\chapter{Methodology}

From the outset, the objectives were clear: to create a project management tool that integrated the power of large language models (LLMs) while maintaining security, locality, and accountability. Existing solutions in the market tended to sacrifice these values in exchange for convenience, relying on cloud platforms that expose users to surveillance, data harvesting, and opaque decision-making. The aim of this project was to show that another path was possible.

Yet the path was not straightforward. The final system—a lightweight Python application with a QML interface—was the result of iteration, reconsideration, and adaptation. Early versions of the project looked very different. At first, the system was conceived as a web application, powered by a centralised server that would process requests and host LLM modules. This design was motivated by performance. By placing the model on a server, more powerful language models could be deployed, models that would be too resource-intensive for individual user machines. The web interface, meanwhile, would give users familiar access through their browsers, requiring no installation and simplifying the experience.

The concept was attractive in theory. It promised scalability, flexibility, and the possibility of stronger AI performance. However, it soon became apparent that this approach was not viable within the scope of the project. Building and maintaining a server-side architecture, with user authentication, task persistence, and model hosting, would require a skillset in web frameworks and frontend languages such as React. Lacking prior experience with these tools, progress was slow. More importantly, the approach threatened to undermine the very principles the project was founded upon. Even if the server were hosted on-site rather than in the cloud, users would still be asked to trust a system where their data travelled across a network to reach the point of processing. This risk, combined with the time limitations of the project, made the web-based design untenable.

The decision was made to pivot toward a different architecture: one centred on Python, running locally on the user’s machine. This choice immediately aligned with the ethical requirements. By keeping everything local, the system avoided the pitfalls of remote storage and network transfer. Users would retain complete control over their data, with no risk of third-party access. The use of Python also made development more realistic within the given timeframe. Its rich ecosystem of libraries, coupled with the developer’s familiarity with the language, made it possible to build and iterate quickly. To provide a modern interface without the complexity of a web stack, QML was adopted. This allowed the frontend to look and feel polished, while still communicating directly with the Python backend in a lightweight and transparent way.

This chapter retraces this journey, describing the requirements that guided development, the design choices that were made, the treatment of data, the integration of the LLM, and the architecture that ultimately emerged. The emphasis is not on presenting a rigid blueprint but on showing how each stage of development reflects the project’s principles: privacy, accountability, and usability.

\section{Requirements Gathering}

The requirements that defined the project stemmed directly from the problem statement. The system had to be more than just another project management tool. It had to demonstrate that intelligent assistance could be delivered without surrendering user autonomy to the cloud. Four requirements in particular stood out.

First, the system had to operate entirely offline. Users should never be asked to send their tasks, notes, or metadata to an external server. Second, it had to integrate an LLM in a way that was transparent and accountable. Users needed to see how inputs were processed and have the option to review the prompts and outputs that shaped the model’s responses. Third, the system had to remain usable. Privacy and accountability are hollow values if the software is too cumbersome for everyday use. And finally, the system had to be feasible. Within the scope of a student research project, it needed to remain realistic in terms of development time and available expertise.

These requirements were not static but sharpened as the project developed. The initial attempt to build a web application reflects an early prioritisation of performance and scalability: the belief that integrating a powerful model was the most important step. However, as difficulties mounted and the risks of network-based processing became clearer, the requirements were reinterpreted. Privacy and offline operation took precedence, even if that meant using smaller models. Feasibility was also decisive. The choice of Python was not merely pragmatic but a recognition that delivering a working system was more valuable than pursuing an over-ambitious architecture that could not be completed.

\section{Design Approach}

The development approach was iterative. Instead of a grand design drawn up at the start, the project evolved in cycles of building, testing, and refining. Features were introduced one at a time such as task creation, event logging, project dashboards with each integrated into the system before moving on to the next. This incremental style gave the project flexibility. If a feature proved impractical or conflicted with the ethical principles, it could be dropped without derailing the whole system.

Testing played a critical role in this cycle. Although not always preventative, testing acted as a feedback loop. Bugs were often discovered through use, prompting the creation of small corrective tests. These would then remain in place, preventing regressions. This reactive testing was well-suited to a research-driven project where exploration mattered as much as stability.

Importantly, design was not only about functionality but about principle. Every decision was weighed against the objectives. The abandonment of the web-based architecture illustrates this. The design could have delivered more powerful LLM performance, but it conflicted with the requirements of privacy and feasibility. By choosing instead to simplify and localise, the design process remained faithful to the project’s aims.

\section{Data Sources and Processing}

Unlike many AI-driven projects, this system deliberately avoided external datasets. The only data it processed was that provided directly by users: task descriptions, deadlines, notes, and project events. This decision was crucial for privacy. By not depending on external sources, the system ensured that user information remained contained within the local environment.

Nonetheless, the handling of data required care. User inputs could be messy, ambiguous, or inconsistent. To ensure that the LLM could process them reliably, a preprocessing step was introduced. This involved cleaning the text, removing unnecessary characters, and formatting it into structured prompts. The aim was not to distort the meaning of the user’s input but to reduce ambiguity and guide the model toward more accurate classifications.

This process was refined over time. Early outputs from the model were sometimes vague or inconsistent, leading to adjustments in preprocessing rules. By shaping the input more clearly, the system was able to produce more consistent outputs. Because these rules were transparent, users could also inspect and understand how their inputs were being transformed, strengthening the sense of accountability.

\section{LLM Integration}

Selecting and integrating an LLM required careful compromise. The abandonment of the web-based architecture meant that the system could not rely on the largest, most powerful models. Instead, smaller models capable of running locally were used. While this meant sacrificing some raw performance, it aligned with the ethical requirements of security and offline use.

To compensate for these limitations, the project emphasised prompt design. Prompts were crafted to provide the model with clear context and structure, reducing the chance of misclassification. This was an iterative process: prompts were tested on sample inputs, revised in response to errors, and refined until they produced consistent outputs.

Equally important was transparency. Every interaction with the LLM was logged. Users could see not only the output but also the exact prompt that had been sent. This made it possible to audit the system’s behaviour, to distinguish between errors caused by vague input and those caused by the model itself. In this way, the integration strategy preserved accountability, addressing one of the central concerns raised in the problem statement.

\section{System Architecture}

The architecture that emerged was simple but effective. At its core were three components: a Python backend, a QML frontend, and a lightweight SQLite database.

The backend managed logic, persistence, and model interaction. It was written in Python, chosen for its balance of accessibility and power, and for the developer’s familiarity with the language. The frontend was written in QML, allowing the interface to look modern while remaining lightweight. The database stored tasks, projects, and events in a reliable but compact format.

What makes this architecture distinctive is its tight integration. Backend functions were exposed directly in the QML interface, allowing changes in logic to be reflected immediately in the frontend. This made iteration rapid and transparent, even if it sacrificed some of the modular separation found in more industrial architectures.

By discarding unnecessary layers of complexity—such as web servers, authentication flows, or cloud connectors—the system remained true to its ethical principles. It was fully local, fully offline, and fully under the user’s control.