essential for ensuring that the evolving prototypes and backend functionalities were meeting expectations. The interactive nature of these reviews allowed stakeholders to provide detailed feedback on the chatbot's behavior, leading to continuous refinements of both the design and backend logic.

Key areas of focus during these reviews included validating the accuracy of the chatbot's responses, ensuring smooth navigation, and confirming that the user interface was intuitive. The feedback provided by stakeholders was incorporated into subsequent iterations of the prototypes and backend logic, ensuring that the system continued to evolve in line with the project's goals.

After successfully validating the requirements through both the Figma prototypes and early backend tests, the PO was ready to move into the actual development phase. The transition marked the end of the Requirement Validation phase, signaling that the requirements had been thoroughly vetted and were ready for implementation. With the core functionalities validated and feedback incorporated, the development of the full prototype, including both the backend and frontend systems, began using test data.

4.5 Requirement Management

The Requirement Management phase focused on maintaining control over the evolving requirements throughout the project. As new insights emerged from stakeholder feedback and validation sessions, it was essential to manage changes systematically to ensure that all modifications were properly documented and traceable. A structured approach was adopted to track every change in the requirements, ensuring that the PO and stakeholders remained aligned on any updates or shifts in priorities. This process helped avoid scope creep and ensured that only approved changes were implemented, safeguarding the integrity of the original project objectives.

Regular reviews of the requirements document were conducted to reflect the evolving nature of the project, particularly as feedback from the validation phase led to refinements in the system's functionality. By maintaining up-to-date records of all changes, the PO was able to ensure consistency between the original requirements and the final deliverables. This approach ensured that any changes were justified, reviewed, and incorporated without disrupting the overall project flow, providing a robust framework for managing the dynamic aspects of the chatbot's development. The complete set of these finalized documents can be found in Appendix 2.3 for further reference.

5 Prototyping

5.1 Functional Selection

5.1.1 Identifying Core Functionalities Based on User Needs

In the functional selection phase, several key features were identified and extracted from the requirements gathered during the Requirements Engineering process in 4. These features were chosen based on their potential to validate the core interaction and functionality of the chatbot system, ensuring that the prototype effectively addresses the primary needs of the stakeholders. The focus was placed on fundamental aspects that would provide a meaningful demonstration of the system's capabilities without requiring full-scale implementation of the entire product.

The first selected feature was the basic chat functionality, which forms the core of the user interaction with the system. This functionality allows users to communicate their needs through a conversational interface, enabling the chatbot to capture and process input in real-time. Given that the chatbot is designed to guide users through the process of finding products and services, this feature is essential for testing how well the system can interpret user queries.

Secondly, the prototype incorporates a chat history feature, which ensures that users can review their previous interactions with the chatbot. This functionality not only improves the user experience by providing continuity but also allows for finishing started orders or asking questions to placed orders, enhancing the overall usability of the system.

A crucial element of the system is information extraction from the conversation, where the chatbot analyzes the user's input to identify specific needs. This feature is pivotal to understanding the customer's requirements, whether for hardware, software, or services, and plays a central role in shaping the subsequent actions of the system.

The fourth selected feature is the matching of the extracted customer need with the supplier catalog. This functionality ensures that the system can search the internal catalogs and identify the most relevant products or services that meet the user's needs. This aspect of the prototype will demonstrate the effectiveness of the backend logic in delivering accurate and relevant results from the catalog, allowing users to make informed decisions.

Lastly, order tracking was included as a feature to allow users to monitor the status of their requests after placing an order. While not fully implemented in the prototype, this functionality simulates the experience of tracking an order's progress, which is a vital part of the user journey in the final system.

By focusing on these selected features, the prototype can provide a realistic representation of how the final system will handle key user interactions, offering valuable insights into both technical feasibility and user experience. The selected features are not only aligned with stakeholder requirements but also serve as a solid foundation for evaluating the prototype's effectiveness in fulfilling the system's objectives.

5.1.2 Breaking Down Core Functionalities into Specific Features

After identifying the core functionalities that the system needs to fulfill, the next step is to break these down into more specific features. Each of the basic functionalities, such as chat interaction, information extraction, and order tracking, can be addressed with a combination of technical components and architectural decisions. By doing so, we ensure that the prototype meets the user needs effectively while laying the groundwork for future development.

The first basic functionality, enabling chat interaction, requires more than just a user interface where users can type messages. For this interaction to be meaningful, there must be a way to manage the exchange of information between the frontend (where the user interacts) and the backend (where the logic and data processing occur). To achieve this, the system should implement a RestAPI, which will allow seamless communication between the chatbot interface and the backend services. This ensures that user inputs are received and processed efficiently, and responses are sent back in real-time , creating a dynamic and responsive user experience. The RestAPI will act as the backbone for handling asynchronous requests, ensuring that the system remains scalable and flexible as more complex features are added.

For the chat history feature, the system needs to ensure that users can retrieve past conversations, allowing them to continue where they left off. This is essential for maintaining continuity, especially in scenarios where a user might need to revisit an earlier request or clarify an ongoing conversation. To address this need, a PostgreSQL database should be used. PostgreSQL is well-suited for this task due to its ability to handle structured data and complex queries , ensuring that chat history can be stored and accessed efficiently. This database solution not only supports the retrieval of individual conversations but also allows the system to manage user sessions, ensuring that each user can securely access their personal chat history.

The ability to extract information from the conversation is another key feature that requires a more granular approach. The chatbot needs to understand the specific needs expressed by the user—for example whether they are looking for hardware, software, or services. This requires an intelligent system capable of processing natural language and extracting relevant information. To accomplish this, the system should utilize Generative AI (GenAI), which will be integrated through an OpenAPI model. By leveraging Artificial Intelligence (AI), the chatbot will be able to perform more advanced tasks, such as recognizing the user's intent, identifying key terms, and extracting actionable information from the conversation. This goes beyond simple keyword matching, allowing the system to offer personalized and accurate results based on the user's expressed needs.

Next, the system must provide users with the ability to retrieve their assigned chats after logging in. This need arises from the requirement for users to be able to access their ongoing conversations from different devices or after a session ends. To meet this need, a login page should be implemented where users can authenticate themselves and load their assigned chats from the database. Upon successful login, the system should query the PostgreSQL database to retrieve the user's active or previous conversations, allowing for seamless continuation of their interactions with the chatbot. This feature ensures that conversations are not lost and users can interact with the system across multiple sessions. However, this feature should only be simulated in the prototype, as there will be a Single Sign-On (SSO) in the future, but this may not be integrated without an internal Group Privacy Security Assessment (PSA).

In addition to these user-facing functionalities, the system must remember user settings, such as login states and preferences, to provide a seamless experience. For example, users might choose between dark and light modes or prefer not to re-enter their login credentials after refreshing the page. To ensure these preferences are maintained, the system should utilize cookies. Cookies allow the system to store small pieces of information about the user's preferences and login state, ensuring that these settings are remembered across different sessions without requiring the user to input them repeatedly.

Finally, beyond simple chat interactions, there is a need for the system to handle more complex user inputs, such as file uploads. Users might need to upload documents—such as service descriptions—so the system can extract relevant information. This requires the system to not only process unstructured text but also to extract actionable data from uploaded files. To address this need, a file upload functionality should be integrated into the system. This feature will allow users to upload documents, and the system, through OpenAIs model, can analyze the content and extract the necessary information, such as service descriptions or specifications. This expands the chatbot's functionality beyond simple text interactions, allowing it to handle more complex user requirements and data inputs.

By breaking down these core functionalities into specific technical solutions, the prototype will be able to address the key user needs effectively. The proposed technical features—such as the RestAPI for communication—ensure that the system is both flexible and robust enough to meet its intended purpose. Each feature plays a crucial role in creating a responsive, intelligent, and user-friendly system that will serve as a foundation for the final product.

5.2 Construction

5.2.1 Frontend

5.2.1.1 Project Folder Structure

5.2.1.2 Project Setup

The project setup involves the initialization and configuration of the core Vue.js application, integrating essential plugins and defining the main structure of the application. This setup includes several key files: `main.ts`, `App.vue`, `index.ts` (in the plugins folder), and `index.vue`. These files work together to create a consistent and well-organized foundation for the chatbot system.

The `main.ts` file is responsible for bootstrapping the Vue.js application, registering essential plugins, and mounting the root component. This entry point initializes the app with Vue.js and applies global configurations.

Code 1 demonstrates how the application is initialized and mounted.

