From Planning to Code: Automating Software Development Lifecycle with Reasoning Models and Agentic Tools

Master Thesis

by

Sushant Huilgol

Matriculation number: 11038057

2025-09-24

SRH University Heidelberg

School of Information, Media and Design

Applied Computer Science (ACS)

Degree course: “M.Sc. Applied Computer Science “

Major field: “Business Computing “

First Supervisor — Prof. Dr. Gerd Moeckel

Second Supervisor — Prof. Paul Tanzer

Declaration of Authorship

I hereby declare that my herewith submitted paper is my own original work. I have written it independently without outside help and have not used any sources other than those indicated - in particular, no sources not named in the references.

I have appropriately indicated any direct quotations or passages taken from literature, and the use of intellectual property from other authors by providing the necessary citations within the work. This applies equally to the sources used for text generation by Artificial Intelligence (AI).

I hereby declare that the paper was not previously presented to another examination board, and I also confirm that the PDF version of this paper is identical in content to the hard copy.

Heidelberg, 23.09.2025

Sushant Huilgol

Acknowledgement

I want to thank everyone who helped make this master's thesis possible. Working on this research about automating software development with AI has been quite a journey, and I couldn't have done it alone.

My biggest thanks go to my supervisors, Prof. Dr. Gerd Moeckel and Prof. Paul Tanzer. They really went above and beyond in helping me figure out this complex topic. Whenever I got stuck or confused, they were there with practical advice and honest feedback. Their patience meant a lot in the success of this thesis. I'm grateful to SRH University Heidelberg for giving me the chance to dig into this research. The resources here made a real difference. What I appreciated most was how the program connected classroom learning with real industry problems, which is exactly what this thesis tried to tackle.

A special thanks to the developers and project managers who took time out of their busy schedules to evaluate my system. Getting feedback from people who actually work in software development was eye-opening. Their honest opinions helped me understand where this technology might actually be useful and where it still falls short.

I also want to acknowledge all the researchers whose work I built on. Reading through papers and studies gave me the foundation to understand what had already been tried and what gaps still existed. Without their groundwork, I wouldn't have known where to start.

This thesis ended up being more challenging than I expected, but also more rewarding. It represents not just my own effort, but all the help and support I received along the way. I'm thankful to have had such good guidance and resources to make it happen.

Abstract

Building software involves many steps - planning everything out, designing how it'll work, writing the actual code, testing it thoroughly, deploying it to users, and then keeping it maintained. While AI has gotten pretty good at helping with specific parts like generating code, nobody has really figured out how to automate the whole process from start to finish. Most current tools just tackle one piece at a time, like helping write functions or managing simple tasks, but they can't handle the bigger picture or adapt when things get complicated.

This thesis presents a prototype that tries to bridge this gap by combining AI reasoning with specialized agent tools to handle two key phases: planning projects and generating code. The system works by having different AI agents collaborate - each one focuses on a specific job like mapping out business goals, identifying potential risks, or breaking work into manageable chunks. When given a project brief, these agents work together to create detailed plans with timelines, resource needs, and technical recommendations. Then, during development, the system can categorize tasks by what technology they involve and actually write code snippets that fit with the overall architecture.

To see how well this actually worked, experienced developers and project managers tested it out and provided feedback. They rated different aspects of the generated plans and code, plus shared honest thoughts about what was useful and what wasn't. The findings were interesting - the system does create genuinely helpful planning documents and actionable development tasks. But it's definitely not ready to replace human judgment entirely. It works best as an assistant that can speed up the early stages of projects and give developers a solid starting point, rather than something that can run projects on autopilot. The results show both the real potential of this approach and where current AI still hits its limits when dealing with complex, multi-step workflows.

Table of Contents

1 Introduction

The software development lifecycle is one of the most important processes followed in any software industry or project. This process is divided into different stages such as planning, design, coding, testing, deployment, and maintenance. While there have been major advancements in the usage of AI, particularly Large Language Models (LLMs) like OpenAI, Google's Gemini, DeepSeek, etc., in automating code generation and developer support, the end-to-end automation of the software development lifecycle remains largely unexplored. Current systems mainly rely on basic AI capabilities that are pre-trained using available data. Due to this, the systems have a narrow scope and they lack reasoning depth, and cannot adapt to handle a full cycle of software development tasks.

Recently, the capabilities of AI systems have been accelerated by advancements in reasoning models, agentic tools, deep search, etc. Models such as OpenAI, Google's Gemini, and DeepSeek demonstrate various techniques such as chain-of-thought reasoning, multi-agent collaboration, and many more, which highlight the growth of AI in the future. Reasoning models offer structured thinking and decision-making capabilities by decomposing a complex task and reasoning through the intermediate steps, while AI agents extend their reach and fetch data from the real world. When combined, they can make powerful systems that can manage end-to-end workflows with very minimal human intervention.

Despite all these advancements in AI, there remains a gap in using these improvements to create a system that can automate the entire software development lifecycle. The solutions available currently only focus on the development part of the lifecycle, like code generation or task automation, and do not provide any approach towards other phases like planning, design, execution, and refinement.

This thesis aims to fill that gap by designing and evaluating a prototype that leverages the strengths of both reasoning models and agentic tools to automate

tasks of the project planning phase and some parts of code generation. By combining reasoning models and agentic tools, the system aspires to produce outputs that are more accurate, sensible, actionable, and context-aware. The thesis also aims to explore how the system can be evaluated effectively using different metrics.

1.1 Objectives

The main objective of this thesis is to investigate whether the combination of reasoning models along with agentic tools can efficiently automate the key phases of the software development lifecycle, producing high-quality and actionable outputs. In this thesis, the main focus is on 2 of these phases, namely Planning and Development. The main task is to build a prototype that demonstrates this integration and evaluates its performance based on some defined benchmarks.

To achieve this, the thesis will aim to complete the following objectives:

- Design and develop a modular prototype system that combines reasoning models with agentic tools.
- Use reasoning models to create structured and detailed project plans from high-level requirements, which will automate project planning tasks.
- Allow automated code generation based on project plans while guaranteeing that the output is correct, modular, and easy to maintain.
- Assess the system's performance with both qualitative and quantitative measures. This includes checking accuracy, completeness, correctness, and usability.
- Compare the system's outputs to traditional or human-generated results to evaluate its effectiveness and practical value.

1.2 Research Design

This section presents the research questions that guided the study and the method used to tackle them. The research follows an exploratory and practical approach aimed at

building a system that uses reasoning models and agent tools to automate important tasks in the software development lifecycle.

1.2.1 Research Questions

This thesis is based on the following main research question:

Main Research Questions (MRQ): Can reasoning models, when combined with agentic tools, automate key stages of the software development lifecycle, such as project planning and code generation, while delivering precise, useful, and context-aware outputs?

To investigate this main question, the following sub-research questions are developed:

Sub Research Question 1 (SRQ 1): How effectively can reasoning models be used to create structured and complete project plans from high-level task descriptions?

Sub Research Question 2 (SRQ 2): To what extent can the system generate correct, modular, and maintainable code based on those plans?

Sub Research Question 3 (SRQ 3): How do AI agents improve the ability of reasoning models in aiding automated software development?

1.2.2 Research Design Approach

The research takes an exploratory, implementation-based approach. It focuses on building and evaluating a prototype system that combines the strengths of reasoning models and agentic tools.

The design process is iterative and includes the following phases:

- **Conceptualization**: Based on the literature review, study the current methods, identify gaps and opportunities in automating software development with AI.

- **System Modeling**: Design a modular architecture that includes a user interface, backend logic, reasoning capabilities, and integration of agentic tools.
- **Prototyping**: Develop and test the system in stages, allowing for ongoing evaluation and improvement.
- **Performance Evaluation**: Use defined metrics to assess how well the system meets the goals of automation and output quality.

This approach allows flexibility in implementation. It also encourages continuous validation of design decisions through hands-on experimentation and feedback.

1.2.3 Research Methodology

By combining the research questions with the design approach, a research methodology to guide the creation and assessment of the proposed system was developed. Each sub-research question (SRQ) connects to specific phases in the system design process. This structure allows us to investigate how effective reasoning models and agent tools are in automating software development tasks. The research follows a design science approach, focusing on iterative development, creating artifacts, and evaluating them in real-world settings.

The methodology consists of five main stages:

- Problem Identification (linked to MRQ & SRQ1)
- System Design and Development (linked to SRQ2 and SRQ3)
- Demonstration and Testing
- Evaluation
- Reflection and Conclusion

These stages connect to the practical implementation of the research questions. They ensure the results are both theory-based and supported by empirical evidence. Figure 1.1 shows the full structure of the research methodology. It visually represents the connection between the research questions, design phases, and evaluation steps.

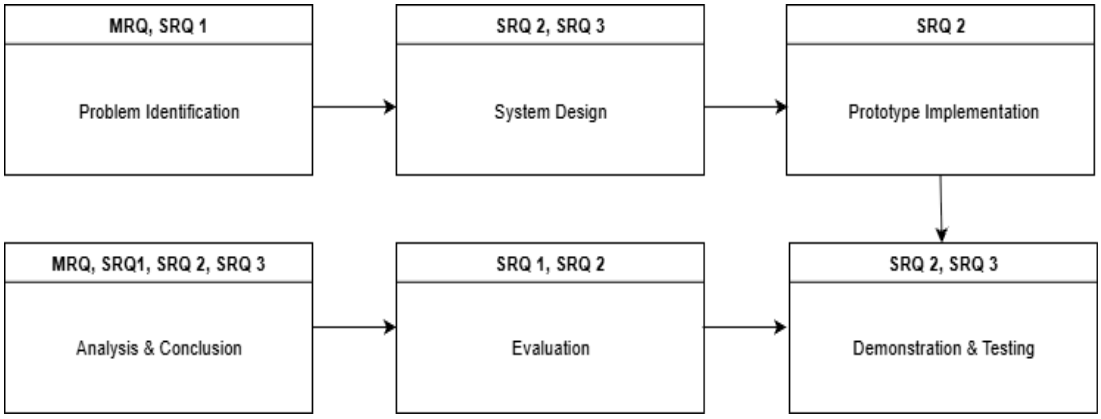

Figure 1.1: Research Methodology (Author)

1.3 Structure of the Thesis

This thesis work is organized in a clear format. Each chapter builds on the previous one to explain the research process, system development, and evaluation. The chapters guide the reader through the motivation, background, methodology, implementation, results, and future outlook of the study.

Chapter 1: Introduction. This chapter provides the background, motivation, research objectives, and design approach that guide the thesis.

Chapter 2: Literature Review. This chapter reviews existing research on reasoning models, agentic tools, and AI-assisted software development. It identifies gaps and informs the design of the system.

Chapter 3: Methodology & Implementation. This chapter describes the architecture, components, and technologies used to develop the prototype system. It also includes detailed implementation details of the application. It shows how the pipeline is created for the seamless integration of reasoning models and AI agents.

Chapter 4: Evaluation. This chapter presents the methods, metrics, and results used to assess the system's performance in automating planning and code generation.

Chapter 5: Conclusion. This chapter summarizes key findings and reflects on the limitations of the project.

Chapter 6: Future Scope. This chapter outlines long-term possibilities and ideas for extending the system's capabilities beyond the current implementation.

Supplementary sections such as **References**, **List of Figures**, **List of Tables** and **List of Abbreviations** are included at the end of the document for clarity and completenes.

2 Literature Review

This chapter covers the background research needed to understand the core concepts behind this work. The discussion begins with the software development lifecycle - how software projects typically unfold from start to finish - with particular focus on the planning and development phases since these are central to the research. The chapter then breaks down what Large Language Models actually are, how they're built, what they can and can't do, and which ones are currently dominating the market. The chapter also explains "agentic tools" - essentially, ways to give AI systems the ability to interact with real-world environments instead of just generating text responses. Several frameworks exist for building these kinds of systems, and the main ones are explored here. The final section looks at recent breakthroughs in AI reasoning and agent technology, plus identifies where current research still falls short - gaps that directly motivated this particular project.

2.1 Software Development Lifecycle

2.1.1 Definition and importance of SDLC

The Software Development Life Cycle (SDLC) is basically a structured way to build, implement, and improve software systems. It came about during the engineering boom and was really a response to what people call the mid-20th-century software crisis - a time when tons of projects were failing, budgets were going over by around 45%, and big IT projects were consistently running late by more than 7% [1]. At its core, SDLC is what [2] describe as a step-by-step approach to solving problems through five connected phases:

- **Requirement Analysis**, which includes gathering stakeholder needs using structured interviews and use case modeling.
- **Design**, which centers on creating system components using High-Level and Low-Level Design specifications.
- **Development**, which converts designs into executable code.

- **Testing**, which checks functionality through structured verification methods.
- **Maintenance**, which maintains operational integrity through ongoing improvements.

This framework is important for four main reasons:

- First, it helps reduce risks. The Standish Group looked at data from 2015-2020 and found something pretty telling - projects that followed SDLC methods succeeded 31% of the time, while projects that just winged it with ad-hoc approaches failed 19% of the time. The difference comes down to having quality checkpoints built in that catch vague or confusing requirements before anyone starts coding [1].
- Second, another key advantage is much smarter resource management. If you ever run simulations for different staffing setups, you'll see that just randomly assigning people to projects is a surefire way to create bottlenecks. Think about it: if a large project doesn't have enough programmers, the whole thing can grind to a halt, stuck waiting for critical development work to be completed [1].
- Third, this structured approach also acts as a guardian for quality. It aligns perfectly with frameworks like CMMI, which uses a five-level system to help organizations mature their processes across 22 different areas. This means you're systematically getting better at core tasks, whether it's nailing down requirements from the start (REQM) or using real data to steer project decisions (QPM) [2].
- Fourth, it's all about building systems that are ready for the real world. Modern SDLC practices have embraced concepts like stateless RESTful APIs and containerized deployments. In practical terms, this is what allows an e-commerce website to handle a massive surge of users during a sale without crashing or slowing to a crawl. The foundation is built to be both adaptable and scalable from the ground up. [3].

2.1.2 Common SDLC models

Waterfall Model

The Waterfall model takes a straight-line approach to building software. It moves through phases one after another: Requirements, Design, Implementation, Testing, and

Maintenance. This model works well when requirements are set in stone from the beginning, like when you're building software for aircraft systems where safety is everything. In those cases, having extremely detailed documentation isn't just helpful - it's absolutely necessary for passing audits and meeting regulatory standards. However, findings from discrete-event simulations highlight significant weaknesses [1]. The model's rigid, phase-locked structure, where results are fixed upon completion, leads to resource shortages. In their simulations of 100 projects, large-scale implementations faced 34 delays, averaging 9.7 units of time due to a lack of programmers. These issues prompted the development of hybrid models, which combine Waterfall's emphasis on documentation with Kanban's approach to workflow visualization. Recent Project Management Institute (PMI) data (2020) shows that Waterfall remains important. About 56% of government and healthcare projects continue to use it for regulatory purposes.[1].