Code 1: Bootstrapping the Application (`main.ts`)

```
import './styles/main.scss'

// Plugins
import { registerPlugins } from '@/plugins'
import { VueQueryPlugin } from '@tanstack/vue-query'

// Components
import App from './App.vue'

// Composables
import { createApp } from 'vue'

```

```
const app = createApp(App)

registerPlugins(app)
app.use(VueQueryPlugin)

app.mount('#app')
```

The project imports a global stylesheet (`main.scss`) to ensure consistent styling across all components. The `registerPlugins` function, imported from the `plugins` folder, sets up essential plugins such as Vuetify and Vue Router. The `VueQueryPlugin` plugin is used for data fetching and caching, improving the app's efficiency and reactivity in managing API data. Finally, the root component (`App.vue`) is mounted to the `#app` element in the HTML, starting the application.

The `App.vue` file defines the overall layout of the application using Vuetify's layout components. It serves as the root component that houses all other views and components.

The template in Code 2 illustrates how Vuetify components are used to create a structured layout.

Code 2: Setting Up the Root Layout (`App.vue`)

```
<template>
  <v-app>
    <v-main>
      <router-view />
    </v-main>
  </v-app>
</template>
```

Vuetify's `v-app` and `v-main` components create a structured layout , ensuring that the application adheres to material design principles. The `<router-view />` dynamically renders the current route's component based on the Vue Router configuration.

The `plugins/index.ts` file contains the `registerPlugins` function, which is used to register essential plugins such as Vuetify and Vue Router with the Vue.js application. This function centralizes plugin configuration, making it easier to manage.

The following code snippet shows how plugins are registered using `registerPlugins`, as shown in Code 3.

Code 3: Registering Essential Plugins (`plugins/index.ts`)

```
import vuetify from './vuetify'
import router from '../router'

// Types
import type { App } from 'vue'
```

```

export function registerPlugins(app: App) {
  app.use(vuetify).use(router)
}
```

The function takesa Vue app instance as an argument and applies the necessary plugins using `app.use()`. This pattern keeps the plugin registration clean and consistent across the application.

The `index.vue` file serves as a key layout component that defines the main structure and user interface elements of the application. It includes the navigation bar, app bar, and the main container where chat functionality and other key features are rendered.

The app bar at the top of the page, shown in Code 4, provides access to the sidebar, theme switcher, and user login dialog.

Code 4: Setting Up the App Bar (`index.vue`)

```
<v-app-bar prominent :elevation="0" color="background" class="pr-3">
  <template #prepend>
    <v-btn icon="$sidebar" size="small" color="secondary" variant="text" @click.stop="
        drawer = !drawer" />
  </template>

  <v-toolbar-title>
    <span class="text-page-header-color bg-logo-background text-center pa-2 mr-4">
      <v-icon icon="$telekom" size="xs" class="my-2" />
    </span>
    <span class="text-page-header-color font-weight-bold">Digital Chief Procurement
        Officer</span>
  </v-toolbar-title>

  <v-spacer></v-spacer>
  <div class="d-flex align-center">
    <ThemeSwitcher />
    <UserLoginDialog v-model="showUserLoginDialog" :persistend="
        showUserLoginDialogPersistend" />
  </div>
</v-app-bar>
```

The app bar includes a sidebar toggle, the project title, and user-related features such as the theme switcher and login dialog.

The main container divides the page into two sections: one for the chat and the other for displaying the checklist, service description, and shopping cart. This layout provides an intuitive and organized interface, as shown in Code 5.

Code 5: Main Container Layout (`index.vue`)

```
<v-main>
```

```
  <div class="main-container d-flex justify-between ga-3 w-100 no-wrap">
    <div class="w-50 rounded-lg" :class="{ 'ml-3': drawer }">
      <Chat />
    </div>
    <div class="w-50 rounded-lg d-flex flex-column ga-3">
      <Checklist />
      <ServiceDescription />
      <ShoppingCart />
    </div>
  </div>
</v-main>
```

This layout ensures that the chat remains prominent, while additional information is organized on the right side, enhancing the user's interaction experience.

5.2.1.3 API Handling

The API handling is an essential part of the chatbot system, providing interaction between the frontend and the backend. The API connects various services, such as case management, message sending, and supplier handling, to ensure that the system operates seamlessly. This section covers on one side the most important functions from the `api.ts`, `v1.json`, and `v1.d.ts` files, which define the API's structure, types, and requests. On the other side, there are three composables in the project that play a significant role in facilitating API interactions: `useApiClient.ts`, `useCasesApi.ts`, and `useMessagesApi.ts`. These composables encapsulate API-related logic, making it easier to reuse and manage the interactions between the frontend and the backend.

The `api.ts` file defines core types and utilities that are crucial for API interactions. It leverages TypeScript's type system to define API schema models, ensuring that all requests and responses are strongly typed and consistent throughout the application.

The following snippet defines the types for `Case`, `State`, and `Message`, as shown in Code 6.

Code 6: Core API Types (`api.ts`)

```
export type Case = components['schemas']['Case']
export type State = ArrayType<NonNullable<components['schemas']['Case']['state']>>
export type Message = components['schemas']['Message'] & {
  loading?: boolean
}
```

The `Case`, `State`, and `Message` types are derived from the OpenAPI schema (defined in the `v1.json` file). These types ensure that every interaction with the API adheres to the structure defined by the backend. The `Message` type is extended to include a `loading` property , allowing the frontend to track when a message is being processed, providing better feedback to the user.

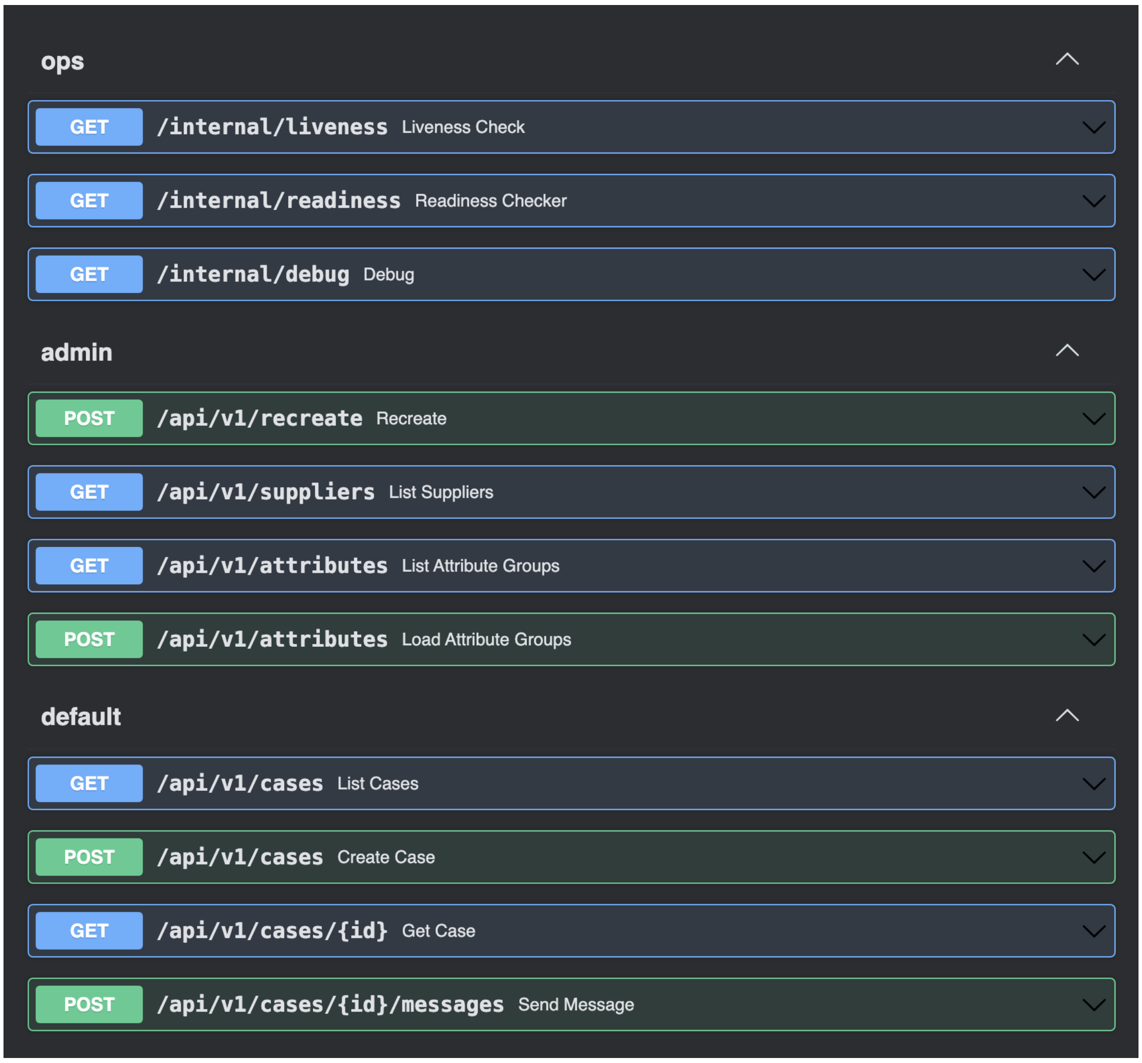
The `v1.json` file represents the OpenAPI schema forthe backend. It defines all available API endpoints, their parameters, and their expected responses. This structure is crucial for understanding how the frontend interacts with the backend.

Code 47, stored in Appendix 3.1, defines the `/api/v1/cases` endpoint for listing and creating cases.

This endpoint supports both listing (GET) and creating (POST) cases. The GET method allows filtering by `user_id` and limiting the number of cases retrieved. The POST method requires a `CreateCaseRequest` body to create a new case. Each operation has a unique `operationId` (e.g., `list_cases_api_v1_cases_get`), which is used to reference and document the API call.

Because OpenAPI is used, the API Endpoints can be shown in the schema from Swagger as shown in Figure 5.

Figure 5: Swagger UI [62]


ops

GET /internal/liveness Liveness Check

GET /internal/readiness Readiness Checker

GET /internal/debug Debug

admin

POST /api/v1/recreate Recreate

GET /api/v1/suppliers List Suppliers

GET /api/v1/attributes List Attribute Groups

POST /api/v1/attributes Load Attribute Groups

default

GET /api/v1/cases List Cases

POST /api/v1/cases Create Case

GET /api/v1/cases/{id} Get Case

POST /api/v1/cases/{id}/messages Send Message

This made it easy to test the API Endpoints without implementing them directly in the code.

The `v1.d.ts` file defines the TypeScript types generated from the OpenAPI schema, ensuring that all API requests and responses are typed correctly. This provides strong typing for all interactions with the API, helping prevent runtime errors and ensuring that the frontend and backend stay in sync.

The following snippet shows the type definition for the `Case` schema, as demonstrated in Code 7.

Code 7: Case Schema Definition (`v1.d.ts`)

```
export interface components {
  schemas: {
    Case: {
```

[62] Own illustration

```
      /**
      * Id
      * Format: uuid
      */
      id: string;
      /** State */
      state?: components["schemas"]["StateAttributeGroup"][];
      /** Messages */
      messages?: components["schemas"]["Message"][];
      /**
      * Created At
      * Format: date-time
      */
      created_at: string;
      /** Title */
      title?: string | null;
      case_type?: components["schemas"]["SystemCaseTypeEnum"] | null;
    };
  }
}
```

This schema defines the structure of a `Case` object, which includes fields such as `id` , `state`, `messages`, `created_at`, and optional fields like `title` and `case_type`. The `state` field contains an array of `StateAttributeGroup` objects , while the `messages` field contains an array of `Message` objects, demonstrating the complexity of the data model.

The `useApiClient.ts` file defines a function that sets up and returns an API client. It uses the `openapi-fetch` library to create a client that interacts with the defined OpenAPI schema. The client is configured with a base URL, and headers can be added as needed. This client serves as the primary way to make API calls throughout the application.

Code 8 demonstrates how the `useApiClient` function is defined.

Code 8: Setting up the API Client (`useApiClient.ts`)

```
export const useApiClient = () => {
  return createClient<paths>({
    baseUrl: import.meta.env.VITE_API_BASE_URL,
    headers: {
      // Authorization: `Basic ${import.meta.env.VITE_API_CREDENTIALS}`
    }
  })
}
```

The `useApiClient` function encapsulates the configuration of the API client, allowing other composables to easily import and use it. This approach keeps the API setup centralized and manageable, making it easy to configure base URLs and headers in one place.

The useCasesApi.ts file defines the composable for interacting with cases-related API endpoints. It utilizes the API client to fetch and create cases and employs vue-query for managing data caching, mutation, and state.

The following snippet shows how the findAll function is implemented using vue-query, as demonstrated in Code 9.

Code 9: Fetching All Cases (useCasesApi.ts)

```
const findAll = useQuery({
  queryKey: ['cases'],
  queryFn: async () => {
    return (
      await client.GET('/api/v1/cases', {
        params: {
          query: {
            user_id: userStore.userId
          }
        }
      })
    ).data?.cases
  },
  ...defaultQueryOptions
})
```

This function uses vue-query to fetch all cases for a specific user. It defines a useQuery with a unique queryKey (['cases']) and a queryFn that performs a GET request to the /api/v1/cases endpoint, passing the user's ID as a query parameter. The default options are applied to the query to manage retry behavior and error handling.

The useCasesApi composable also includes functions for creating a case and fetching a case by its ID. It leverages Vue's reactivity to make these operations seamless within the application, allowing components to respond automatically to changes in the data.

The useMessagesApi.ts file defines the composable for interacting with message-related API endpoints. It primarily handles sending messages for a given case and fetching the list of messages. Similar to useCasesApi.ts, this composable uses the shared useApiClient and integrates with vue-query for data management.

The create function in useMessagesApi sends a new message to the backend by making a POST request to /api/v1/cases/{id}/messages, with the message content as the request body. This function allows the chatbot to send user-generated messages and system messages effectively.

The composable useMessagesApi.ts also includes a findAll function to fetch all messages associated with a specific case. This function is accomplished by extending the `findById` method from `useCasesApi`, ensuring consistency in handling API interactions across the project.

5.2.1.4 User Login and Session Handling

This section focuses on the user login and session management, enabling users to authenticate themselves by entering a User Identification (ID). The system leverages Vue.js components, Vuetify elements, and reactive state management to build an intuitive login dialog. This feature ensures that only authenticated users can interact with the chatbot, thus personalizing the user experience and maintaining the session across multiple interactions.

The login component is implemented using TypeScript (TS), providing type safety and enhanced tooling support, which is beneficial for large-scale projects. Below is a breakdown of the code's key functionalities.

The following code snippet imports essential modules and components used throughout the login process. See Code 10.

Code 10: Importing Dependencies (`UserLoginDialog.vue`)

```
import { userStore } from '@/stores/user-store'
import { onUpdated, ref, useTemplateRef } from 'vue'
import { VTextField } from 'vuetify/components/VTextField'
```

The `userStore` is imported from a centralized state management store to handle user-specific data, such as the user ID. The `onUpdated`, `ref`, and `useTemplateRef` functions are imported from Vue, essential for reactive programming, where component properties update automatically when the state changes. The `VTextField` from Vuetify is used to render the input field in the login form.

Reactive references and validation rules are established in Code 11 to handle user input.

Code 11: Reactive Variables and Validation (`UserLoginDialog.vue`)

```
const model = defineModel<boolean>()
defineProps<{ persistend: boolean }>()

const userIdInput = useTemplateRef<VTextField>('userIdInput')
const userId = ref<string>()
const userIdRules = ref<((v: string) => string | boolean)[]>([
(v: string) => (!!v && !!v.trim()) || 'User ID is required'
])
```

The `model` defines a reactive model to control the visibility of the login dialog, while the `persistend` prop determines whether the dialog can be closed without user interaction. The `userId` is a reactive reference holding the user input, while `userIdRules` ensures that the input is valid, requiring a non-empty string to maintain data integrity.

The `onUpdated` lifecycle hook, shown in Code 12, ensures the synchronization of the user ID with the session state stored in the `userStore`.

Code 12: Synchronizing State with Lifecycle Hooks (`UserLoginDialog.vue`)

```
onUpdated(() => {
userId.value = userStore.userId
})
```

This ensures that the correct user ID is displayed in the form if it has been previously set.

Code 13 handles the user login process.

Code 13: Handling User Login (`UserLoginDialog.vue`)

```
function login() {
if (!userId.value) return

userStore.login(userId.value)
model.value = false
}
```

The `login` function first checks if the `userId` field is populated. If so, the `userStore.login()` function is called to store the user ID for future interactions. After a successful login, the login dialog is closed, providing a seamless user experience.

The Hypertext Markup Language (HTML) template uses Vuetify components to create the login interface, binding state variables for reactivity and ensuring user input validation.

The login dialog is defined as follows in Code 14.