Agile Model

Agile methodologies mark a significant change from plan-driven to value-driven development. They focus on delivering work in small, manageable increments through time-limited sprints, constant collaboration with stakeholders, and ongoing adjustments to the backlog. This approach began with the Agile Manifesto in 2001, transforming web engineering by adapting to shifting requirements seen in digital contexts, such as changing cybersecurity standards or the need for designs that work across different devices [3]. Research by [4] shows that using Scrum can cut time to market by 30 to 35%. Daily check-ins and sprint reviews help eliminate misalignments. Test-Driven Development (TDD) decreases defect rates to 0.4 per function point by incorporating quality checks within the development process instead of leaving them for later stages. However, some critics point out gaps in documentation; more complex financial systems see a 22% rise in audit failures due to inadequate requirement traceability [5]. Kanban adaptations improve this by visualizing workflows and limiting work in progress to 3 to 5 tasks per developer, which reduces time lost on switching tasks by 40% [3].

Spiral Model

Developed by Barry Boehm in 1986, the Spiral model incorporates risk analysis into a repeating structure of four parts: Planning (setting objectives), Risk Assessment (spotting technical and operational threats), Engineering (creating prototypes), and Evaluation (gathering feedback from stakeholders). Each cycle or "spiral" increases system functionality while addressing uncertainties. This is especially beneficial for large e-commerce platforms where requirements develop over time [6]. It is reported that risk-driven prototyping can identify scalability issues early, leading to a 38% reduction in rework following deployment in cloud-native applications. When paired with CMMI Level 3 processes, failure rates drop by 22% thanks to the use of standardized risk checklists and formal design reviews [2]. However, the model requires advanced risk management skills. Teams without trained project managers may encounter cycles that are 31% longer due to failing to prioritize threats properly [3].

RAD and Lean Models

Rapid Application Development (RAD) emphasizes speed. It uses Joint Application Development (JAD) workshops and Computer-Aided Software Engineering (CASE) tools, shortening development cycles by 60% for marketing campaign microsites [6]. Lean Software Development applies manufacturing strategies to cut waste, like unnecessary features. It employs Value Stream Mapping and Kaizen continuous improvement. Using "Build-Measure-Learn" cycles, it validates Minimum Viable Products (MVPs) with real users and adjusts based on user data, resulting in 40% higher feature adoption rates [3].

2.1.3 Overview of SDLC stages

Planning and Requirement Analysis

This foundational phase turns stakeholder needs into actionable specifications through detailed elicitation techniques. Business analysts hold workshops to define functional

requirements, such as "the system shall process 500 transactions per second," and non-functional constraints like "response time should be 2 seconds or less under 10,000 users." These are documented in Software Requirements Specifications (SRS) along with traceability matrices [2]. Web engineering uses Balsamiq wireframes to prototype user interface flows and MoSCoW prioritization (Must/Should/Could/Won't) to manage changes in scope [3]. Impact analysis matrices help track requirement changes against configurable items, which prevents 27% of downstream defects [2]. Simulations by [1] show that not having enough analysts leads to unclear requirements, which cause design flaws and increase failure rates by 19% in later phases.

Design Phase

Design splits into High-Level Design (HLD) and Low-Level Design (LLD). HLD defines system layout using UML deployment diagrams and Entity-Relationship (ER) models, outlining server clusters, API gateways, and database sharding strategies for scalability [2]. These outputs undergo formal reviews by enterprise architects to ensure they fit with infrastructure needs. LLD then breaks down modules into pseudocode specifications, database schemas, and interface contracts. Test engineers create boundary value test cases for input validation [2]. Web engineering requires RESTful API contracts with OpenAPI specifications, not stateful sessions via JSON Web Token (JWT) tokens, and responsive designs using Cascading Style Sheets (CSS) Flexbox/Grid [3]. Simulation data indicate that unoptimized design phases average 8.3 time units but can peak at 18.1 units for complex integrations, consuming 34% of project budgets when iterative refinement is ignored [1].

Development Phase

Developers turn LLD artifacts into working code using established practices. Reusable component libraries cut down on duplicate coding by 18%. Peer reviews enforce Google Style Guides, and static analysis tools like SonarQube flag security vulnerabilities before commits [2]. Web engineering uses React/Angular for component-based user interfaces, WebSocket APIs for real-time notifications, and

Redis caching to speed up database queries by 40% [3]. This phase tends to be the most prone to bottlenecks. Resource modeling by [1] shows that a lack of programmers increases wait times for large projects to 9.7 units, which is 247% longer than adequately staffed teams. Third-party dependencies add further risks; Node Package Manager (NPM) library vulnerabilities account for 31% of security breaches in JavaScript applications [3]

Testing Phase

Testing employs a tiered strategy. Unit Testing (JUnit/Pytest) checks individual methods at over 80% coverage. Integration Testing (TestNG) ensures modules work together through contract testing. System Testing (Selenium/Cypress) runs end-to-end workflows against requirement specifications [2]. Web engineering expands on this with specific checks. Cross-browser testing (BrowserStack) ensures consistency across Chrome, Firefox, and Safari. Open Web Application Security Project (OWASP) Zed Attack Proxy (ZAP) penetration tests find Structured Query Language (SQL) injection and cross-site scripting vulnerabilities. Lighthouse audits check performance budgets [3]. Automated regression suites in DevOps pipelines cut test cycles by 50%, but large projects still see a 28.6% failure rate because of untested edge cases in authentication workflows [1]. User Acceptance Testing (UAT) in environments that mimic production and use real customer datasets catches 37% of usability issues before release (Hossain & Yas et al., 2023) [5].

Deployment and Maintenance

Deployment strategies must balance risk and speed. Phased rollouts aim at 10% user groups with automated rollback triggers if error rates exceed 1%. Blue-green deployments switch traffic between identical environments with failover in less than a second [3]. Maintenance, which takes up 60-70% of total lifecycle costs, covers three areas: corrective actions (hotfixes for serious bugs), adaptive updates (iOS/Android SDK migrations), and perfective improvements (performance tuning) [1]. Proactive maintenance using log analytics (ELK Stack) and automated alerts reduces mean time

to repair by 53%. In contrast, reactive methods increase technical debt by 23% each year [2]. For long-term web applications, "recycle phases" restart development cycles to change monolithic architectures into microservices, extending system viability by 4-7 years [2].

Critical Analysis and Emerging Trends

The world of software development (SDLC) is getting a serious shake-up from AI. It's not just a future concept anymore; we're seeing Machine Learning models that can actually predict how requirements might change, with some hitting 89% accuracy by learning from past projects. Then you have tools like GitHub Copilot that are already cutting coding effort by more than a third in some cases [1]. But AI isn't the only force at play. On the horizon, quantum computing is starting to demand that we rethink the SDLC altogether to accommodate qubit-based algorithms. At the same time, there's a growing push for sustainability, giving rise to "Green SDLC" practices. Think energy-efficient coding that can trim a cloud platform's carbon footprint by 18%, or using serverless architectures that automatically reduce wasted resources [3]. Of course, these exciting innovations land right on top of the ongoing challenges teams face every day: figuring out governance for hybrid models, managing technical debt, and fully integrating security with DevSecOps. What this all highlights is that the SDLC is no longer a rigid, one-size-fits-all methodology. It's transforming into a much more dynamic and adaptable discipline.

2.2 Planning and Development Phases in SDLC

2.2.1 Role of project planning: requirements gathering, scope definition, task estimation.

Project planning lays the groundwork for software development. It involves requirements gathering, defining the scope, and task estimation. The research highlights various methods and challenges:

Requirements Gathering and Scope Definition:

The Waterfall Model [1] puts a lot of emphasis on getting requirements finalized right from the start. This approach creates a formal Software Requirements Specification (SRS) document that's supposed to lock everything in place before moving forward. The whole idea is that requirements won't change much once they're set, but this can backfire badly if the initial analysis misses important details.

Web Engineering principles [3] point out how tricky this gets with web applications, where development cycles move quickly and security requirements keep evolving. They agree that Waterfall can work for projects where everything is clearly defined upfront, but they push for iterative approaches like Agile, Spiral, or Prototype models instead. These methods handle changing user needs and ongoing feedback much better during the planning phase. They suggest techniques like wireframing and creating Documents of Understanding (DoU) with actual screen mockups as smart ways to gather requirements [2].

The CDP Model [2] gets pretty specific about how to break down requirement gathering. It categorizes "Wire-framing," "Document of Understanding (with screens)," and "Scope of Work (SOW)" as three distinct methods for the "Requirement Gathering or Analysis (SRS Preparation)" phase. It also emphasizes reviewing the SRS through both "Intra Team Review" and "Inter Team Review" processes.

Task Estimation and Resource Planning:

A significant finding from the Waterfall simulation study [1] is its focus on estimating resources and identifying bottlenecks. The study employs discrete-event simulation with the SimPy framework in Python to model resource needs for analysts, designers, programmers, testers, and maintenance staff based on project size (small, medium, large). It demonstrates that an initial resource allocation, such as 10 programmers, can create considerable bottlenecks during implementation, causing delays. A "stepwise algorithm" refines resource allocation until "zero-wait time" is achieved, increasing the programmer count to 38. This highlights the importance and difficulty of precise resource estimation during planning.

The CDP Model [2] places planning within the CMMI framework, linking business

goals, such as delivering on time and meeting estimated efforts, to Process Performance Objectives (PPOs). It identifies metrics like "Requirement Analysis Effort per point" as key factors (X-factors) affecting PPOs. Process Performance Baselines (PPBs) and Process Performance Models (PPMs) are developed using historical project data and statistical tools like Minitab to support future project estimation and planning.

Methodologies, Frameworks, and Tools:

Models include Waterfall (rigid, sequential), Iterative/Incremental (flexible, includes feedback), Spiral (risk-driven), Agile (user-focused, iterative), Prototype (clarifies uncertain requirements), and V-Model (links development phases to testing). The choice of model relies on project stability [3].

Frameworks: CMMI (Capability Maturity Model Integration) offers a structured way for organizations to improve how they plan and manage projects by focusing on specific process areas like Requirements Management (REQM), Project Planning (PP), and Project Monitoring and Control (PMC). These areas help teams build stronger planning habits and adopt best practices such as Project Performance Baselines (PPBs) and Project Performance Models (PPMs) [2]. To support these improvements, various tools are used. Discrete-event simulation tools like SimPy help estimate resources and identify potential bottlenecks in workflows [1]. Statistical software such as Minitab is useful for testing data patterns, creating control charts, running ANOVA (Analysis of Variance), and developing PPBs and PPMs [2]. Additionally, MS Excel plays a practical role in recording project data and calculating performance metrics, making it easier to track progress and make informed decisions [2].

Comparison: The Waterfall paper [1] outlines a structured, simulation-driven approach to estimating resources during the early planning stages of a project. In contrast, the Web Engineering paper [3] highlights the importance of adaptability in web development, advocating for iterative models that better accommodate the evolving nature of such projects. The CDP paper [2] offers an in-depth look at planning sub-practices within a CMMI-based quality framework, focusing on measurement and

historical data. Pandey et al. provide a general overview that reinforces the essential role of requirement analysis and SRS.

2.2.2 Role of development: coding, integration, and initial testing.

The development phase turns design specifications into functional software through coding, integration, and initial testing.

Coding Practices:

The Waterfall model describes development as the phase where the software design becomes a set of programs or program units based on the Low-Level Design (LLD). It uses coding tools, such as compilers and debuggers, along with languages like C, C++, Python, Java, and PHP [1].

Web Engineering principles focus on technologies and architectures suitable for the web. This includes Client-Server Architecture, RESTful APIs, Asynchronous Communication (AJAX, WebSockets), and Responsive Design [3].

The CDP Model [2] categorizes development (Coding, including Unit Testing) into sub-practices: Reusable Code, Non-Reusable Code, and Third-Party Components. It specifically includes Unit Testing in the development phase. It also requires Code Review through methods such as Sample Based, Run Complete Checklist, and Expert Review.

Integration & Initial Testing:

Typically, Waterfall postpones integration and system testing to a separate phase after development [1, 7]

Iterative/Incremental models, which include Agile and Web Engineering approaches, combine development, integration, and testing within each iteration or sprint. This process allows for continuous validation of smaller components [3].

Web Engineering emphasizes Continuous Integration and Continuous Deployment (CI/CD) as key practices within the DevOps model. These practices automate the build, testing, and deployment processes [3].

The CDP Model differentiates between unit testing (done during development) and later quality control (QC) activities. QC Test Preparation includes creating Detailed Scenario-Based Test Cases, a Simplified Checklist, or tests Separate for Functional and Non-Functional/UI. QC Testing then includes Test Case Driven Testing, Ad-Hoc Testing, and Functional and Performance Testing [2].

Evolution/Variation in Approaches:

A clear change is evident from the strictly sequential coding phase in Waterfall to the integrated development and testing cycles in Iterative/Agile models, and the highly automated CI/CD pipelines in DevOps [3]. Web-specific principles, such as REST, statelessness, and asynchronous communication, represent a specialization within development practices [3].

The CDP Model formalizes the specific choices developers make, like using reusable code and third-party components. It also incorporates quality assurance, including code review and unit testing, directly into the development phase definition [2].

2.2.3 Challenges in these phases that could benefit from automation.

Literature highlights a number of planning and development challenges where automation can be of significant value.

2.2.3.1 Challenges Identified

- **Resource Bottlenecks and Idle Time**: The Waterfall simulation example [1] captures this issue the best. Poor resource allocation resulted in huge bottlenecks, for example, for programmers during implementation. This led to project delays, with average waiting times averaging to 9.7 units for large projects, with other resources, such as maintenance, going idle. Identification of the best mix of resources at a faster pace is a complicated task and requires several trials.

- **Volatility of Requirements and Scope Creep**: This is especially seen in web applications [3]. The constantly changing nature of the requirements makes it risky to try to freeze everything in the beginning (Waterfall). Managing these fluctuating requirements effectively is difficult.
- **Accurate Estimation**: Effort, time, and resources cannot be estimated accurately. The application of history data (PPBs) and statistical models (PPMs) in the CDP approach [2] and the requirement for simulation in Waterfall [1] reflect this.
- **Delays in Defect Detection**: The Waterfall model delays rigorous testing, which will be more expensive to fix later [1, 7].
- **Lack of Efficient Testing Processes**: Timely test case preparation, testing, and test case audit is cumbersome [2].
- **Challenges in Integration**: Manual integration in sequential models can be complex and prone to errors [7].
- **Maintaining Consistency and Standards**: Maintaining coding standards and design specifications is difficult to ensure manually.

2.2.3.2 Automation Strategies and Technologies

- **Simulation for Planning Resources**: The Waterfall paper by [1] recommends the use of the SimPy discrete-event simulation library in Python to automate. This allows organizations to simulate different resource allocation scenarios prior to initiating a project, which can be used to determine potential bottlenecks and plan for optimal resource levels, e.g., have 38 programmers instead of 10. This is seen as a risk-free and efficient means of experimenting and planning.
- **CI/CD Pipelines**: DevOps and modern web development practices [3] emphasize automation across key stages like deployment, integration, unit testing, and build processes. Tools like Jenkins streamline these tasks, offering fast feedback loops and making integration and deployment smoother and less error-prone.
- **Automated Testing**: While the papers don't explicitly mention specific tools, their emphasis on testing and the CI/CD environment clearly points toward the use of automated unit testing frameworks like JUnit and pytest. It also implies the value of incorporating automated functional and

regression testing tools to avoid the delays and inconsistencies that often come with manual testing.