Code 14: Login Dialog (`UserLoginDialog.vue`)

```
<v-dialog v-model="model" max-width="400" :persistent="persistend">
<template #activator="{ props: activatorProps }">
<v-btn v-bind="activatorProps" variant="text" class="text-none">
<div v-if="userStore.userId">{{ userStore.userId }}</div>
<div v-else>Login</div>
<template #append>
<v-icon v-if="userStore.userId">mdi-account-check</v-icon>
<v-icon v-else>mdi-account-plus</v-icon>
</template>
</v-btn>
</template>
</v-dialog>
```

The `v-model` directive links the dialog's visibility to the `model` reactive variable. If the user is logged in (i.e., `userStore.userId` is set), the button displays the user ID; otherwise, it prompts the user to log in. Icons are dynamically displayed to indicate the login state.

The input field for the user ID is defined as shwon in Code 15.

Code 15: User ID Input Field (`UserLoginDialog.vue`)

```
<v-text-field
ref="userIdInput"
v-model="userId"
:counter="10"
:rules="userIdRules"
label="User ID"
@keyup.enter="login"
clearable
></v-text-field>
```

The `v-text-field` is bound to the `userId` reactive reference, with a 10-character limit enforced by the `counter` attribute. Validation rules are applied via `userIdRules` , ensuring that users cannot submit invalid IDs. The `@keyup.enter="login"` directive enables users to press "Enter" to submit the form, simplifying interaction.

The login button is implemented as follows. See Code 16.

Code 16: Login Button (`UserLoginDialog.vue`)

```
<v-btn @click="login" :disabled="!userId"> Login </v-btn>
```

The button is enabled only when a valid `userId` is present, preventing users from submitting the form without valid input.

User login and session handling are critical for managing how users access the chatbot system. It ensures that only authorized users can interact with the chatbot, which is essential for both security and personalization. Vuetify components provide a professional and responsive interface, while Vue's reactive system handles the dynamic nature of login states, ensuring a smooth user experience without unnecessary page reloads or delays.

5.2.1.5 Storing Information

This section focuses on how the chatbot system stores and manages key pieces of data related to user sessions and selected cases. Proper state management and persistence of information are crucial for maintaining a seamless user experience, especially when handling interactions across multiple sessions or refreshing the application. Two key pieces of state—the selected case and the user session—are managed using Vue's reactivity system. The state is stored locally, allowing the application to maintain important information between page loads.

The `selected-case.store.ts` file manages the state of the selected case throughout the chatbot session. It uses Vue's `reactive` function to create a reactive store that can be shared across components. This is essential for keeping the selected case in sync across different parts of the application.

The following code snippet shows how the `selectedCaseStore` is set up and managed, as demonstrated in 17.

Code 17: Setting up the Case Store (`selected-case.store.ts`)

```
export const selectedCaseStore = reactive<CaseStore>({
  case: {
    id: '',
    created_at: ''
  },
  setCase(caseData: Case) {
    this.case = caseData
  },
  clearCase() {
    this.case = {
      id: '',
      created_at: ''
    }
  }
})
```

In this snippet, `selectedCaseStore` is a reactive object that holds the current case state. This object provides essential methods to manage the state. The `setCase` method updates the case state with new data whenever a case is selected, ensuring that the correct data is displayed across components when switching between different cases during a chat session. Additionally, the `clearCase` method resets the case state, which is particularly useful when the user finishes working on a case or logs out.

By using a reactive store, the case state remains in sync across all components that rely on it, making the user experience consistent and dynamic.

The `user-store.ts` file handles the state of the user's session, including login and logout functionality. It uses Vue's reactive system in combination with the `useLocalStorage` utility from the `@vueuse/core` library to persist the user's `userId` across sessions.

The following code snippet shows how the `userStore` manages the user's login and session data, as demonstrated in 18.

Code 18: Managing the User Session (`user-store.ts`)

```
const storedUserId = useLocalStorage<string | undefined>('user-id', undefined)

export const userStore: User = reactive<User>({
  userId: storedUserId.value,
  login(userId: string) {
    storedUserId.value = userId
    this.userId = userId
  },
  logout() {
    storedUserId.value = undefined
    this.userId = undefined
  }
})
```

This code leverages Vue's reactive system in combination with the `useLocalStorage` utility from the `@vueuse/core` library to ensure that the user's `userId` persists across sessions. The implementation begins by initializing `storedUserId` from local storage, and this value is then used as the default for `userId` within the reactive store.

When the `login` method is triggered, the user's `userId` is stored both in the reactive state and in local storage, allowing other components to reactively update based on the user's login status. Conversely, when the `logout` method is executed, both the reactive state and local storage are cleared, ensuring that the session is properly ended when the user logs out.

This approach ensures that the user does not have to log in again after refreshing the page or closing and reopening the application, thanks to the persistence of the `userId` in local storage.

5.2.1.6 Basic Chat Functionality

The basic chat functionality enables users to interact with the chatbot, allowing both text-based communication and document uploads. This functionality is powered by five components: *Chat.vue* for managing the chat interface, *ChatInput.vue* for handling user input, *ChatMessage.vue* for rendering individual messages, *ChatHistory.vue* for rendering the conversation and *MessageLoadingAnimation.vue* for the loading animation. Each of these components is explained in detail below.

The `Chat.vue` component is responsible for managing user interactions with the chatbot, including sending messages and uploading files. It communicates with the backend via API calls and dynamically updates the chat history as messages are sent and received.

The structure incorporates composables for fetching data and managing state, as well as improved handling of file selection.

The following code snippet demonstrates the updated imports and setup for the component, which includes composables for managing API interactions and a reactive store for handling the selected case. This is shown in Code 19.

Code 19: Updated Imports and Setup (`Chat.vue`)

```
import { useCasesApi } from '@/composables/useCasesApi'
import { useMessagesApi } from '@/composables/useMessagesApi'
import { selectedCaseStore } from '@/stores/selected-case.store'
import { Message } from '@/types/api'
import _ from 'lodash'
import { computed, ref, useTemplateRef } from 'vue'
```

In this snippet, `useCasesApi` and `useMessagesApi` are custom composables that handle API calls for retrieving and managing case data and messages. The `selectedCaseStore` stores the currently selected case, ensuring that messages are linked to the correct context.

Several reactive variables are defined to track the state of the message being sent and the system's response, as shown in Code 20.

Code 20: Reactive Variables for Message Tracking (`Chat.vue`)

```
const pendingMessage = ref<Message | undefined>()
const responseLoadingMessage = ref<Message | undefined>()

const messagesApi = useMessagesApi(selectedCaseId)
const casesApi = useCasesApi()
```

The `pendingMessage` holds the message that the user is sending, while `responseLoadingMessage` tracks when the system is processing a response, providing visual feedback to the user. These variables are essential for enhancing the user experience by showing that the system is actively handling the request.

The `sendMessage` function sends the user's message to the backend and handles the system's response. The function is detailed in Code 21.

Code 21: Send Message Function (`Chat.vue`)

```
const sendMessage = async (message: string) => {
  pendingMessage.value = {
    content: message,
    role: 'user',
    timestamp: new Date().toISOString()
  }

  responseLoadingMessage.value = {
    content: '',
    role: 'system',
    timestamp: new Date().toISOString(),
    loading: true
  }

  const result = await messagesApi.create.mutateAsync({
    content: message
  })

  if (result) {
    selectedCaseStore.case.state = result.state
    selectedCaseStore.case.messages = result.messages

    if (selectedCaseStore.case.title !== result.title) {
      selectedCaseStore.case.title = result.title
      casesApi.findAll.refetch()
    }
  }

  pendingMessage.value = undefined
  responseLoadingMessage.value = undefined
}
```

This function updates the local state to reflect that a message is being sent and that the system is processing a response. The message is sent using `messagesApi`, and once a response is received, the chat state is updated, and any visual indicators (such as the loading message) are cleared. This ensures a smooth and responsive chat interface.

The template for `Chat.vue` renders the chat history via the `ChatHistory` component and includes the message input field (`ChatInput`), as shown in Code 22.

Code 22: Template for Chat.vue (`Chat.vue`)

```
<template>
  <div v-if="selectedCaseId" class="bg-surface">
    <ChatHistory
      :messages="messages"
      :is-fetching="messagesApi.findAll.isFetching.value && false"
      @break-ice-with-message="sendMessage($event)"
      @break-ice-with-document="showFilePicker"
    />
    <ChatInput
      @send-message="sendMessage($event)"
      @show-file-picker="showFilePicker"
      @clear-file-selection="clearFileSelection"
      :selected-file-name="selectedFile?.name"
      :disabled="messagesApi.findAll.isFetching.value"
    />
    <input id="fileUpload" type="file" ref="fileUpload" @change="selectFile" hidden />
  </div>

  <span v-else class="d-flex flex-column h-100 justify-center" align-center ga-2 bg-
      surface">
    <v-icon class="opacity-50" color="blue-50" size="60" icon="$chat" />
    <b>Case required</b>
    <div class="text-center">To start communicating, create or select a case first.</
        div>
  </span>
</template>
```

The ChatHistory component is responsible for rendering the conversation, while ChatInput provides the user with an input field to send messages. The template ensures that both messages and files can be sent and provides feedback to the user while data is being fetched.

The component ChatInput.vue handles user input, including text messages and file uploads. It allows users to compose messages and trigger specific actions like sending or attaching files.

The Code 23 sets up the message input and prepares the component for emitting events when the user sends a message or selects a file.

Code 23: Defining Message and Emitting Events (ChatInput.vue)

```
const message = ref('')

const { selectedFileName = undefined } = defineProps<{ selectedFileName: string |
    undefined }>()
const emit = defineEmits<{
  sendMessage: [message: string]
  showFilePicker: []
  clearFileSelection: []
}>()
```

The message is a reactive reference that tracks the user's input, and selectedFileName tracks any file selected by the user. The emit function enables communication with the parent component, notifying it when a message is sent or the file picker is triggered. This ensures dynamic UI updates and seamless interactions.

The core function for sending messages is defined in Code 24.

Code 24: Send Message Function (ChatInput.vue)

```
function sendMessage() {
  if (message.value) {
    emit('sendMessage', message.value)
    message.value = ''
  }
}
```

When the user presses "Enter" or clicks the send button, the `sendMessage` function emits the `sendMessage` event and resets the input field. This provides smooth and intuitive message handling.

The message input field is set up with actions for sending messages and handling file uploads in the template of Code 25 and Code 26.

Code 25: Message Input Field (`ChatInput.vue`)

```
<v-text-field
  v-model="message"
  @keydown.enter.prevent="sendMessage"
  bg-color="surface"
  variant="outlined"
  color="primary"
  placeholder="Enter message..."
  rounded="lg"
  no-resize
  hide-details
>
```

Code 26: Trigger Buttons (`ChatInput.vue`)

```
<v-btn
  @click.prevent="emit('showFilePicker')"
  variant="text"
  size="small"
  density="default"
  icon="$paperclip"
  color="secondary"
/>
<v-divider vertical></v-divider>
<v-btn
  @click="sendMessage"
  :disabled="!message && !selectedFileName"
  variant="text"
  density="default"
  icon="$send"
  size="small"
  color="secondary"
/>
```

The `v-text-field` is bound to the reactive `message` property, automatically reflecting user input. The attached buttons allow users to trigger the file picker or send a message, ensuring the interface is user-friendly with easy-to-access actions.

The component `ChatMessage.vue` is responsible for rendering individual chat messages, differentiating between messages sent by the user and those from the system or the chatbot. It also incorporates a loading animation for messages still being processed.

The Code 27 shows how the props are defined and used in the component.

Code 27: Defining Message Props (`ChatMessage.vue`)

```
import { Message } from '@/types/api'

defineProps<Message>()
```

This component accepts a `Message` object as a prop, which contains details about the message content, its sender , and its status.

Messages are displayed differently based on their role, as shown in the Code 28.

Code 28: Message Display Based on Role (`ChatMessage.vue`)

```
<div
class="chat-message-row"
:class="{
  'justify-start': role === 'assistant' || role === 'system',
  'justify-end': role === 'user'
  }"
  >
  <div class="chat-message" :class="role">
    <div class="system-icon">
      <v-icon color="white" size="small" icon="$emojiHappy" />
    </div>

    <MessageLoadingAnimation v-if="loading" />
    <div v-else class="chat-message-text">{{ content }}</div>

    <div class="user-icon">
      <v-icon color="white" icon="mdi-account" />
    </div>
  </div>
</div>
```

Because of `<MessageLoadingAnimation v-if="loading" />` a loading animation is displayed if the message is still loading; otherwise, the message content is shown. Messages are aligned based on their role, with user messages appearing on the right and assistant or system messages on the left. This visual distinction helps users easily identify the origin of each message, making the chat interface more intuitive.

The messages are styled based on the `role`, with specific colors and padding applied to distinguish between different types of messages. See Code 29 for the styling of user messages.

Code 29: Message Style Based on Role (`ChatMessage.vue`)

```
&.user {
  align-items: flex-start;

  .system-icon {
    display: none;
  }

  .user-icon {
    display: block;
  }

  .chat-message-text {
    background-color: rgba(var(--v-theme-secondary), 0.1);
    margin-right: 12px;
    border-radius: 10px 0 10px 10px;
    padding: 8px 14px;
  }
}
```

The user messages are styled with a secondary theme color and appropriate padding to enhance readability and visually separate them from messages originating from the assistant or system. This contributes to a clearer and more aesthetically pleasing chat interface. Same is done with `.system` or `.assistant` with different styling.

The `MessageLoadingAnimation.vue` component provides a simple, bouncing loader animation that is displayed when the chatbot is processing a message. This component gives users feedback that the system is still working, preventing confusion during response delays.

The following template renders the bouncing loader animation, which consists of three small dots that bounce in a sequential animation. This behavior is shown in Code 30.

Code 30: Bouncing Loader Template (`MessageLoadingAnimation.vue`)

```
<div class="bouncing-loader">
  <div></div>
  <div></div>
  <div></div>
</div>
```

This code creates the basic structure for the loader animation, where three div elements are used to represent the dots that will bounce during the loading process.

The corresponding Cascading Style Sheets (CSS) defines the animation behavior for the loader, creating a smooth, continuous bouncing effect. The Sassy Cascading Style Sheets (SCSS) code in Code 31 applies the necessary styling.

Code 31: Bouncing Loader Animation (`MessageLoadingAnimation.vue`)

```
.bouncing-loader > div {
  animation: bouncing-loader 0.6s infinite alternate;
}

@keyframes bouncing-loader {
  to {
    opacity: 0.1;
    transform: translate3d(0, -5px, 0);
  }
}
```

The `bouncing-loader` class assigns an animation to each dot. The keyframes define the animation, where the opacity of the dots decreases to 0.1, and they move slightly upwards, creating the bouncing effect.

The `ChatHistory.vue` component is responsible for displaying the chat messages in a scrollable interface. It uses Vuetify's `v-virtual-scroll` to ensure efficient rendering, even with a large message history.

The Code 32 shows the setup for this component, including reactive data and the `scrollToBottom` function, which ensures the chat window scrolls to the latest message.

Code 32: Setup and Scroll Functionality (`ChatHistory.vue`)

```
import { Message } from '@/types/api'
import { useTemplateRef, watch } from 'vue'
import { VVirtualScroll } from 'vuetify/components'

const props = defineProps<{ messages: Message[]; isFetching: boolean }>()
const scroll = useTemplateRef<InstanceType<typeof VVirtualScroll>>('scroll')

function scrollToBottom() {
  if (scroll.value) {
    scroll.value.scrollToIndex(props.messages.length - 1)
  }
}

watch(
  () => props.messages.length,
  () => setTimeout(scrollToBottom, 100)
)
```

The `scrollToBottom` function ensures that when new messages are added to the chat, the scroll position is automatically updated to show the most recent message. This improves the user experience by keeping the conversation in view without requiring manual scrolling.

The following template, shown in Code 33, renders the message history and displays a loading indicator when messages are being fetched.

Code 33: Template for ChatHistory.vue (`ChatHistory.vue`)

```
<template>
  <div class="chat-container d-flex flex-column justify-center">
    <div v-if="isFetching" class="text-center">
      <v-progress-circular color="primary" indeterminate />
    </div>
    <v-virtual-scroll
      v-else-if="messages.length"
      class="chat-history"
      :items="messages"
      ref="scroll"
    >
      <template #default="{ item }">
        <ChatMessage v-bind="item" />
      </template>
    </v-virtual-scroll>
  </div>
</template>
```

When `isFetching` is true, the component displays a progress indicator, letting the user know that messages are being loaded. Once the messages are available , they are rendered in a scrollable list using `v-virtual-scroll`.

These components together form the foundation of the chat's visual feedback system, allowing users to understand when the system is processing a message. This feature enhances the chatbot's user experience by keeping users engaged during response delays.