- **Automated Code Review and Analysis**: Static code analysis tools play a key role in streamlining code reviews within the CDP model. They can automatically check for adherence to coding standards, flag potential security issues, and catch bugs early helping to reduce the burden on manual reviewers and improve overall code quality. [2].
- **Requirements Management Software**: Although not explicitly discussed in the papers, it's reasonable to assume that specialized software could support tasks like requirement tracing, impact analysis, and change control. Automating these processes would help manage the volatility often seen in dynamic development environments, reducing the risk of missed dependencies or overlooked changes. [2, 3].

2.2.3.3 Case Study Example:

Our primary case study is from the Waterfall simulation paper [1]. They executed 100 projects of different sizes. The initial scenario, with assigned resources being 5 Analysts, 5 Designers, 10 Programmers, 20 Testers, and 5 Maintenance staff, had incredible bottlenecks, mainly for programmers in implementation. Slacks averaged 21-34 per project size, while mean wait times were 5.3-9.7 units. Using an automated stepwise procedure in their SimPy simulation, they optimized resources over 40 runs to achieve zero-wait times (15 Analysts, 18 Designers, 38 Programmers, 49 Testers, and 10 Maintenance personnel). Automation not only eradicated wait times but also minimized total project completion time by a large margin, from 7522.174 to 5754.000 units, and also lowered peak phase durations. For instance, the maximum duration of the implementation stage of large projects was reduced from 62.4 to 20.0 units. This indicates the obvious benefits of utilizing simulation-based automation for resource enhancement in planning.

Finally, planning and development phases are crucial but come at the cost of challenges like requirement volatility, estimation flaws, resource limitation, and testing inefficiencies. The scholarly literature points towards a shift towards iterative approaches for flexibility and wholeheartedly supports automation through simulation

to improve planning, CI/CD for simpler development and integration, and automated testing to improve efficiency, predictability, and quality and address these ongoing issues.

2.3 Large Language Models (LLMs)

2.3.1 Definition and architecture (transformers, attention mechanisms)

LLMs are big AI models trained on massive text datasets. At their core is the Transformer architecture, which ditched older recurrent networks for self-attention mechanisms that process words in relation to each other all at once. They come in three flavors: decoder-only models (like GPT) for generating text, encoder-only models (like BERT) for understanding text meaning, and encoder-decoder models (like T5) for tasks like translation that need both skills. [8]

They're AI models built on neural networks with billions of parameters, learning from massive text data without labels. Their core uses transformers that spot connections between words in sequences. Key bits involve chopping text into tokens, tracking word positions, and stacked layers that process context. [9]

LLMs are based on the Transformer architecture, which uses self-attention mechanisms to process sequences in parallel, replacing sequential Recurrent Neural Network (RNNs)/ Convolutional Neural Network (CNNs) [10]. Key components include positional embeddings (e.g., rotary embeddings) and layer normalization for stable training [10].

2.3.2 Capabilities (text generation, summarization, reasoning, etc.)

These models are versatile. They can write new text, summarize long passages, translate languages, and even tackle basic reasoning, especially using tricks like chain-of-thought prompting where they "think step-by-step." How well they work depends on the task: they're great for brainstorming ideas (where perfect accuracy isn't needed),

okay for finding lots of relevant info (like customer support), and increasingly useful for high-stakes jobs like medical diagnosis when combined with other tools. [8]

These models handle jobs like writing text, translating languages, summarizing content, answering questions, and judging sentiment. Surprisingly, they can reason through problems and adapt to new tasks with just a few examples. They even shine in tricky areas like healthcare analysis or teaching tools. [9]

LLMs enable text generation (e.g., essays, code), summarization, and translation across 46 languages (e.g., BLOOM) [10]. They perform reasoning (e.g., context-aware responses) and sentiment analysis for market insights. They also exhibit few-shot learning, adapting to new tasks with minimal examples [11].

2.3.3 Limitations (context length, factual accuracy, hallucination)

LLMs aren't flawless. They often hallucinate (make up convincing but false info, termed "bullshit" in the paper), can amplify biases from their training data, and stumble on simple logic like counting letters in a word because they see words as tokens, not characters. They also suffer from catastrophic forgetting (losing old skills when learning new ones), risk model collapse if trained too much on their own outputs, and can be tricked via jailbreak prompts into giving harmful responses. Their memory is also limited by their context window. [8]

They sometimes make up believable but false facts which researchers call hallucinations. Their memory is limited too, struggling with super-long documents or chats. Plus, they inherit biases from training data, need h

eavy computing power, and trip up on words with multiple meanings. [9].

Limitations include high computational costs (training requires massive resources), bias propagation (e.g., racial/gender biases from training data), and context constraints affecting long-text tasks [11]. They also exhibit hallucinations (inaccurate outputs) and scalability challenges [10].

2.4 Leading LLMs in the Market

2.4.1 OpenAI

OpenAI's GPT-4.5 is their "largest and most capable GPT model for chat," focusing on unsupervised learning instead of chain-of-thought reasoning. It processes both images and audio data. The model lineup includes GPT-4.5 Preview, e3-mini (a "fast, flexible, intelligent reasoning model"), and GPT-4.0 (a "fast, intelligent, flexible GPT model"). These proprietary models have "multimodal capabilities" but require commercial licensing. OpenAI has not disclosed GPT-4.5's exact parameter count. [12]

Reasoning models: o3, o3-mini, o5, o4-mini [13]

2.4.2 DeepSeek

DeepSeek-R1 is a "671B parameter Mixture-of-Experts model" with "37B activated parameters per token," trained via "large-scale reinforcement learning" focused on reasoning. It excels in mathematics, code generation, and genomic data analysis. The model is "30 times more cost-efficient than OpenAI-o1 and 5 times faster." The R1-0528 refresh and DeepSeek-V3 (which "tops the Chatbot Arena open-source leaderboard with an Elo score of 1,382") were released in 2025. [12]

Reasoning models: DeepSeek-R1, DeepSeek-R1-0528 [14]

2.4.3 Claude

Claude 4 Sonnet integrates "multiple reasoning approaches" and leverages an "extended thinking mode" using "deliberate reasoning or self-reflection loops." This allows iterative refinement of outputs for accuracy. It has a "200K-token context window" (over 3× larger than Claude 2) for processing "lengthy documents or extensive conversations," showing "strong improvements in coding and front-end web development." [12]

Reasoning model: Claude 4 Sonnet, Claude 4 Opus [15]

2.4.4 Gemini

Gemini 2.5 features "enhanced reasoning capabilities" with a "1 million token context window" and "self-fact-checking features." It generates "fully functional applications and games from a single prompt" and handles text, images, and code. As a proprietary model, it raises security concerns for sensitive data. Google's open-source alternative Gemma 3 supports "context windows up to 128,000 tokens" and comes in "1 billion, 4 billion, 12 billion, and 27 billion parameters." [12]
Reasoning models: Gemini 2.5, Gemini 2.5 pro [16]

2.5 Agentic Tools and AI Agents

2.5.1 Definition of agentic tools and agent-based systems

AI Agents are self-operating programs built to tackle specific tasks alone [17]. Agentic AI goes further with its teams of specialized AI working together, breaking down big goals and coordinating actions like a human squad would.

Agentic AI systems exhibit "autonomy, reactivity, proactivity, and learning ability" [18]. They operate independently through "planning, learning, and environmental data to make decisions without necessarily requiring direct human intervention," enabling complex task execution [19]. Hierarchical architectures organize interactions across multiple agent levels (e.g., master, orchestrator, micro-agents).

2.5.2 The idea of "tool use" in LLMs

LLMs stay limited until they're given tools like search APIs or code executors [17]. When stuck, they call these tools to grab live data or run tasks, turning them from talkers into doers. Think of it like giving them hands to interact with the world.

Agentic AI integrates with external tools (e.g., email, code execution, search engines) via "multi-agent systems" to perform diverse tasks [18]. Systems like LangChain and AutoGen enable "language understanding, reasoning, planning, and decision-making,"

optimizing workflows. This tool integration facilitates "complex workflow" creation and automation.

2.5.3 Why tool-use makes LLMs more interactive, context-aware, and capable

Tools let LLMs act checking real-time info, scheduling, or analyzing data [17]. This bridges the gap between thinking and doing, making them adaptive problem-solvers instead of just text generators. They break out of their static boxes.

Tool use enhances interactivity by enabling dynamic exchanges (e.g., answering questions via image analysis) [18]. It improves context-awareness by combining text, images, and sound for richer interactions. Capability expands through workflow optimization, including resource allocation and automation opportunities [20].

2.6 Popular Agentic Frameworks

2.6.1 CrewAI

The paper describes CrewAI as a framework for building teams of AI agents. It says that CrewAI lets users define agents with specific roles and goals. The framework focuses on structured collaboration between these specialized agents to solve complex tasks, according to the authors. CrewAI provides ways to sequence agent interactions within a team. [18]. For updated releases about CrewAI, visit their official website for documentation. [21]

2.6.2 Auto-GPT

According to the paper, Auto-GPT is an open-source framework that allows LLM agents to operate autonomously. It is recognized for breaking down high-level user goals into smaller, actionable sub-tasks automatically [22]. The paper explains that Auto-GPT agents work iteratively: they plan actions, execute them (such as performing web searches), and learn from the results to improve their approach. [18]

2.6.3 IBM Watson

The paper presents IBM Watson as an early leader in enterprise AI. Its main capability, the authors point out, is processing large amounts of text-heavy, unstructured information and making sense of it. Watson is used for complex tasks that require deep knowledge, such as answering difficult questions or analyzing data in specific fields. It is especially useful in areas like healthcare and legal work, where reviewing many documents is essential. [18]

2.6.4 Microsoft AutoGen

The paper describes Microsoft AutoGen as a tool for creating conversational agents. It serves as a way to build AI helpers designed for specific tasks. One major advantage is its flexibility; you can arrange different methods for these agents to communicate with each other or with people [23]. The framework allows you to adjust how the agents behave and uses large language models (LLMs) to support conversations, incorporating them into specific step-by-step processes. [18]

2.6.5 Comparisons and use cases

Table 2-1: Comparison of different AI Agent Frameworks (Author)

Framework	Primary Focus	Collaboration Style	Typical Applications
Auto-GPT	Autonomous task decomposition	Solo agent, iterative refinement	Research automation, exploratory tasks
CrewAI	Role-based multi-agent teams	Structured team interaction	Project management simulation, workflows
Microsoft AutoGen	Customizable conversable agents	Flexible conversational patterns	Building chatbots, multi-agent dialogues

IBM Watson	Enterprise knowledge processing	Integrated system components	Healthcare Quality Assurance (QA), legal doc analysis, domain-specific insights

2.7 Reasoning Models and Chain-of-Thought Techniques

2.7.1 Introduction to reasoning models and prompting techniques.

2.7.1.1 Reasoning Models

Large Language Models (LLMs) have progressed beyond basic token generation by introducing the concept of "thought," which is a sequence of tokens that represents intermediate steps in reasoning. This allows LLMs to imitate human reasoning processes, such as tree search and reflective thinking. [24]

2.7.1.2 Prompting Models

Methods like Chain-of-Thought (CoT) and Tree-of-Thoughts (ToT) clearly direct models to create intermediate reasoning steps. These techniques greatly improve performance on tasks that need structured and multi-step reasoning, even without extra training. [24]

Chain-of-thought prompting improves reasoning by having models generate step-by-step rationales. This method works without finetuning but only emerges in very large language models. [25]

2.7.1.3 Limitations of LLMs

LLMs have difficulty with complex reasoning and matching outputs to human expectations. They rely heavily on high-quality labeled data, which is costly to produce, and can experience catastrophic forgetting during fine-tuning. Test-time methods also lead to increased token use and higher computational demands. [24]

Creating high-quality rationales is costly, and there is no guarantee that the generated reasoning paths are correct. The approach is also costly to deploy as it only works with large-scale models. [25]

2.7.2 Chain-of-thought, self-reflection, ReAct (Reason + Act)

2.7.2.1 Chain-of-Thought

Chain-of-thought (CoT) reasoning is a method where large language models (LLMs) generate a sequence of steps to derive an answer from a question. However, this approach is seen as a "static black box" because the model relies only on its internal representations to produce these thoughts. It does not connect to the outside world. This limitation can lead to significant problems, such as fact hallucination and error propagation during the reasoning process. For example, in tasks like multi-hop question answering, CoT is better at forming the reasoning structure but can easily produce made-up facts or thoughts. This results in a high false positive rate in success modes and is a major failure point [26]

2.7.2.2 ReAct (Reason + Act)

The ReAct paradigm combines reasoning and action by prompting LLMs to produce verbal reasoning traces and specific actions in an alternating way. This lets the model reason dynamically, create plans, and adjust them while interacting with external environments for more information. One key advantage of ReAct is its better interpretability and trust. People can distinguish between the model's internal knowledge and the information it gathers from external sources. However, ReAct also has some drawbacks. While blending reasoning and action improves groundedness, this setup can limit flexibility in forming reasoning steps, which may lead to more mistakes. A common error in ReAct is that the model repeats earlier thoughts and actions instead of determining the next appropriate step. Additionally, finding useful information through search is crucial for ReAct. If the search results are not helpful, it can disrupt the model's reasoning and make recovery more difficult [26]

While this can enhance performance, research questions its necessity, showing that bypassing the explicit thinking process can lead to better accuracy-token and accuracy-latency tradeoffs in many cases [27]

2.7.3 Role in decomposing tasks, planning, and decision-making

Chain-of-Thought (CoT) prompting is a method that helps large language models solve complex tasks by breaking them into smaller, sequential reasoning steps. This approach resembles how humans think when tackling multi-step problems [25]The essence of this method is to provide the model with examples of a problem, the reasoning process, and the final answer. This technique allows the model to use more computational resources for challenges that require more reasoning steps and has been shown to improve performance on tasks involving arithmetic, commonsense, and symbolic reasoning. [25]

In multicultural teaching, a similar method called Multicultural Prompt Learning (MPL) is applied. This method breaks down the complex task of creating multicultural content into smaller sub-tasks, such as defining cultural requirements, identifying key elements like cultural background, symbols, and architecture, and providing guidance for each element. This step-by-step breakdown helps the model build content gradually, ensuring that the final output is logically coherent and culturally authentic. [28]

Additionally, applying CoT supports planning and decision-making. For example, in the SayCan robot planning dataset, CoT prompting mapped a natural language instruction to a sequence of robot actions. The model generates an explanation that details its reasoning before creating a step-by-step plan. This highlights its usefulness in complex decision-making situations that involve multiple steps [25].

3 Methodology & Implementation

This chapter outlines the design, technological basis, and implementation of the prototype system, which was created to automate project planning and code generation using reasoning models and agent tools.

3.1 System Overview

3.1.1 High-Level System Description

The system is a modular approach to an AI-powered application, which is designed to automate planning and development phases of SDLC using LLMs and Agentic tools. Some of the key features of the system are:

- Accepts structured or unstructured requirements as input and delivers a structured project plan
- Allows refinement of the plan
- Converts plans into actionable JIRA tickets
- Generates modular code snippets based on tickets or user requirements

3.1.2 User Interaction Flow

A high-level flow of how the user uses the application is as follows:

- User submits the project requirement either by filling out a form or by uploading a file
- AI agents use that input and create a well-structured prompt to give to the LLM
- LLM generates a structured project plan
- The user can refine the generated plan multiple times until they are satisfied
- User can use the “Suggest JIRA Tickets” option and get suggested tickets from the system
- The user can edit/add/delete the tickets and add them to the JIRA dashboard

- Users can select the code snippets they want from the development tasks defined under various categories such as frontend, backend, database, etc.