5.2.1.7 Supplier Matching in Chat

This section introduces the supplier selection functionality within the chat, allowing users to browse a list of potential suppliers and select one based on criteria such as price and name. The feature is implemented through the `SupplierGallery.vue` component, which displays supplier options and lets users select a supplier for further interaction. This helps in narrowing down options for services or products that match the user's needs, streamlining the supplier matching process.

The following code defines the structure of a supplier and initializes a list of suppliers to be displayed in the chat, as shown in Code 34.

Code 34: Defining the Supplier Interface and Data (`SupplierGallery.vue`)

```
interface SupplierGalleryItem {
  id: string
  name: string
  image: string | undefined
  price: string
  url: string
}

const suppliers: SupplierGalleryItem[] = [
  {
    id: '1',
    name: 'Supplier 1',
    image: undefined,
    price: '125.000 - 150.000 €',
    url: 'https://www.google.com'
  },
  {
    id: '2',
    name: 'Supplier 2',
    image: 'https://via.placeholder.com/150',
    price: '150.000 - 200.000 €',
    url: 'https://www.google.com'
  },
  {
    id: '3',
    name: 'Supplier 3',
    image: 'https://via.placeholder.com/150',
    price: '200.000 - 250.000 €',
    url: 'https://www.google.com'
  }
]
```

The `SupplierGalleryItem` interface defines the structure for each supplier, including properties like `id`, `name`, `image`, `price`, and `url`. The `suppliers` array holds the data for three example suppliers, each with its own details. Some suppliers have images, while others do not, showing flexibility in how supplier information is displayed.

The following function allows the user to select a supplier from the list, storing the selected supplier in the reactive reference `selectedSupplier`, as demonstrated in Code 35.

Code 35: Selecting a Supplier (`SupplierGallery.vue`)

```
let selectedSupplier = ref<SupplierGalleryItem | undefined>(undefined)
function selectSupplier(supplier: SupplierGalleryItem) {
  selectedSupplier.value = supplier
}
```

The `selectedSupplier` reference keeps track of the currently selected supplier. The `selectSupplier` function updates this reference when the user clicks on a supplier, ensuring that the selected supplier is highlighted in the UI. This allows users to easily compare and select from different supplier options.

The template code below renders the list of suppliers as clickable cards. Each card displays the supplier's name, price, image (if available), and a button for selecting the supplier. The card structure is shown in Code 36.

Code 36: Rendering the Supplier Cards (`SupplierGallery.vue`)

```
<div v-for="supplier in suppliers" :key="supplier.id" class="supplier-card">
  <div class="d-flex ga-2 align-center">
    <img v-if="supplier.image" :src="supplier.image" alt="Supplier image" />
    <v-icon v-else icon="mdi-web" />
    <h4>{{ supplier.name }}</h4>
  </div>

  <p>{{ supplier.price }}</p>

  <div>
    <b>Further links:</b>
    <p>
      <a :href="supplier.url">{{ supplier.url }}</a>
    </p>
  </div>

  <div class="text-center">
    <v-btn
      :color="
        !selectedSupplier || selectedSupplier?.id === supplier.id ? 'secondary' : '#
            d0d0d2'
      "
      rounded="xl"
      class="select-button text-none"
      :class="{
        selected: selectedSupplier?.id === supplier.id
      }"
      @click="selectSupplier(supplier)"
      flat
    >
      <template #prepend>
        <v-icon icon="$plus" size="small" />
      </template>
      Select Supplier
    </v-btn>
  </div>
</div>
```

Each supplier is represented by a card with details such as name, price, and a link to the supplier's website. If an image is available, it is displayed; otherwise, a default icon is shown. The "Select Supplier" button dynamically changes its color and style depending on whether the supplier is selected or not, providing a clear visual indication of the active supplier.

5.2.1.8 Checklist for Extracted Information

This section focuses on rendering a dynamic checklist that automatically updates based on the attributes extracted from the chatbot conversation. The checklist helps ensure that all required information is provided, enhancing the user experience by visually tracking the progress of filling out necessary details. The three components involved—`Checklist.vue`, `Checkbox.vue`, and `ChecklistItem.vue` —work together to display attributes as checklist items and indicate whether they have been completed.

The `Checklist.vue` component is responsible for rendering the overall checklist, which displays each attribute as a separate checklist item. The checklist is populated based on the state of the current case, dynamically fetched from the `selectedCaseStore`.

The following snippet demonstrates how the attributes are computed based on the case state. Only attributes that are not related to the "service description" are included in the checklist, as shown in Code 37.

Code 37: Computing Attributes for Checklist (`Checklist.vue`)

```
const attributes = computed(() =>
  selectedCaseStore.case.state
    ?.flatMap((s) => s.attributes)
    .filter((a) => a.attribute_id !== 'service_description')
)
```

This code processes the case's state to extract relevant attributes for display as checklist items, filtering out irrelevant attributes like "service description." The computed property ensures that the checklist dynamically updates as new information is added during the conversation.

The template, shown in Code 38, shows how the checklist items are rendered, using the `ChecklistItem` component for each attribute.

Code 38: Rendering the Checklist Items (`Checklist.vue`)

```
<div v-if="attributes" class="overflow-y-auto d-flex flex-column ga-3">
  <ChecklistItem
    v-for="attribute in attributes"
    :key="attribute.attribute_id"
    v-bind="attribute"
  />
</div>
```

Each attribute is rendered using the `ChecklistItem` component, representing an individual item in the checklist. If no attributes are available, a placeholder message prompts the user to continue the conversation to provide the necessary information.

The `Checkbox.vue` component represents a simple visual checkbox indicating whether a checklist item has been completed. It uses Vuetify's `v-icon` to display a checkmark when the item is marked as "checked."

Code 39 shows how the checkbox state is managed via the `isChecked` prop.

Code 39: Managing the Checkbox State (`Checkbox.vue`)

```
const { isChecked = false } = defineProps<{
  isChecked: boolean
}>()
```

This allows the checkbox to display its state (checked or unchecked) based on the value passed from the parent component.

The template below in Code 40 renders the checkbox, which changes its appearance depending on whether it is checked.

Code 40: Rendering the Checkbox (`Checkbox.vue`)

```
<div class="checkbox" :class="{ checked: isChecked }">
  <v-icon class="check-icon" size="35" color="success" icon="$check" />
</div>
```

When `isChecked` is true, the `check-icon` is displayed, visually indicating that the attribute has been completed.

The `ChecklistItem.vue` component is responsible for rendering each individual item in the checklist. It receives an attribute as a prop and checks its value to determine whether the item is completed and what tags or values should be displayed.

The following snippet defines the props and computes the tags for each item, as demonstrated in 41.

Code 41: Computing Tags for Checklist Item (`ChecklistItem.vue`)

```
const tags = computed<string[]>(() => (value ? [value].flat().map((x) => String(x)) :
    []))
```

The `tags` property is computed from the attribute's value , allowing the component to display relevant tags or information associated with the item. The `.flat()` method ensures that arrays are handled correctly, and `String(x)` converts each value to a string for display.

The template below renders the checklist item, including the checkbox and any associated tags , as shown in 42.

Code 42: Rendering the Checklist Item (`ChecklistItem.vue`)

```
<div class="d-flex align-center ga-2">
  <Checkbox :is-checked="!!value" />
  <h4>{{ title }}</h4>
</div>
<div class="d-flex align-center ga-2">
  <div style="width: 34px; height: 100%"></div>
  <div v-if="tags.length" class="d-flex ga-2">
    <div v-for="tag in tags" :key="tag">
      <v-chip variant="flat" density="compact" color="chip-background" label>
        <b>{{ tag }}</b>
      </v-chip>
    </div>
  </div>
</div>
```

This section of the template displays a checkbox, the item title, and any tags associated with the item. The `v-chip` component is used to display the tags in a compact format, making the information visually accessible and easy to understand.

5.2.1.9 Theme and User Settings Persistence

This section covers how the application persists the user's theme preferences (dark or light mode) across sessions, ensuring a consistent user experience. The `ThemeSwitcher.vue` component enables users to toggle between themes, with their choice saved using local storage. Vue's reactive system and Vuetify's theme management are utilized to implement this functionality.

The user's selected theme is stored in the browser's local storage, allowing the theme preference to persist even after the user closes and reopens the application. The following code snippet demonstrates how local storage is utilized for theme persistence, as shown in 43.