3.1.3 Core System Components

The system consists of a frontend interface using React, a backend service built with Python, an LLM integration layer that employs OpenAI's o3, and an agent orchestration engine based on CrewAI. These components work together to allow for dynamic task handling, agent coordination, and interactive output generation.

3.2 System Architecture

The system has a modular and layered architecture to improve scalability and maintainability. It uses a client-server approach where the frontend is responsible for accepting user inputs and managing interaction flows, while the backend is responsible for managing AI-driven services, including tasks such as agent-based task delegation and LLM generation. The architecture is capable of handling both structured inputs using a form and unstructured inputs using file uploads, enabling flexibility to the user. Agentic modules reason over the input data independently and extract domain-specific insights, which are then combined into a well-structured prompt for the LLM. This reasoning process ensures that the final output is comprehensive, context-aware, and aligned well with the current best practices. The system also has a feature for iterative refinement of the generated output, enabling an interactive and evolving planning experience.

3.2.1 High-Level Architecture Overview

At a high level, the system consists of the following components:

Frontend Interface (React): Provides a modern user interface for entering requirements, viewing the project plan, refining the plan, generating tickets, and interacting with generated code.

Backend Server (FastAPI - Python): Acts as the main controlling layer for the system. It handles inputs from the frontend, manages API routing, triggers agent workflows, and supports interactions with the LLM.

Agent Orchestration Engine (Crew AI): Responsible for the coordination of multiple AI agents to sequentially perform specialized tasks such as analyzing inputs, structuring project plans, breaking down tasks, extracting best practices that are being followed, and many more.

Data processing & Normalization: Ensures both structured and unstructured inputs are transformed into a standard format.

LLM Integration Layer (OpenAI o3): Handles all the language model interactions, such as project planning, refinement of the plans based on user feedback, and code generation using OpenAI o3 reasoning model. This layer is responsible for sending the requirements given by the agent modules to the OpenAI API and presenting them to the user. o3 is chosen for its reasoning capability, low-latency performance, and cost-effectiveness.

Figure 3.1 shows the high level architecture of the application.

3.2.2 Data Flow and Component Interaction

A typical flow of data within the system is as follows:

Input Submission: The user submits requirements either using the form or by uploading a requirements file.

Normalization: Unstructured inputs from files are parsed and transformed into a standard format.

Crew AI Orchestration: AI agents are activated in a sequential order to generate a structured and well-organized prompt by reasoning using multiple parameters.

Plan Generation: LLM receives the prompt from the agents and generates a structured project plan as a response.

Plan Delivery and Refinement: The plan is presented to the user, who may refine the entire plan or selected sections or ask to generate tickets for the system.

Ticket Generation: Based on the final version of the plan, tickets are generated and suggested to the user, where the user may modify them and submit them to the dashboard.

Code Generation: The user has pre-defined options, upon choosing which, the system generates code snippets for that selection.

3.2.3 Architecture Diagram

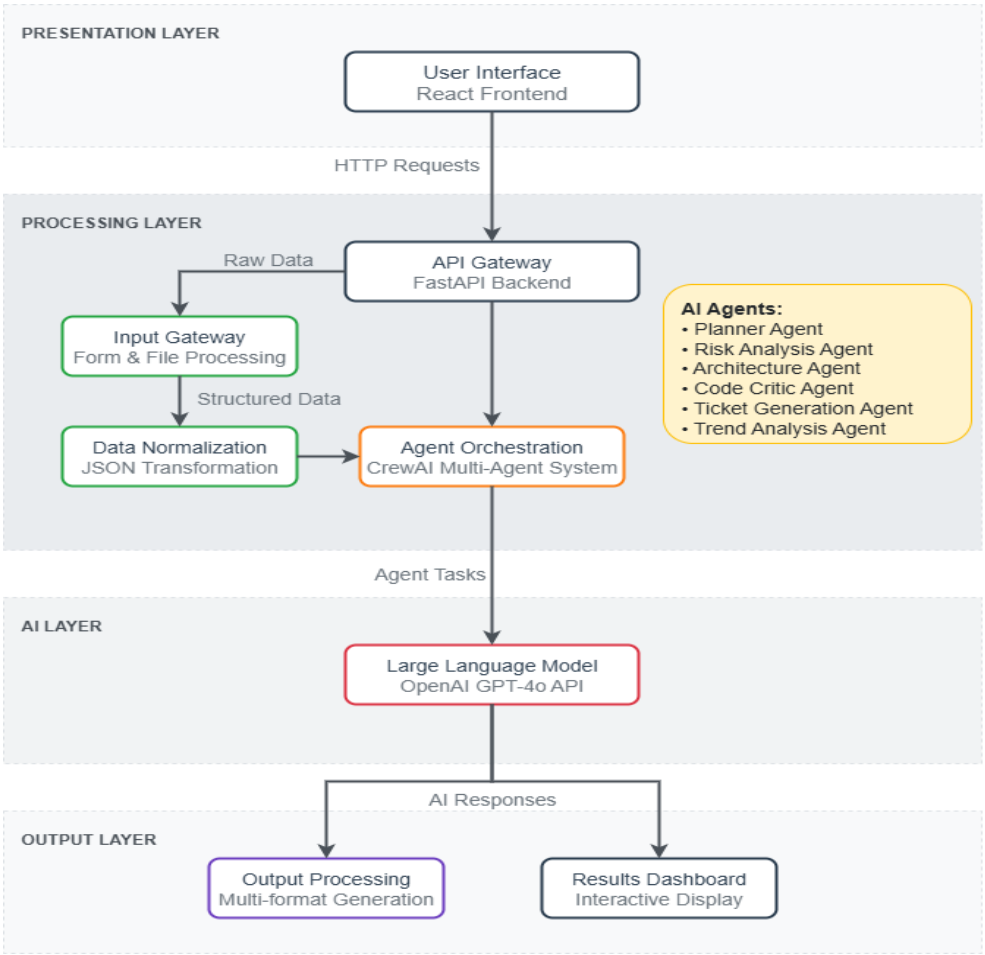

Figure 3.1: High-Level Architecture Diagram (Author)

3.3 Technology Stack

The system is built using a modular technology stack tailored for real-time interaction and AI integration. Each technology was selected due to its strength in handling web applications, AI capabilities, and asynchronous communication between the services.

3.3.1 Frontend

React: It is a widely used JavaScript library for building interactive and innovative User Interfaces (UI). Thanks to its component-based architecture, virtual DOM for optimized performance, and a very strong community support, it is very easy to build dynamic, fast, and scalable UIs, especially for single-page applications [29]. It has a complete ecosystem with tools such as router, redux, and a widely supported library of third-party components. It allows a developer to break down complex UI tasks into manageable sub-tasks, reuse code efficiently, and maintain a clean codebase.

In this thesis, it is responsible for rendering forms, capturing user inputs, displaying generated project plans, JIRA tickets, code, etc.

Material UI: It is a React component library that offers designs based on Google's Material Design that are customizable and that make the UI look attractive. It speeds up the development process and ensures that the UI is consistent with its ready-to-use elements and the built-in responsiveness in them [30].

3.3.2 Backend

FastAPI (Python): It is a high-performance modern web framework for building APIs with Python. Since it's built on top of Pydantic and Starlette, it has powerful async capabilities and automatic data validation. Speed, intuitive syntax using Python type hints, and automatic generation of interactive API docs via Swagger UI and ReDoc are some of the reasons why developers prefer FastAPI.

Its capability of handling requests asynchronously makes it a perfect choice for managing long-running tasks, such as LLM queries and agent orchestration. The documentation for FastAPI was very helpful during the project's implementation [31].

Uvicorn: An Asynchronous Server Gateway Interface (ASGI) server used to run the FastAPI application, offering support for asynchronous, non-blocking Input/Output (I/O).

3.3.3 Language Model Integration

OpenAI o3: This is the primary LLM used in the system. Its reasoning capabilities, tool-use potential, and low-latency performance make it the most well-suited model to use for this thesis. Its main role in this thesis is to power project generation and plan refinement.

3.3.4 Agent Orchestration

CrewAI: A framework based on Python used to create and manage AI agents with different roles. Agents operate in sequence or parallel, share context, and are managed using CrewAI's task and memory management tasks [21]. In this thesis, a few tasks assigned to the agents are Project Intake Analyst, Business Object Mapper, Risk Identifier, etc.

3.3.5 Utilities and Supporting Libraries

Pydantic: A very powerful Python library used for data validation and parsing using type hints. A very popular library among modern Python applications where clean and reliable data structures are expected, such as APIs, machine learning pipelines, etc.

Python Standard Libraries: Used for parsing input files, managing I/O, and formatting outputs.

Markdown and JSON Handlers: Utilities to format AI-generated content for frontend rendering.

3.3.6 Development Tools and External References

Visual Studio Code (VS Code): Used as the primary development environment for developing the system for both the frontend and backend. It offered integrated support for Python and Typescript, making it easy to streamline the development process.

3.4 Module 1: Project Planning

3.4.1 Overview

The project planning module serves as the most foundational component of the system. The primary purpose of this module is to generate a detailed and well-structured project plan using the user-provided software requirements as inputs via a pre-defined form in the system or via unstructured text files that the user already has. The generated plan is structured in a way that contains very important deliverables such as Executive Summary and Project Charter, Business Goals and Objectives, Work Breakdown Structure (WBS), Risk Assessment and Mitigation, Architecture Recommendation, Timeline and Sprint Plan, Resources and Team Structure, Budget Allocation, Quality and Governance Plan, and Best Practices and Modern Trends.

Within the overall system architecture, this module acts as the main entry point for automation. It sets the path for all the following processes, including ticket generation and code generation. By combining structured data processing, LLM-based reasoning, and multi-agent orchestration, this module transforms high-level business ideas into actionable development tasks.

One of the interesting features of this module is its support for iterative interaction. Users can refine the generated project plan by submitting feedback about the changes, additions, or deletions they want to the current plan until they get a result with which they are satisfied. The feedbacks given by the users are directly given to the LLM for regeneration or specific updates. Additionally, users can also use the feature of automated ticket generation that generates tickets based on the current plan, enabling seamless transition from planning to task management.

3.4.2 Input Methods

This module accepts two primary methods for collecting input from the users: **form-based structured input** and **file-based unstructured input,** enabling flexibility for the users to define the project requirements using either the interactive UI or by uploading a specification document.

3.4.2.1 Form-Based Structured Input

This is a pre-defined form on the system's UI that the users can leverage to enter their project ideas. The data that the form captures are as mentioned below:

Project Overview Section:

- **Project Name (*):** As the name suggests, it is a plain text field where the user must enter a name for their Project. This is a mandatory field.
- **Client or Stakeholder Name:** This field expects the name of the client or stakeholder for whom the project is being developed. It is a plain text field. This is not a mandatory field.
- **Project Category:** Specifies the domain of the project. (e.g., Internal, Client-facing). This is a dropdown field. This is not a mandatory field.
- **Project Description (*):** A brief overview of the project's purpose, goals, scope, and any other essential data that can help LLM understand the requirements of the user properly and align with their vision. It is a plain text area for users to freely enter their requirements. This is a mandatory field.

Figure 3.2 shows the page where the users enter the Project Overview details.

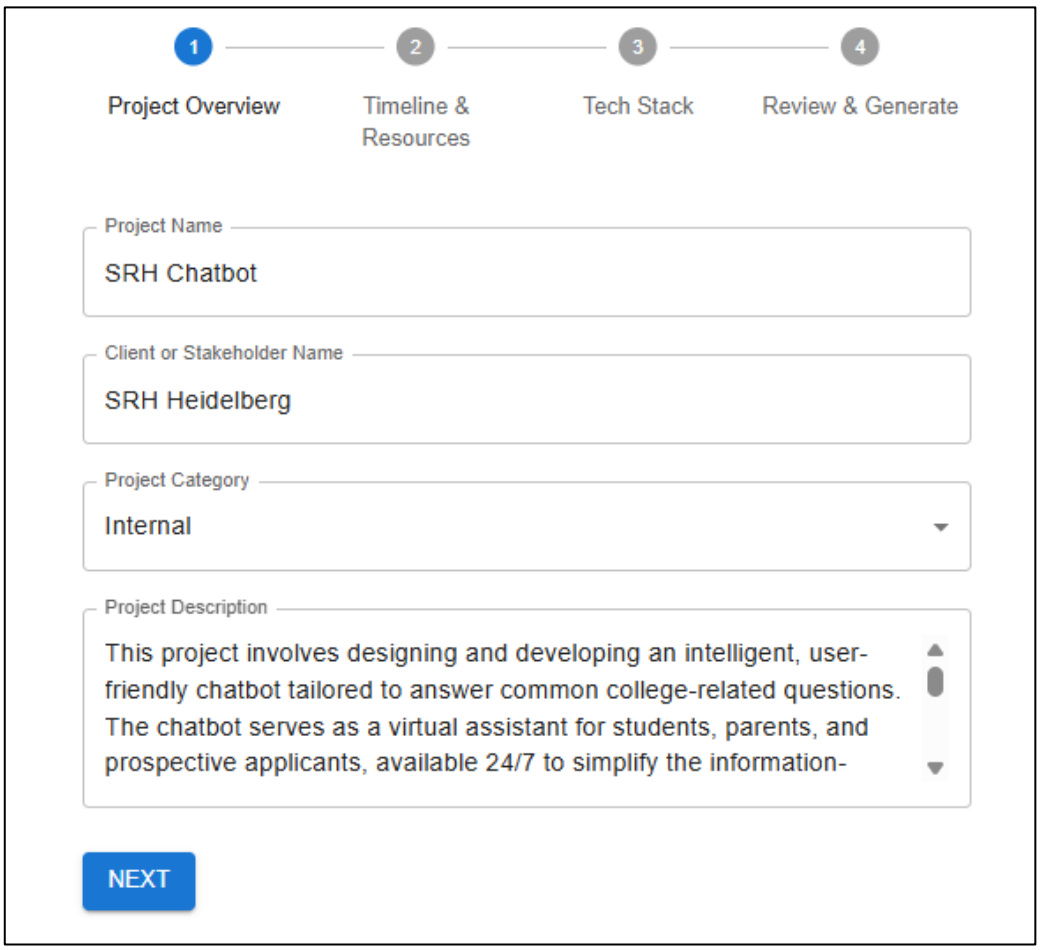

Figure 3.2: Project Overview Section (Author)

Timeline and Resources Section:

- **State Date (*):** Captures the start date of the project. This field is required for scheduling. It is a typical date select field. This is a mandatory field.
- **Expected Duration:** Captures the user's expectation about when the project must be completed. This is a plain text that accepts only numbers with a dropdown indicating options such as days, weeks, and months. This is not a mandatory field.
- **Team Size:** This field expects the number of team members involved in the project execution. It is a plain text field that accepts only numbers. This is not a mandatory field.

- **Budget in Euros (*):** This field captures the allocated financial resources for the project in terms of euros. It is a plain text field that accepts only numbers. This is a mandatory field.
- **Team Experience:** This field captures the aggregate of the team's experience. This is a dropdown with the options: Beginner, Intermediate, and Advanced. This is not a mandatory field.
- **Team Location Type:** This is a field that is used to determine whether the team will be working onsite, remote, or hybrid across regions. This is not a mandatory field.

Figure 3.3 shows the page where the users enter the Timeline & Resources details.

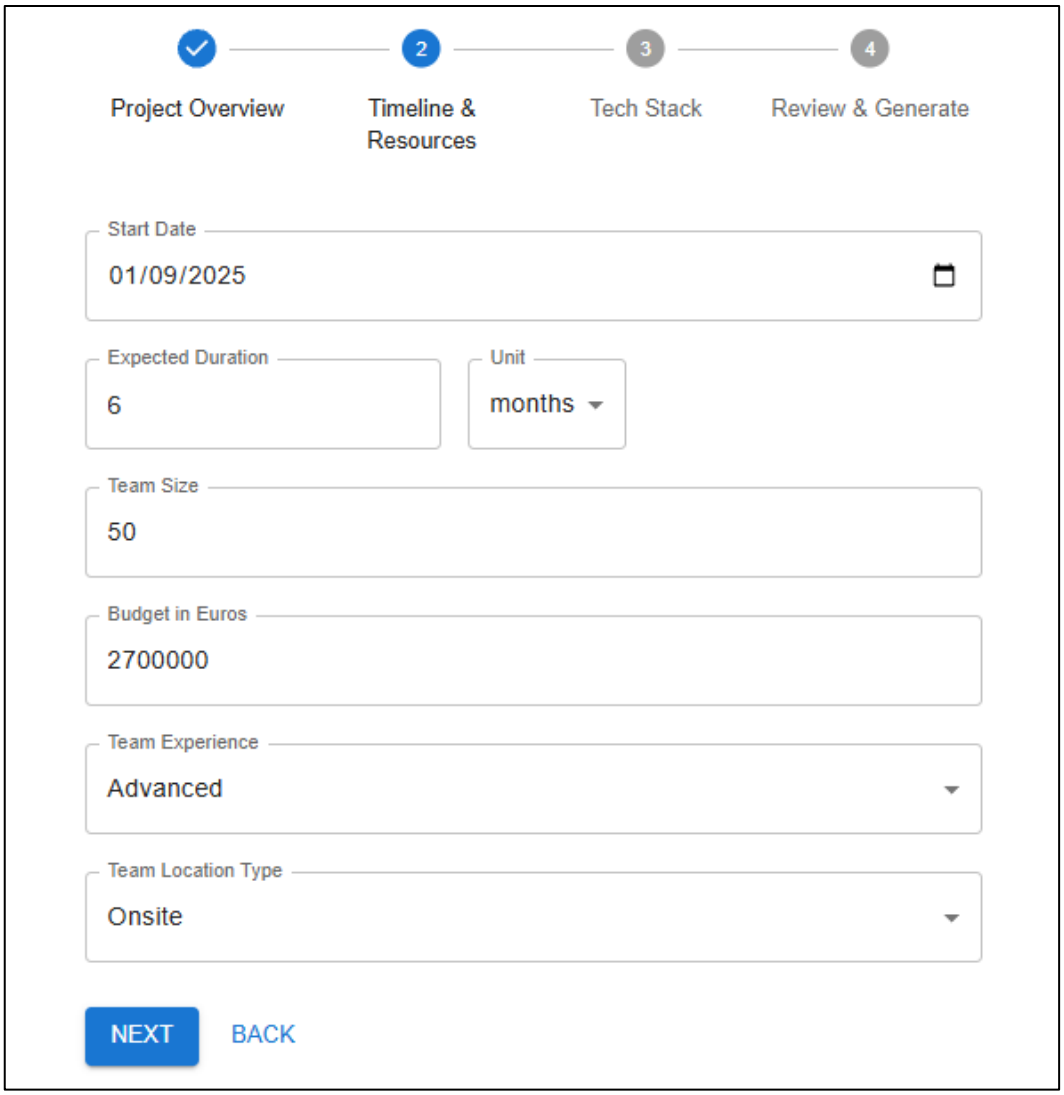

Figure 3.3: Timeline & Resources Section (Author)

Technology Stack Section:

All the fields in this section are multiple checkboxes and non-mandatory. The user can either select more than one option, in which case the LLM will determine the most well-suited technology for their project. If the user does not select any of the options, the LLM will make a decision for the project based on the agent's inputs.

- **Frontend Technologies:** The users can select the most advanced frontend frameworks that they prefer. The available options are React, Angular, and Vue.js.
- **Backend Technologies:** The users can select the most advanced backend frameworks that they prefer. The available options are Node.js, C#/.Net and Go.
- **Database Technologies:** The users can select the most advanced database frameworks that they prefer. The available options are SQL, MongoDB, and Firebase.
- **Cloud Technologies:** The users can select the most advanced cloud frameworks that they prefer. The available options are AWS, Azure, and Google Cloud.
- **DevOps Technologies:** The users can select the most advanced DevOps frameworks that they prefer. The available options are Docker, Kubernetes, and Jenkins.
- **Design Tools:** The users can select the most advanced design tools that they prefer. The available options are Figma and Adobe XD.
- **Other Technologies:** This is a plain text where the user can enter any other technologies that they prefer, such as AI integration or libraries such as Material UI, etc.

Figure 3.4 shows the page where the users enter the Technology Stack details.

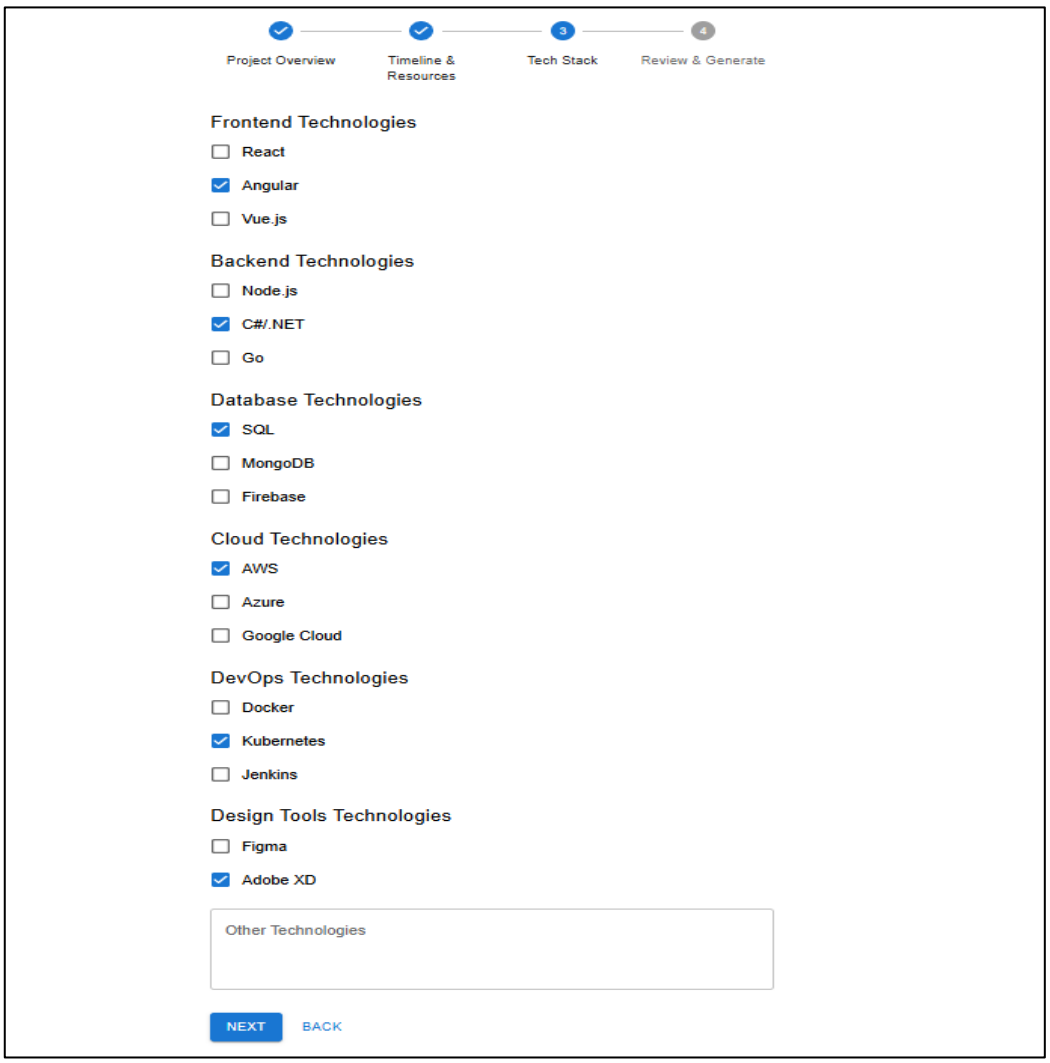

Figure 3.4: Technology Stack Section (Author)

3.4.2.2 File-based Unstructured Input

As an alternative to the predefined form, users can upload unstructured requirement documents. These files can be in different forms, such as .txt, .pdf, and .docx. These files may contain business goals, technical expectations, or a natural language description of the requirements they have for their desired project. Once uploaded and submitted, the following process is carried out on the uploaded file:

- The file is parsed using a backend service
- The text is extracted and cleaned if necessary

- The content extracted from the file is forwarded to a data normalization pipeline (described in section 3.4.3)

This method of input is useful for users who have their requirements documented in a file and do not prefer to fill out a form. Also note that the current system does not consider images. Only files through which text can be extracted are parsed.

3.4.2.3 Input Validation and Schema Enforcement

All inputs, irrespective of whether structured or unstructured, are validated against a unified schema using Pydantic on the FastAPI backend. The validation schema enforces:

- Required fields (for form inputs)
- Supported file types
- Acceptable value types (lists, string, enums)
- Character limits to avoid prompt overflows

Any input that does not adhere to these validation rules is rejected, and a descriptive error message is returned to the frontend so that any normal user might understand what went wrong and how to fix it.

3.4.3 Data Normalization

The data normalization is one of the most important stages, which is responsible for transforming the user input, irrespective of whether it is structured or unstructured, into a unified format that can be further processed by the CrewAI agents and LLMs. This makes sure that all further module work with a consistent and semantically rich representation of the user's requirements, regardless of the original input source.

3.4.3.1 Normalization Process

The system uses different paths to normalize an input based on the type of the input:

Structured Input Pathway (Form → *`/api/generate-project-plan`*):

- Data arrives as a validated ***ProjectInput*** *Pydantic model*

- This data is fed into ***build_crew(),*** a function used to construct a task-oriented CrewAI pipeline with relevant agents.
- The outputs generated by the agents are combined into a high-level prompt using ***build_prompt_from_agents()***.

Unstructured Input Pathway (File → text input, not yet formalized in backend):

- call_llm_to_extract_json_from_free_text(), a function used to parse raw free-text briefs.
- The extracted content is cleaned and validated into a ProjectInput model.
- This data is fed into ***build_crew(),*** a function used to construct a task-oriented CrewAI pipeline with relevant agents.
- The outputs generated by the agents are combined into a high-level prompt using ***build_prompt_from_agents()***.

Unified Prompt Preparation:

Irrespective of the user's choice of path, both structured and unstructured inputs are ultimately translated into a well-refined LLM prompt through templates. This acts as a functional normalization layer where task definitions and formatting rules guide the LLM to interpret input in a consistent, agent-augmented context.

Output of Normalization:

The final output of this stage is a consistent prompt that acts as a starting point for project planning via OpenAI's endpoint. This prompt is designed in the following way:

- Executive Summary & Project Charter
- Business Goals and Objectives
- Work Breakdown Structure (WBS)
- Risk Assessment and Mitigation
- Architecture Recommendation
- Timeline and Sprint Plan
- Resource and Team Structure
- Budget Allocation

- Quality and Governance Plan
- Best Practices and Modern Trends

Figure 3.5 shows the flow of the input normalization process.

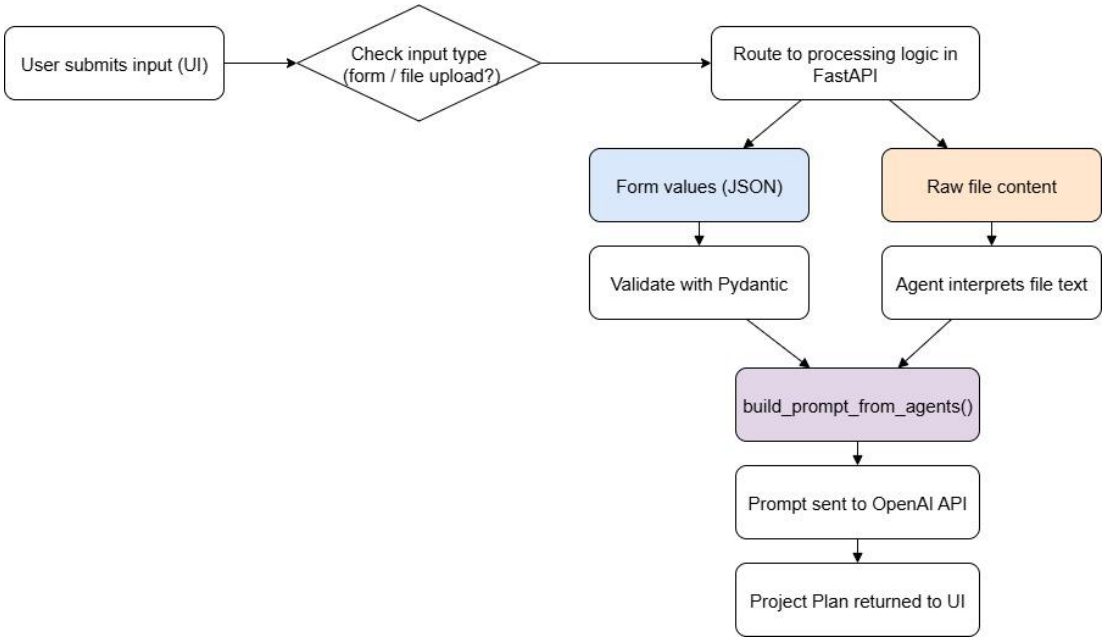

Figure 3.5: Input Normalization Flow (Author)

3.4.4 Agentic Workflow (CrewAI Orchestration)

The system uses an agentic orchestration framework known as CrewAI to coordinate reasoning and output generation for two key phases: project planning and JIRA ticket generation. CrewAI allows modular agent definition and sequential task execution, ensuring the output is structured and is well aligned with the user's expectation.

3.4.4.1 Agent Usage in the System

Currently, the system has agents for:

Project Planning (/api/generate-project-plan): Multiple agents (defined in build_crew) process the normalized form inputs and collaborate to generate a structured project plan

JIRA Ticket Generation (/api/generate-jira-tickets-from-plan): A single agent (ticket_generator_agent) which uses the finalized project plan and derives a list of JIRA tickets to suggest for the user.

3.4.4.2 Agentic Workflow: Project Planning Phase

In the project planning phase:

- The system constructs a CrewAI instance via the *build_crew() function* using the structured *ProjectInput* coming from the frontend.
- The agents defined in the crew are assigned different roles, such as mapping business goals, flagging project risks, suggesting optimal architecture, breaking the project into sprints, injecting modern best practices etc,.
- These agents operate in a sequential mode using the *crew.kickoff() function*, which means that each agent processes the output of the previous agent.
- The final output of these agents is used to construct a well-structured prompt and sent to OpenAI's o3 model to generate a project plan.