Code 43: Using Local Storage for Theme Persistence (`ThemeSwitcher.vue`)

```
const storedTheme = useLocalStorage('selected-theme', theme.global.name.value)
```

This line uses `useLocalStorage` from `@vueuse/core` to store the current theme in local storage. The key `selected-theme` is used to save the theme's name. If no previous value is found, the default theme (`theme.global.name.value`) is stored.

When the component is mounted, the stored theme is applied to ensure the user's preference is loaded and set correctly. This ensures that the selected theme from the last session is applied when the user returns to the application, as demonstrated in 44.

Code 44: Syncing Theme on Mount (`ThemeSwitcher.vue`)

```
onMounted(() => (theme.global.name.value = storedTheme.value))
```

Thisline of code ensures that, as soon as the component is mounted, the theme stored in local storage is applied to the application.

The component watches for any changes in the theme and updates the local storage accordingly , ensuring that the latest theme choice is always saved. This is shown in 45.

Code 45: Watching for Theme Changes (`ThemeSwitcher.vue`)

```
watch(
  () => theme.global.name.value,
  (value) => (storedTheme.value = value)
)
```

The `watch` function observes changes to `theme.global.name.value` (the current theme). Whenever the user switches themes, the new value is stored in `storedTheme` , updating the local storage.

The template part of the component in Code 46 includes a Vuetify `v-switch` that allows users to toggle between light and dark themes.

Code 46: Template for Theme Toggle (`ThemeSwitcher.vue`)

```
<v-switch
  v-model="theme.global.name.value"
  false-value="telekom-light"
  true-value="telekom-dark"
  class="px-3"
  color="primary"
  hide-details
>
  <template #prepend>
    <v-icon icon="$darkMode" color="primary" size="small" />
    <span class="pl-4">Dark Mode</span>
  </template>
</v-switch>
```

The `v-switch` is bound to `theme.global.name.value`, allowing users to switch between the `telekom-light` and `telekom-dark` themes. The current theme is dynamically updated , and the `v-switch` provides a user-friendly toggle for light and dark modes.

All this ensures that the user's theme preference is saved and restored across sessions , enhancing the user experience by maintaining a consistent interface. The combination of local storage and Vuetify's theme management provides a seamless and intuitive approach to theme persistence, ensuring that the user's selected theme is applied even after the application is reopened.

5.2.2 Backend

5.2.3 Deployment

5.3 Evaluation

5.4 Further Use

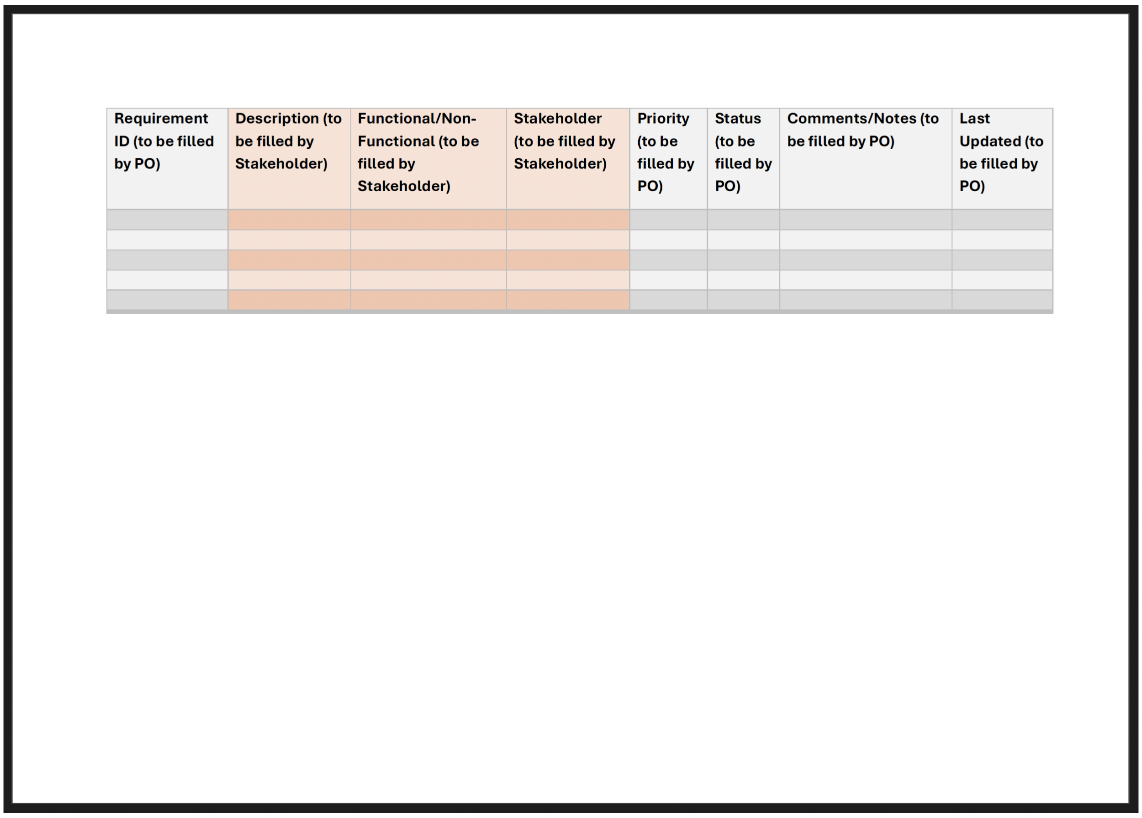
6 Fazit

Wünsche Euch allen viel Erfolg für das 7. Semester und bei der Erstellung der Thesis. Über Anregungen und Verbesserung an dieser Vorlage würde ich mich sehr freuen.

Appendix

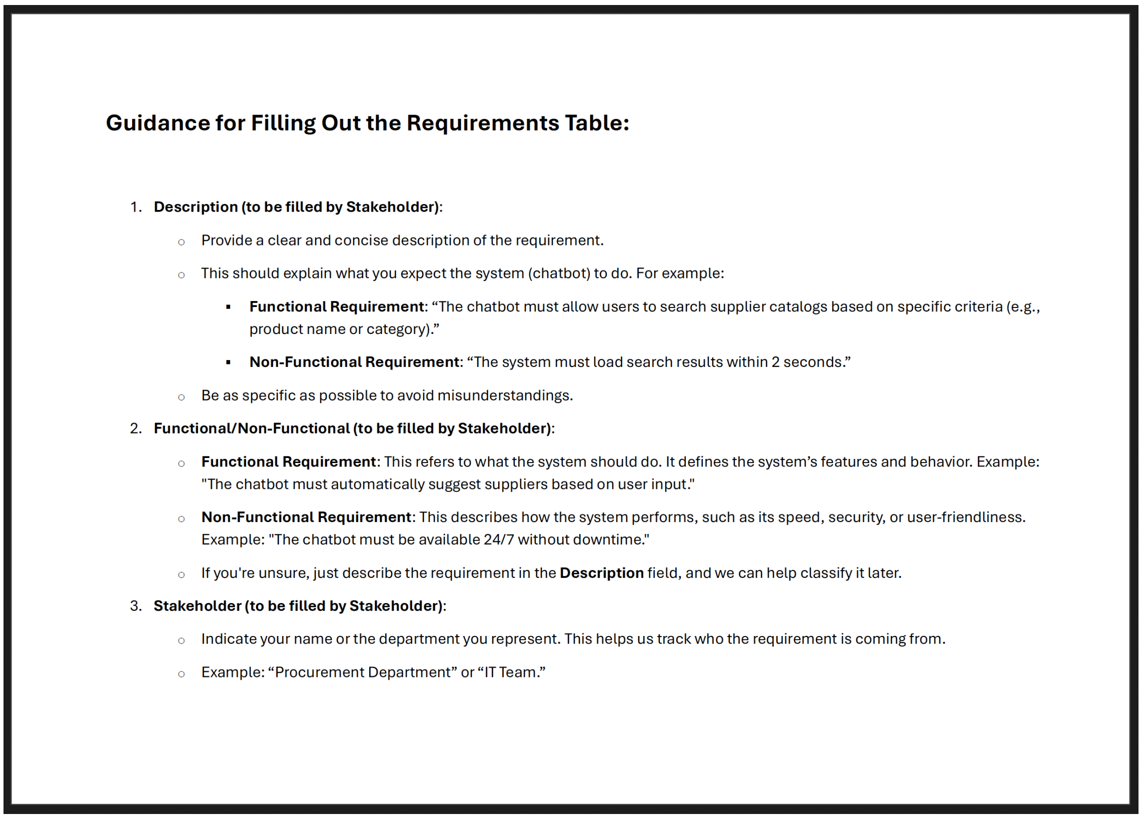
Appendix 1: Requirement Document

Figure 6: Requirement Document (Page 1)