3.4.4.3 Agentic Workflow: Ticket Generation Phase

The ticket generation flow (*/api/generate-jira-tickets-from-plan*) is simpler:

- A single agent (ticket_generator_agent) is initialized with a system task to break down the project plan into actionable, JIRA-friendly tasks.
- It receives the project plan that is finalized by the user as input and is instructed to return a list of JIRA-styles tickets in the form of JSON.
- That output is parsed, cleaned, and returned to the frontend.

3.4.4.4 Task Sequencing and Control

Task sequencing is implemented by:

- Ordering the tasks in the CrewAI task list
- Agents passing output to each other using a shared context
- Sequential orchestration mode which is set using *process="sequential"* to guarantee pre-defined processing order.

3.4.4.5 Context and Memory Sharing

CrewAI internally manages context propagation between agents. This allows:

- Each agent is to access relevant user requirements to carry on with its task.
- Downstream agents to leverage prior decisions (e.g., *Timeline Estimator* to use prior decisions of agents like *Risk Identifier*, *Architecture Recommendation*)
- A shared memory model maintained as a structured Python dictionary.

3.4.4.6 Orchestration Logic in Backend

The initiation of the orchestration takes place with the FastAPI route handlers:

- *build_crew(data)* sets up the planning agents.
- *Crew.kickoff()* runs the task sequence.
- *ticket_generator_agent* is manually passed into a *Crew()* with a specific task for ticket generation.
- Error handling and output validation (e.g., JSON cleanup) are also handled in these routes.

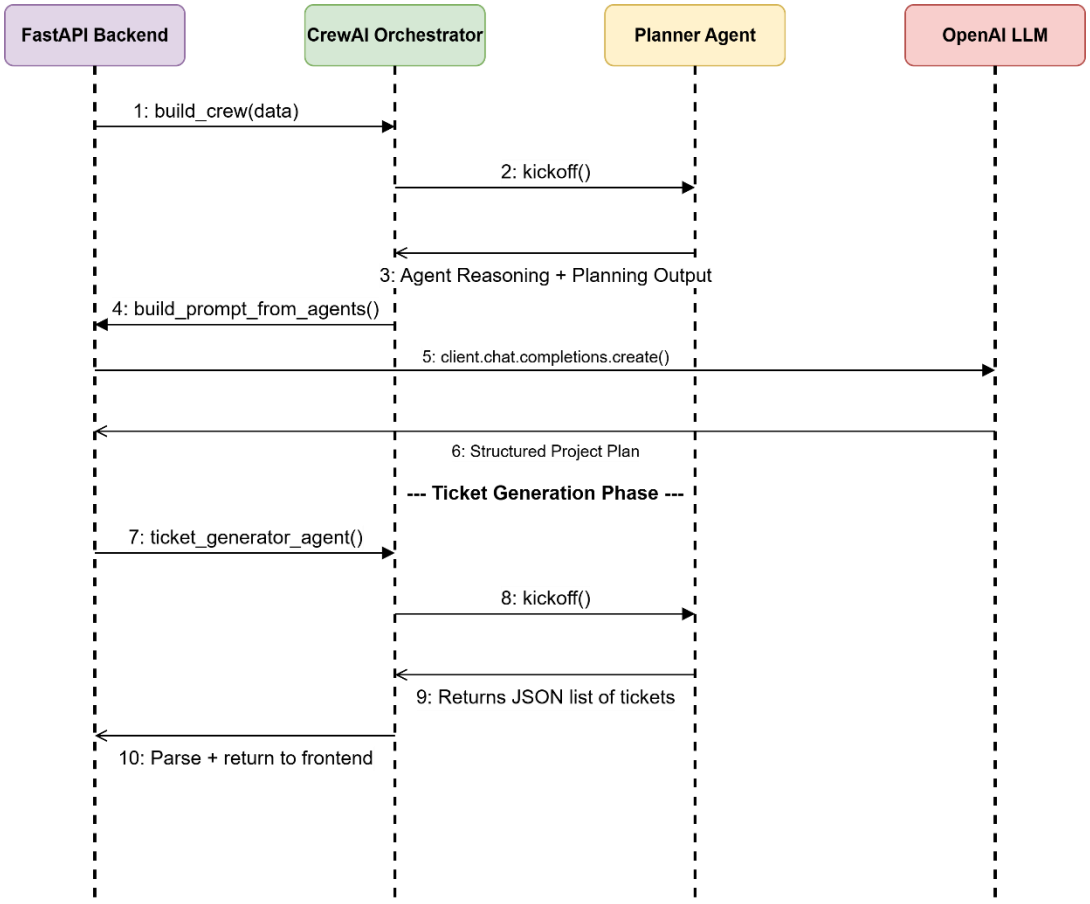

Figure 3.6: Agentic Workflow Sequence Diagram (Author)

3.4.5 LLM Prompting & Generation

The Large Language Model (LLM) part of the system creates the initial project plan and processes plan changes based on user feedback. This happens through direct prompts to OpenAI's o3 model using the OpenAI API. In contrast to the agentic workflow, which manages multi-step planning and ticket creation, the LLM is used for its skill in generating natural language plans and can adjust them based on human input using its reasoning capabilities.

3.4.5.1 Prompt Design Strategy

Prompts sent to the LLM are designed in a way to balance clarity and adaptability. Each prompt includes two main parts:

System Role Content: This defines the LLM's identity and sets limits on its behavior (e.g., "You are a software project planner. Extract the following…").

User Role Content: This contains either the regular user input or an adjusted plan with user feedback for improvement.

This separation helps the LLM to consistently grasp its context and what is expected in its output, even after several rounds of refinement.

3.4.5.2 Initial Project Plan Generation

For the first version of the project plan:

- The normalized input JSON from the agents is transformed into a natural language description
- This description is then embedded into the user role prompt
- The system then instructs the model to generate a well-structured markdown-formatted plan with information such as project name, executive summary, business goals, risk assessment, WBS, architecture recommendation, etc..
- The response of LLM is parsed and displayed in the frontend dashboard

3.4.5.3 Model Selection and Performance

- OpenAI's o3 was selected as the main model because of its strong reasoning capabilities, improved speed, and low latency.
- The parameters of the API, such as Temperature and max token limits, are tuned for reliability and soundness.
- The system keeps a balance between cost and performance by using LLM selectively. It only applies to tasks that need open-ended language generation or adjustment.

3.4.6 Plan Refinement and Ticketing Features

After the system generates the initial project plan based on the user input, the system provides the user with 2 key features: **plan refinement** and **automated ticket generation.** These features help user to refine the plan multiple times to meet their need and then transition from the planning phase to creating tasks for the development.

3.4.6.1 Plan Refinement

Once the initial plan is generated, the users can refine it by submitting additional feedback using the free-text area in the frontend interface. The system supports both major and minor refinements, such as:

- Modifying a specific section
- Replacing an entire section
- Adding new information in the current sections or adding a new section

The refinement process follows the following steps:

- User submits feedback using the free text area in the frontend.
- The backend takes the feedback and the current project plan and creates a well-structured prompt for refinement.
- The o3 model is called to regenerate the plan with the given feedback.
- The updated plan is then sent back to the frontend to be displayed to the user.

No agents are used during the refinement phase as o3 is directly responsible it.

3.4.6.2 Ticket Generation Workflow

Once the user is happy with the final version of the project plan, they can click the “Suggest JIRA Tickets” button in the UI. This action then triggers the agents that are responsible for using that plan and creating sprint-wise development tasks. The key steps involved here are:

- The finalized plan is received as input to the backend, and it is passed to the CrewAI-based agent workflow.

- Agents designed for this purpose analyze the project plan thoroughly and convert it into development tickets.
- Each ticket includes a title and a description.
- Generated tickets are then sent to the frontend for the user to update/delete/add new tickets if needed, and then publish them to their JIRA dashboard.

3.4.7 Output Format

The system produces two primary outputs: the **project plan** and the **development tickets.** These outputs are structured in such a way that they support both readability and can be used for further processing.

3.4.7.1 Project Plan Output

The project plan generated by the LLM is in the form of a well-structured markdown format, which ensures that the plan can be easily rendered in the frontend while retaining clarity and hierarchy.

The project plan includes the following sections:

- Executive Summary & Project Charter
- Business Goals and Objectives
- Work Breakdown Structure
- Risk Assessment and Mitigation
- Architecture Recommendation
- Timeline and Sprint Plan
- Resource and Team Structure
- Budget Allocation
- Quality Assurance and Governance Plan
- Best Practices and Modern Trends

Formatting of the output:

- Headings and Subheadings use Markdown (##, ###) to organize the content.
- Lists user bullet or numbered formats

3.4.7.2 Development Ticket Output

The main purpose of this feature was to show the possibility that the agents have the ability to generate tickets based on a given input by scanning it thoroughly. Each ticket has two basic fields, namely a title and a description.

3.5 Module 2: Code Generation

3.5.1 Overview

The code generation module is the final stage of the automation pipeline. The main goal of this module is to transform the project plans and tickets from the Planning module into concrete code snippets. While the planning module focuses on defining goals and tasks, this module is designed to focus mainly on helping developers with ready-to-use code snippets that align with the project's technology stack and architecture.

The workflow of this module is organized into two layers. In the first layer, development tasks are extracted and categorized from the finalized project plan by specialized agents. These tasks are then grouped into different categories such as frontend, backend, database, cloud, DevOps, and design. This categorization will help the user to easily have control over what type of code they want the system to generate.

In the second layer, the LLM is responsible for generating actual code snippets. Each selected task is then converted into a prompt that includes data such as task metadata, the relevant section of the project plan, and detailed instructions for producing code in a structured JSON format. The LLM then generates and returns code specific to the language that can be rendered in the dashboard.

This combination of agentic task decomposition and LLM-based code generation in this module ensures that the generated code snippets are not only relevant to the context but also aligned properly with the project plan. This represents a very important step towards bridging the gap between automated planning and automated development.

3.5.2 Task Extraction & Categorization (Agent Layer)

The first step in the code generation module is to scan the finalized planning document, identify, and organize the development-specific tasks. This goal is achieved by the combination of agents designed for extraction and categorization.

3.5.2.1 Development Task Extraction

The ***Development Task Extractor*** is responsible for filtering out all the non-technical tasks, such as documentation, stakeholder analysis, risk mitigation, etc., and only keeps the tasks that are relevant to software development.

These extracted tasks are then structured in a JSON form to ensure compatibility with downstream processing. The Figure 3.7 represents a small sample of how the tasks are extracted from the plan and structured.

```
[
  {
    "task_name": "Implement user authentication",
    "task_description": "Develop a login and registration module using JWT-based
authentication"
  },
  {
    "task_name": "Design database schema",
    "task_description": "Create relational schema for users, tasks, and projects using
PostgreSQL"
  }
]
```

Figure 3.7: Task Extraction Agent Output (Author)

3.5.2.2 Task Categorization

Once the tasks are extracted, they are then grouped into different categories based on the architecture and technology stack mentioned in the project plan. The categories include:

- Frontend (UI components, routing, state management, etc.)
- Backend (API endpoints, services, etc.)
- Database (schema design, stored procedures, etc.)
- Cloud (monitoring, scaling, etc.)
- DevOps (CI/CD pipelines, containerization, etc.)
- Design (UI/UX guidelines, prototyping)

This step makes sure that the structured prompts for code generation are specialized, accurate, and relevant. The **Figure 3.8** is a small sample on how the extracted tasks are categorized.

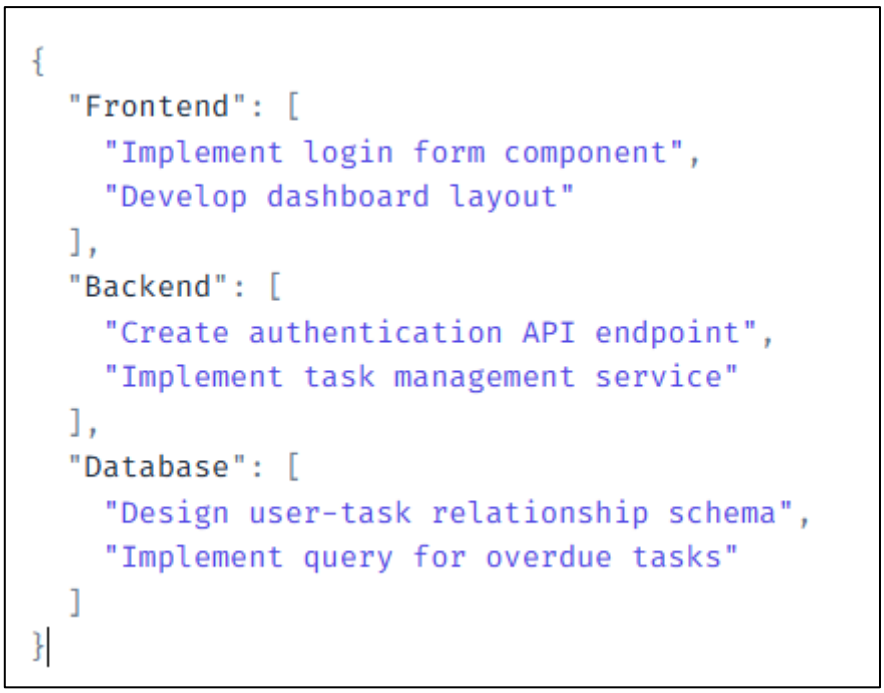

Figure 3.8: Task Categorization Agent Output (Author)

Role in Code Generation

- Categorization offers more detail for the code generation phase. It ensures that each snippet fits the specific technology.
- This also allows developers to choose tasks from a category they like, making the system more interactive and modular.

Figure 3.9 depicts the pipeline of how development tasks are extracted from a project plan and categorized.

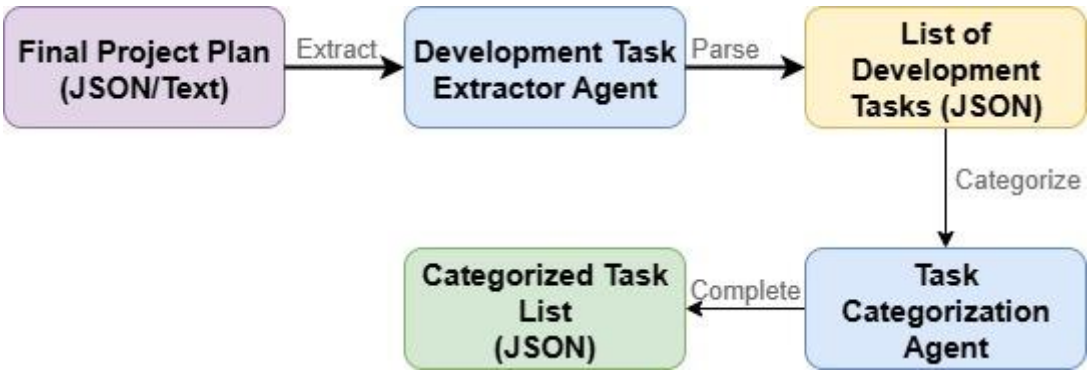

Figure 3.9: Task Extraction and Categorization Pipeline (Author)

3.5.3 Code Prompting & Generation (LLM Layer)

The main responsibility of the LLM layer is to convert a single task that is prepared by the agent layer into a code snippet that is executable and that aligns with the architecture and technology stack of the project. The backend constructs a prompt with two parts (system + user) and calls OpenAI o3 and instructs it to generate an output with a particular JSON schema so that the frontend can easily render the code.

3.5.3.1 Prompt Design

- **System Role:** This role establishes a developer persona, style, safety constraints, and the required JSON schema as the output from the response that the LLM provides. Figure 3.10 shows how a system role is set.

```
{
  "role": "system",
  "content": "You are a precise full-stack developer."
}
```

Figure 3.10: System Role (Author)

- **User Role:** The user role includes data such as the finalized plan, the selected task with the title and description, and other hints such as framework, language, etc. Figure 3.11 shows how a user role is set.

```
prompt = f"""
You are an experienced senior software engineer. Based on the project plan below, generate
a code snippet that addresses the following task.
### PROJECT PLAN {data.final_plan}
### TASK NAME {data.task_name}
### TASK DESCRIPTION {data.task_description}
Use the tech stack (e.g., frontend/backend/frameworks/database/cloud) referenced in the
project plan. Your response must:
-
-
-
Be relevant to the described task.
- Include one high-quality code snippet.
{{
Include the appropriate language in your response.
Return ONLY a valid JSON object in the format:
"task": "Task name",
"language": "Python | JavaScript | SQL | etc.", "snippet": "your code here"
}}
{"role": "user", "content": prompt}
```

Figure 3.11: User Role (Author)

- **Output Contract:** The model returns its response in a strict JSON format. **Figure 3.12** shows a sample of the model response.

```
{
  "task": "Implement user authentication",
  "language": "Typescript",
  "snipper": "import {useEffect, useState } from 'react'; # ...rest of the code ..."
}
```

Figure 3.12: Output contract (Author)

3.5.3.2 Model & Parameters

The system uses **OpenAI's o3** model for code generation for its reasoning capabilities and speed, as it is an important factor since multiple code snippets might need to be generated during the software development stage.

The model is configured with a **temperature** of **1** to maintain determinism and reproducibility. This makes sure that the outputs of a selected task that are generated multiple times are consistent and do not have major variations in them.

Due to the explicit definitions of **system and user roles**, the mode works with a controlled prompt template and ensures that the response is always aligned with the project's plan, task description, and technology stack. This configuration minimizes hallucinations and makes sure that the code generated is trustworthy.

3.5.3.3 Context Narrowing

Hallucinations are often seen in the LLM responses due to the passing of data that is irrelevant to the task that you need the LLM to do. Hence, to minimize the likelihood of hallucinated outputs, the system uses a **context-narrowing strategy.**

In this strategy, instead of passing the entire project plan, the backend selectively takes out the sections that are relevant to the given task. For example, if the task is to create a login page in the frontend, the system takes the frontend architecture, technology stack, and excludes other sections, such as database schema, etc., that are unrelated.

This helps the model concentrate on the given task and does not let it drift into unnecessary domains. Additionally, by passing the category metadata (e.g., Frontend, Backend), the model has a clear path to produce accurate solutions.

3.5.3.4 Safety & Guardrails

The safety is ensured at the prompt level. The **user role** explicitly tells the model to generate a single code snippet that is relevant to the task and return it in a strict JSON format, with the fields task, language, and snippet. This instruction acts as a main guard and ensures that the response is in an expected structure every time.

The **system role** further strengthens it by specifically mentioning that the LLM is a "precise full-stack developer", which encourages focused and accurate output. No other security checks are applied apart from these prompt-based constraints.

3.5.3.5 Error Handling & Retries

Once the LLM model sends the response, the backend attempts to parse it directly as JSON using *json.loads()*. If it fails to parse the response for some reason, such as Markdown code fences, etc., a cleanup step interferes and strips away JSON or markers before retrying again. If it is still invalid, the process throws an error as Hypertext Transfer Protocol (HTTP) 500 and returns the details of the error to the client. Currently, the implementation does not include any retries, repair prompts, or schema validation.

3.5.4 Code Generation Workflow

The code generation workflow in the development module is structured in such a way that it helps users to view the tasks in a very organized manner. The implementation is a sequence of API calls that narrow down the user selection from categories such as Frontend, Backend, etc., to specific tasks related to those categories, and finally to the code generation upon selection of any particular task. The workflow steps are as described below:

3.5.4.1 Category Retrieval

- When the user loads the development screen, the frontend calls the ***/api/get-dev-categories,*** which is a POST endpoint.
- The backend scans the final project plan and extracts the development categories, such as Frontend, Backend, Database, etc., from the technology stack section and returns them in a JSON format.
- These categories are then rendered in the user interface. Task Listing by Category
- Upon selection of a category, the frontend then calls the ***/api/get-tasks-by-category,*** which is another POST endpoint.
- The request body specifies the details of the chosen category.

- The backend returns the tasks that are relevant to that category in a JSON array format
- These tasks are displayed to the user as selectable cards.

3.5.4.2 Code Generation by Task

- When the user selects a task, the frontend calls the final POST endpoint ***/api/generate-code-snippet.***
- The request body for this endpoint includes:
    - final_plan: The finalized project plan
    - task_name: The name of the task selected
    - task_description: Description of the selected task
- Using this body, the system constructs a well-structured system role and user role and hits the o3 endpoint.
- The response received is parsed as JSON.
- If the parsing is successful, the backend returns the JSON object; else, it throws a HTTP 500 error.

Figure 3.13 shows the sequence diagram of how the code generation module works. It depicts the process flow that takes place in the backend of the application during the code generation module.

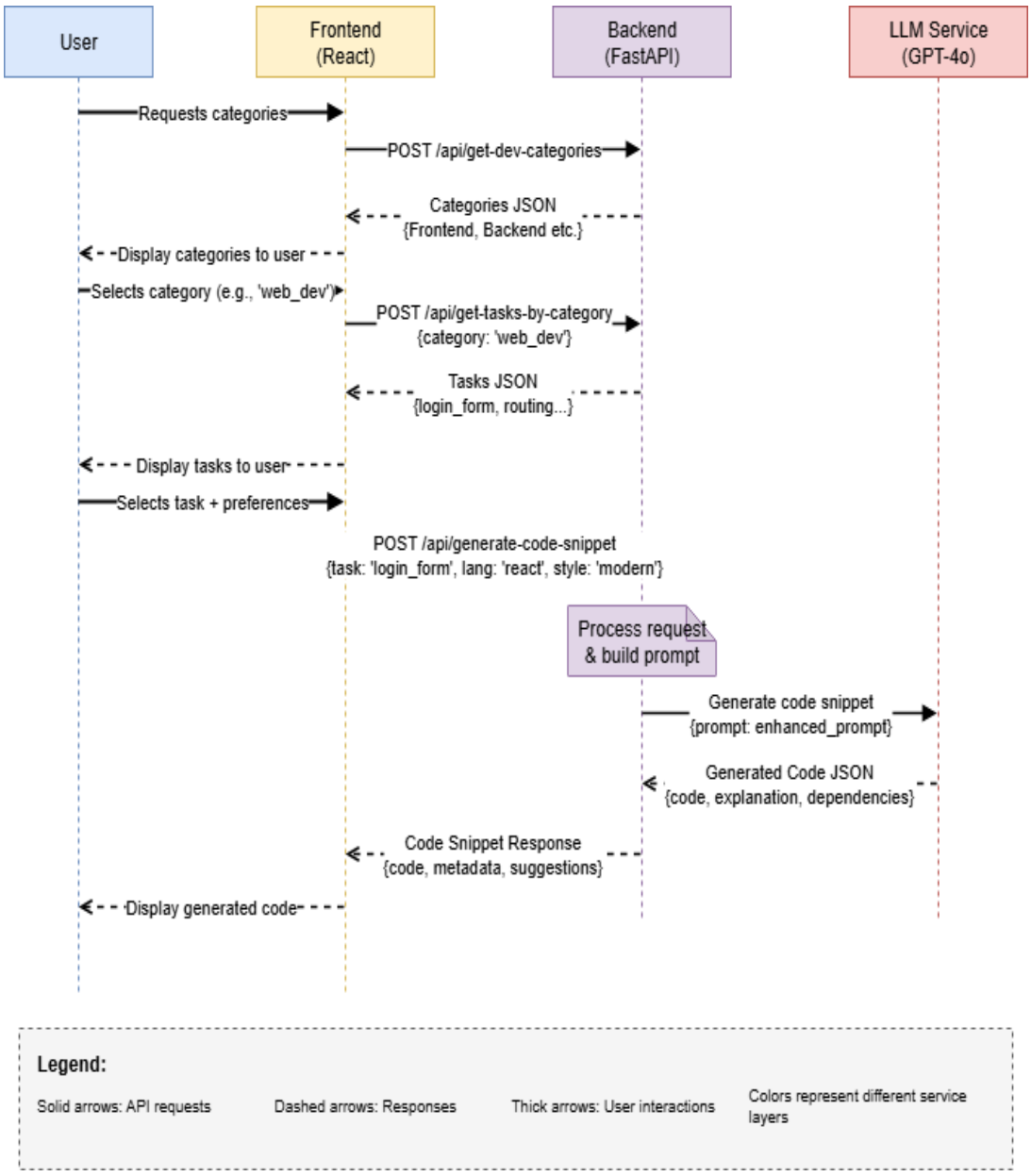

Figure 3.13: Code Generation Workflow - Sequence Diagram (Author)

3.6 Integration Flow

3.6.1 Overview

The integration flow shows how the gap between the **Project Planning Module** and the **Project Development Module** is bridged. Both modules serve distinct purposes. The planning module focuses on understanding the user-given requirements and structuring them into a plan. The development module, on the other hand, focuses on delivering implementation-level code. The system's overall objective is achieved only when these two modules are integrated and work in a smooth pipeline.

The structured project plan generated in module 1 is used as the foundation for module 2. This ensures that the transition from high-level project planning to low-level code generation is consistent, context-aware, and automated. Without this integration, they would work as independent components, and this would require the users to do a lot of manual work, and also, the continuity of the flow is lost in this process.

Thus, the integration of these two modules ensures a user experience where requirements specified at the beginning directly influence the code snippets produced at the end.

3.6.2 Linking Project Planning to Development

The key to the integration flow is how the structured project plan from Module 1 serves as the foundation for all activities in Module 2. By converting the project requirements into a machine-readable format, the system keeps planning decisions intact and reusable during development. This ensures both consistency and traceability.

3.6.2.1 Project Plan Representation

At the end of the planning phase, the system creates a structured project plan. This plan is saved as a JSON object and includes:

- Categories (e.g., Authentication, Database, Frontend)
- Tasks linked to each category

- Task descriptions, including details about expected functionality

This format supports automatic processes because APIs can process it directly without needing additional interpretation. The standardized format also allows both form-based and file-based inputs in Module 1 to fit into the same structure.

3.6.2.2 Role in Development

The project plan serves as the single source of truth for Module 2. All APIs in the development phase, whether fetching categories, listing tasks, or generating code, depend on this stored representation. By connecting back to the plan:

- Categories shown in the development UI come directly from the plan
- Task descriptions guide the code generation prompts
- Context from the overall plan ensures that generated snippets align with the original project requirements

This approach guarantees a smooth connection between what was planned and what is implemented, making the workflow reliable.

3.6.3 API Workflow

The connection between the Project Planning module and the Code Generation module relies on a set of APIs. These APIs enable smooth movement from requirement specification to task planning and then to code snippet generation. All APIs are designed as POST methods to handle flexible JSON payloads.

3.6.3.1 Planning Module APIs

The following endpoints are in charge of creating and refining the project plan:

- POST ***/api/generate-plan***, creates a structured project plan from form-based or file-based inputs.
- POST ***/api/refine-plan***, updates or improves the generated plan based on user feedback.
- POST ***/api/suggest-jira-tickets***, suggests JIRA tickets based on the structured plan.

- POST ***/api/submit-tickets***, sends suggested tickets to the project dashboard.

These APIs make sure that user inputs are turned into a clear and actionable plan that serves as the basis for later development tasks.

3.6.3.2 Development Module APIs

The following endpoints use the project plan to help with development tasks:

- POST ***/api/get-dev-categories***, pulls categories (e.g., Database, Frontend) from the stored plan.
- POST ***/api/get-tasks-by-category***, gets a list of tasks associated with the selected category.
- POST ***/api/generate-code-snippet***, creates a language-specific snippet for a given task using o3.

These APIs ensure that the structured plan is not just theoretical but can be acted upon, allowing the user to explore tasks and obtain usable implementation snippets.

3.6.3.3 Integration Role of APIs

The key to integration is the shared project plan, which is produced in Module 1 and used in Module 2. Planning APIs create, refine, and store the plan, while development APIs access it to show categories, tasks, and generate code. This guarantees that:

- No redundant user input is needed.
- The generated code always matches the original requirements.
- The workflow runs as a single, continuous process.

Error! Reference source not found. gives a quick overview of all APIs used in the s ystems in one place. With the input it takes and the output it provides.

Table 3-1: API Summary Table (Author)

Endpoint	Input	Output
/api/generate-plan	Form/File input (require-ments)	Structured project plan (JSON)

/api/refine-plan	Plan + user feedback	Updated project plan (JSON)
/api/suggest-jira-tickets	Project plan	Suggested tickets (JSON)
/api/submit-tickets	Tickets JSON	Confirmation / Dashboard update
/api/get-dev-categories	Project plan reference	List of categories (JSON)
/api/get-tasks-by-category	Category name	List of tasks (JSON)
/api/generate-code-snippet	Final plan + task details	Task, language, snippet (JSON)

3.6.4 User Interaction Flow

The integration of planning and development is both technical and user-focused. From the user's viewpoint, the workflow aims to reduce redundancy, ensure continuity, and allow a smooth transition from defining requirements to exploring code snippets.

3.6.4.1 Planning Phase UI

- Users start by entering project requirements through a form or by uploading a free-text requirements file.
- Once submitted, the system normalizes the input, organizes workflows, and creates a structured project plan that appears on the planning dashboard.
- At this stage, users can also:
    - Refine the plan based on feedback.
    - Generate suggested JIRA tickets.
    - Submit tickets to the project dashboard.

3.6.4.2 Transition to Development

- After finishing the planning phase, the user moves to the Development screen.
- Upon entry, the frontend automatically calls the backend to retrieve categories from the stored plan using the POST ***/api/get-dev-categories*** endpoint.
- The categories are listed as selectable items, ensuring a direct connection to the work completed in planning.

3.6.4.3 Code Exploration

- When a user picks a category, the system calls POST **/api/get-tasks-by-category** to show the related tasks.
- Selecting a task triggers the POST **/api/generate-code-snippet** endpoint, which returns a validated code snippet in JSON format.
- The snippet appears in a syntax-highlighted code viewer.

This process makes sure that the user does not have to re-enter information at any stage. The planning results directly fuel the development features, leading to a smooth and guided experience from broad requirements to detailed implementation.