Requirement ID (to be filled by PO)	Description (to be filled by Stakeholder)	Functional/Non-Functional (to be filled by Stakeholder)	Stakeholder (to be filled by Stakeholder)	Priority (to be filled by PO)	Status (to be filled by PO)	Comments/Notes (to be filled by PO)	Last Updated (to be filled by PO)

Figure 7: Requirement Document (Page 2)



Guidance for Filling Out the Requirements Table:

1. **Description (to be filled by Stakeholder):**
   - Provide a clear and concise description of the requirement.
   - This should explain what you expect the system (chatbot) to do. For example:
     - **Functional Requirement**: "The chatbot must allow users to search supplier catalogs based on specific criteria (e.g., product name or category)."
     - **Non-Functional Requirement**: "The system must load search results within 2 seconds."
   - Be as specific as possible to avoid misunderstandings.
2. **Functional/Non-Functional (to be filled by Stakeholder):**
   - **Functional Requirement**: This refers to what the system should do. It defines the system's features and behavior. Example: "The chatbot must automatically suggest suppliers based on user input."
   - **Non-Functional Requirement**: This describes how the system performs, such as its speed, security, or user-friendliness. Example: "The chatbot must be available 24/7 without downtime."
   - If you're unsure, just describe the requirement in the **Description** field, and we can help classify it later.
3. **Stakeholder (to be filled by Stakeholder):**
   - Indicate your name or the department you represent. This helps us track who the requirement is coming from.
   - Example: "Procurement Department" or "IT Team."

Appendix 2: Requirement Elicitation

Appendix 2.1: Screen Mock-Ups in Miro

Appendix 2.2: Screen Prototypes in Figma

Appendix 2.3: Filled-out Requirement Documents

Appendix 3: Prototyping

Appendix 3.1: Frontend

Appendix 3.1.1: Example API Endpoint Definition

Code 47: Example API Endpoint Definition (`v1.json`)

```json
"/api/v1/cases": {
  "get": {
    "summary": "List Cases",
    "operationId": "list_cases_api_v1_cases_get",
    "parameters": [
      {
        "name": "user_id",
        "in": "query",
        "required": false,
        "schema": {
          "anyOf": [
            { "type": "string" },
            { "type": "null" }
          ],
          "title": "User Id"
        }
      },
      {
        "name": "limit",
        "in": "query",
        "required": false,
        "schema": {
          "anyOf": [
            { "type": "integer" },
            { "type": "null" }
          ],
          "default": 10,
          "title": "Limit"
        }
      }
    ]
  },
  "post": {
    "summary": "Create Case",
    "operationId": "create_case_api_v1_cases_post",
    "requestBody": {
      "required": true,
      "content": {
        "application/json": {
          "schema": {
            "$ref": "#/components/schemas/CreateCaseRequest"
          }
        }
      }
    }
  }
}
```

Bibliography

Abbasi, *Maryam*, *Bernardo*, *Marco V.*, *Váz*, *Paulo*, *Silva*, *José*, *Martins*, *Pedro* (2024): Adaptive and Scalable Database Management with Machine Learning Integration: A PostgreSQL Case Study, in: Information, 15 (2024), Nr. 9, p. 574

Carrión, *Carmen* (2022): Kubernetes Scheduling: Taxonomy, Ongoing Issues and Challenges, in: ACM Comput. Surv. 55 (2022), Nr. 7, 138:1–138:37

Chen, *Junqiao* (2023): Model Algorithm Research Based on Python Fast API, in: Frontiers in Science and Engineering, 3 (2023), Nr. 9, pp. 7–10

Christensen, *Scott*, *Brown*, *Marvin*, *Haehnel*, *Robert*, *Church*, *Joshua*, *Catlett*, *Amanda*, *Schofield*, *Dallon*, *Brannon*, *Quyen*, *Smith*, *Stacy* (2022): A Python Pipeline for Rapid Application Development (RAD), in: s.l., 2022-01-01, pp. 240–243

Fareez, *MMM*, *Thangarajah*, *Vinothraj*, *Saabith*, *Sayeth* (2020): POPULAR PYTHON LIBRARIES AND THEIR APPLICATION DOMAINS, in (2020)

Floyd, *Christiane* (1984): A Systematic Look at Prototyping, in: *Budde*, *Reinhard*, *Kuhlenkamp*, *Karin*, *Mathiassen*, *Lars*, *Züllighoven*, *Heinz* (eds.), Approaches to Prototyping, Berlin, Heidelberg: Springer, 1984, pp. 1–18

Joshi, *Ankush*, *Tiwari*, *Haripriya* (2024): An Overview of Python Libraries for Data Science | Journal of Engineering Technology and Applied Physics, in ()

Kaluža, *Marin*, *Vukelic*, *Bernard* (2018): Comparison of Front-End Frameworks for Web Applications Development, in: Zbornik Veleučilišta u Rijeci, 6 (2018), pp. 261–282

Krishnamoorthy, *Venkatesh* (2021): Evolution of Reading Comprehension and Question Answering Systems, in: Procedia Computer Science, Big Data, IoT, and AI for a Smarter Future, 185 (2021), pp. 231–238

Li, *Nian*, *Zhang*, *Bo* (2021): The Research on Single Page Application Front-end Development Based on Vue, in: Journal of Physics: Conference Series, 1883 (2021), Nr. 1, p. 012030

Lortie, *Christopher J.* (2022): Python and R for the Modern Data Scientist, in: Journal of Statistical Software, 103 (2022), pp. 1–4

Mokoginta, *Deyidi*, *Putri*, *Desfita Eka*, *Wattimena*, *Fegie Yoanti* (2024): Developing Modern JavaScript Frameworks for Building Interactive Single-Page Applications, in: International Journal Software Engineering and Computer Science (IJSECS), 4 (2024), Nr. 2, pp. 484–496

N., *Nivedhaa*, *Pub*, *Iaeme* (2023): Evaluating Devops Tools and Technologies for Effective Cloud Management, in: 1 (2023), pp. 20–32

Openja, *Moses*, *Majidi*, *Forough*, *Khomh*, *Foutse*, *Chembakottu*, *Bhagya*, *Li*, *Heng* (2022): Studying the Practices of Deploying Machine Learning Projects on Docker, in: Proceedings of the 26th International Conference on Evaluation and Assessment in Software Engineering, EASE '22, New York, NY, USA: Association for Computing Machinery, 2022-06-13, pp. 190–200

Shrivastava, *Samiksha* (2024): Design and Implementation of Chatbot Using Python, in: IJFMR - International Journal For Multidisciplinary Research, 5 (), Nr. 6

Sommerville, *Ian*, *Sawyer*, *Pete* (1997): Requirements Engineering: A Good Practice Guide, 1st ed., USA: John Wiley & Sons, Inc., 1997-03, 404 pp.

Syed, *Zeeshan Haque*, *Trabelsi*, *Asma*, *Helbert*, *Emmanuel*, *Bailleau*, *Vincent*, *Muths*, *Christian* (2021): Question Answering Chatbot for Troubleshooting Queries Based on Transfer Learning, in: Procedia Computer Science, 192 (2021), pp. 941–950

Thakur, *Aadi*, *Chandak*, *M. B.* (2022): A Review on Opentelemetry and HTTP Implementation, in: International journal of health sciences, 6 (2022), Nr. S2, pp. 15013–15023

Internet sources

Cano Rodríguez, *Juan Luis* (2024): Python Packaging Is Great Now: ‘uv‘ Is All You Need, <https://dev.to/astrojuanlu/python-packaging-is-great-now-uv-is-all-you-need-4i2d> (2024-08-10) [Access: 2024-10-05]

Declaration in lieu of oath

I hereby declare that I produced the submitted paper with no assistance from any other party and without the use of any unauthorized aids and, in particular, that I have marked as quotations all passages which are reproduced verbatim or near-verbatim from publications. Also, I declare that the submitted print version of this thesis is identical with its digital version. Further, I declare that this thesis has never been submitted before to any examination board in either its present form or in any other similar version. I herewith disagree that this thesis may be published. I herewith consent that this thesis may be uploaded to the server of external contractors for the purpose of submitting it to the contractors' plagiarism detection systems. Uploading this thesis for the purpose of submitting it to plagiarism detection systems is not a form of publication.

Cologne, 20.10.2024
(Location, Date)

[illegible]
(handwritten signature)