3.6.5 Detailed System Integration

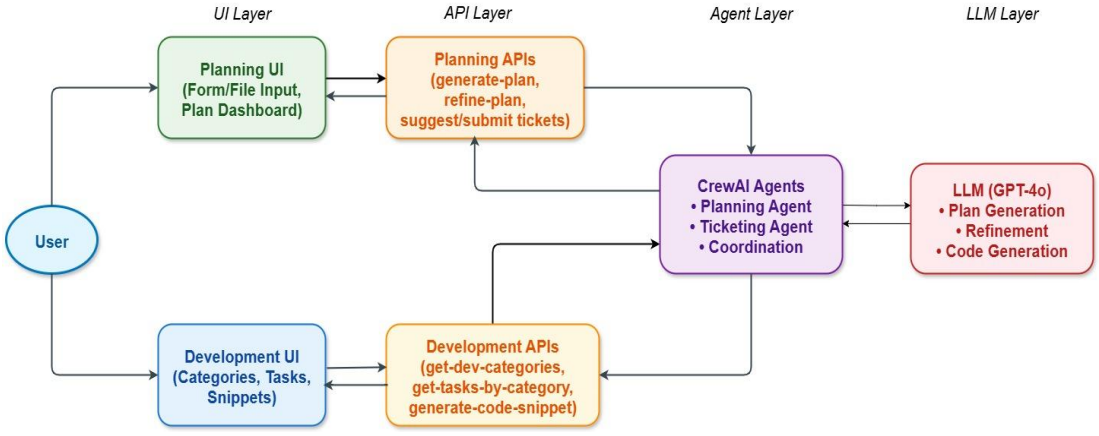

Figure 3.14: Detailed System Integration Flow (Author)

3.7 Summary

This chapter describes how to put into action a system that combines reasoning models and agent workflows to automate key parts of the software development lifecycle, especially project planning and development. The design was divided into separate modules, each handling specific tasks, while connecting smoothly through FastAPI-based backend services. The planning module allowed users to enter requirements in structured or unstructured formats. These requirements were normalized and processed through agent workflows, then enhanced with the reasoning capabilities of o3 to create actionable project plans. Features like refinement, ticket suggestion, and ticket submission further improved its usefulness by linking planning with execution-oriented workflows.

The development module builds directly on the planning results, letting users dive deeper into project requirements. Displaying the plan's categories and tasks enables code generation tailored to each task. APIs keep planning outputs and development inputs in sync, while the LLM delivers context-aware, high-quality code snippets. The entire workflow is interactive, allowing users to jump between categories and tasks and receive validated JSON responses in real time.

Integration across modules relies on a clear orchestration flow that ties together the user interface, backend APIs, CrewAI agents, and the LLM. This setup ensures each module's output becomes the next module's input, reducing redundancy and preserving context throughout the project lifecycle. Agent coordination handles input normalization, ticket creation, and workflow management, while assigning reasoning-heavy tasks to the LLM.

In summary, this implementation demonstrates a modular yet integrated approach to automating project planning and parts of the development workflow. The design shows how reasoning models and agents can collaborate to create a context-aware, end-to-end workflow for software engineering tasks.

4 Evaluation

4.1 Introduction

This chapter presents the evaluation of the implemented system, which centers on two key modules: project planning and project development. Both modules were assessed through user studies involving domain experts and practicing software developers. Structured Google Forms were used to guide participants through a series of questions designed to capture both quantitative ratings and qualitative impressions of the system-generated outputs.

4.2 Evaluation of Project Planning

4.2.1 Methodology

To assess the planning module a detailed project requirement was fed into the system to generate a structured project plan. This plan was shared with a group of experts to gather their feedback. The evaluation process aimed to measure both quantitative ratings and qualitative impressions of the plan.

4.2.1.1 Evaluation Instrument Details

Reference Document (PDF):

- The system-generated project plan was exported as a Portable Document Format (PDF) and provided to reviewers as reference.
- The PDF included all the major sections such as objectives, stakeholder analysis, risk assessment, quality governance, timeline etc.
- Headings, subheadings, and numbered lists were formatted for readability, allowing the reviewers to quickly locate sections referenced in the evaluation form.

Google Form:

- **Section 1: Likert-scale ratings** - Experts rated the project plan against ten different criteria.
- **Section 2: Open-ended questions** - They also described strengths, areas for improvement, and whether they would trust such a plan in practice or no.
- **Instructions:** Reviewers were asked to refer to the PDF and complete the form, ensuring that the ratings reflected the actual content.

4.2.1.2 Evaluation Questions

To ensure transparency, the exact questions used in the Google Form for the Planning Module are mentioned below. The questionnaire was divided into two sections: quantitative ratings which used a 5-point Likert scale and qualitative feedback through open-ended questions.

Quantitative Analysis (Likert-scale) Questions

Participants were asked to rate the system-generated project plan on the following criteria:

- **Clarity**: The plan is clear and easy to understand.
- **Structure**: The tasks are logically grouped and well-organized.
- **Completeness**: The plan covers all major aspects of the project.
- **Actionability**: The tasks are described in a way that developers can act on them.
- **Relevance**: The plan matches the given project requirement.
- **Consistency**: The categories, tasks, and tickets are consistent and aligned.
- **Granularity**: The level of detail in the tasks is appropriate (not too broad or too fine).
- **Feasibility**: The proposed tasks are realistic and achievable with the described technology stack.
- **Readiness**: This plan could be used as a starting point for actual project work.
- **Overall Satisfaction**: Overall, I am satisfied with the generated project plan.

Qualitative Feedback Questions

Participants were asked to provide open-ended responses to capture richer insights:

- What do you think was missing in the generated plan compared to a manual project plan?
- What aspects of the generated plan were better than a manual project plan?
- Would you trust such a plan to guide a real project? Why or why not?

4.2.2 Quantitative Ratings Results

The aggregated scores from the experts showed that the system-generated project plan performed well across most criteria. Ratings for clarity and structure were among the highest, with several participants noting that the layout and organization made the plan easy to follow. Consistency and relevance also received strong ratings, indicating that the outputs aligned with the provided requirements and maintained a coherent style throughout.

On the other hand, criterias such as granularity and feasibility scored lower. This suggests that while the plan effectively captured the main elements of project planning, some experts felt it could benefit from more detailed breakdowns of work packages and effort estimations. Table 4-1 summarizes the average ratings across the ten evaluation criteria:

Table 4-1: Average Ratings for Project Planning Module

Criterion	Average Rating (out of 5)
Clarity	4.4
Structure	4.9
Completeness	4.3
Actionability	3.6

Relevance	4.4
Consistency	4.3
Granularity	3.7
Feasibility	4.4
Readiness	4.6
Overall Satisfaction	4.6

The results show that the generated project plan was generally well-received. The highest ratings were for structure (4.9) and readiness (4.6) which state that participants found the plan logically organized and can be used as a starting point for real projects. Overall satisfaction also received a strong average of (4.6), indicating approval of the system's usefulness.

However certain criterias scored slightly lower. Actionability (3.6) and granularity (3.7) were seen as areas that needed improvement. This suggests that while the plan provided a strong high-level overview, some expected finer detail and more in depth guidance on how to execute the tasks.

4.2.3 Qualitative Feedback

The qualitative feedback added depth to the numerical ratings and clarified how the generated plan was perceived in practice. Many reviewers appreciated how the system wove in a wide range of planning elements often overlooked in traditional project plans. In particular, the inclusion of technical details and architectural considerations felt refreshing, since these specifics are frequently missing from plans drawn up solely by project managers. This emphasis on real-world development realities gave the output a grounded quality, rather than limiting it to high-level business concerns.

Clarity and organization also emerged as a strong point. Participants found the structure easy to follow, with clearly defined sections that made it simple to jump to areas of interest whether business objectives, stakeholder mapping, or risk assessment. The work breakdown structure and the segment dedicated to business goals were repeatedly praised for reflecting practices common in professional settings. For many, these elements signaled that the plan was both thorough and aligned with industry norms.

Several areas for improvement were highlighted as well. The plan's cost breakdown lacked detail across categories like infrastructure, development, and resource allocation. Timelines appeared too sequential, even though many tasks such as front-end and back-end development often proceed in parallel. Budget and timeline estimates were noted as insufficiently specific or accurate, though reviewers acknowledged this is a challenge even in manual planning. It was suggested that risk analysis and architectural recommendations could be expanded to offer more actionable guidance, and that diagrams Gantt charts or flowcharts would make the plan more visually intuitive and presentation-ready.

Regarding trust, the consensus was that the generated plan would not serve as a final execution document but rather as a strong starting point. It covers all essential planning areas and provides enough substance to guide early-stage meetings. One reviewer pointed out that the plan in its current form is ideal for initial presentations, after which it can be tailored to specific project requirements. In this way, the module functions as a supportive assistant to human planners speeding up the early phases of project planning and reducing manual workload rather than replacing them.

4.3 Evaluation of Development

4.3.1 Methodology

The development module was evaluated using the same project plan as input. From this plan, the system generated categories, related tasks, and small code snippets

corresponding to those tasks. Feedback was collected to understand how well these outputs supported practical development work.

4.3.1.1 Evaluation Instrument Details

Reference Materials (Screenshots PDF):

- Screenshots of the generated categories, tasks, and code snippets were compiled into a PDF for reviewers.
- Content was organized by project sections, showing tasks under each category and associated code snippets with syntax highlighting.
- The PDF allowed developers to evaluate outputs without needing access to the live system.

Google Form:

- **Section 1: Likert-scale ratings** - Developers rated the tasks and code snippets against ten different criteria.
- **Section 2: Open-ended questions** - Asked about alignment with project objectives, practical usability, suggestions for improvement, and desired complementary features (e.g., visual diagrams, inline comments).
- **Instructions:** Developers were instructed to reference the PDF while completing the form, focusing on both content accuracy and practical usefulness.

4.3.1.2 Evaluation Questions

To clearly document the evaluation, the exact questions used in the Google Form for the Development Module are provided below. The questionnaire was divided into two sections: quantitative ratings using a 5-point Likert scale and qualitative feedback through open-ended questions.

Quantitative Analysis (Likert-scale) Questions

Developers were asked to rate the system-generated development outputs, including tasks and code snippets, on the following criteria:

- **Task Clarity**: The tasks are clear and easy to understand.
- **Task Relevance**: The tasks are relevant to the overall project.
- **Task Granularity**: The level of detail in each task is appropriate (not too broad or too fine).
- **Code Snippet Relevance**: The code snippets are relevant to the associated tasks.
- **Code Snippet Readability**: The code snippets are easy to read and understand.
- **Code Snippet Correctness**: The code snippets are logically correct and follow expected syntax.
- **Overall Usefulness**: The outputs (tasks and code snippets) are useful for starting the development work.
- **Overall Satisfaction**: Overall, I am satisfied with the development module's outputs.

Qualitative Feedback Questions:

Participants were also asked to provide open-ended feedback to capture more nuanced impressions:

- What do you think was missing in the generated tasks or code snippets?
- What aspects of the outputs were better than manual task creation or coding?
- Would you trust such outputs to guide a real project? Why or why not?

4.3.2 Quantitative Ratings Results

The aggregated responses from developers show that the system-generated tasks and code snippets were generally rated positively across all criteria. Task clarity received an average rating of 4.3, showing that participants found the generated tasks straightforward to understand. Task relevance followed with a score of 4.1, indicating that the tasks aligned well with the requirements of the project plan, though some reviewers felt they could have been more context-specific.

Participants rated the level of detail, or granularity, at an average of 4.0 out of 5. This reflects a solid foundation of task breakdown, though a few participants felt certain tasks could be split into even finer steps to boost clarity and execution precision.

On the code generation side, the relevance of snippets to their associated tasks scored a strong 4.3. Reviewers noted that the system generally produced code closely aligned with the intended functionality, reinforcing confidence in its task-specific output.

Readability averaged 4.2, indicating that the generated snippets were easy to understand without lengthy interpretation. Logical and syntactic correctness received a 4.1 rating, suggesting that the code structure was sound and well-formed, even if occasional tweaks would be expected in a real-world setting.

When asked about practical usefulness, participants gave an average score of 4.2, and overall satisfaction stood at 4.3. These ratings underscore the view that the system functions as a valuable assistant saving time and adding structure while still allowing room for human-led refinement. The consolidated results are presented in Table 4-2.

Table 4-2: Average Ratings for Project Development Module

Criterion	Average Rating (out of 5)
Task Clarity	4.3
Task Relevance	4.1
Task Granularity	4.0
Code Snippet Relevance	4.3
Code Snippet Readability	4.2
Code Snippet Correctness	4.1
Usefulness	4.2
Overall Satisfaction	4.3

4.3.3 Qualitative Feedback

The feedback from developers showed a mixed but generally positive response to what the system could do. A lot of people mentioned that the generated tasks and code snippets gave them something concrete to work with right from the start. What seemed to resonate most was how everything got sorted into clear categories - frontend stuff, backend work, and other technical areas. This made it much easier to jump between different parts of a project without losing track of where things stood. Several developers said this organization alone saved them considerable time during the initial setup phase.

The code generation feature got particularly good reactions, especially when it came to cutting down on boring, repetitive work. People found that basic logic and boilerplate code came out clean and usable, which freed them up to tackle more interesting problems. One developer mentioned how helpful it was for payment gateway integration - even with years of experience, having that initial scaffolding made the whole process less intimidating and more straightforward.

However, the feedback wasn't all positive. A recurring complaint was the lack of inline comments in the generated code. While most people found the code readable enough, they wanted brief explanations of what specific sections were doing. This would help junior developers or anyone working with unfamiliar frameworks. Some participants felt the system sometimes missed the nuances of specialized projects, producing outputs that were technically correct but felt too generic for their particular needs.

Trust levels varied quite a bit among the testers. Some developers felt comfortable using the generated content as a foundation, though they stressed the importance of reviewing everything before putting it into production. Others saw the outputs more as helpful suggestions than ready-to-use solutions. A few suggested adding visual elements like flow charts or architecture diagrams to make the overall design clearer.

The general consensus was that this development module works well as a practical time-saver, particularly in early development phases. People appreciated having structure and clarity, but they also recognized that human judgment and customization would always be necessary. Rather than replacing traditional planning and coding approaches, the system seemed most valuable as a supportive tool that reduces initial effort and provides a solid foundation for further development work.

4.4 Limitations

While the evaluation gives us useful insights into how the system performs, there are some important limitations to keep in mind. The participant pool was pretty small - only eight experts looked at the planning module, and fifteen developers tested the development side. Their feedback was solid and credible, but having more people from different backgrounds would give us a better picture of how well this actually works. It would be helpful to include people from various industries, different company sizes, or with varying levels of experience to see if we missed any major strengths or problems.

Another issue is that the code snippets were only checked for readability and whether the logic made sense, not whether they'd actually work in a real application. These were designed as code fragments, not complete working modules, so we evaluated them on their own rather than as part of a bigger system. This means we might be missing integration problems or issues that would only show up when the code actually runs in a real development environment.

We also didn't have anything to compare against directly. It wasn't realistic to stack our system-generated plans against actual manual plans that companies use, mainly because of confidentiality issues and the fact that every organization structures things differently. So while our results look promising, we can't really say how much better or worse this is compared to what humans produce.

Finally, our evaluation only covered the early stages - planning and initial development support. We still don't know how this would play out over the long haul, how teams would work with it day-to-day, or how it would handle maintenance and ongoing project work. Even with these limitations, the findings show pretty clearly that the system can be a useful tool for speeding up planning and development work, as long as human expertise stays in the picture.

4.5 Summary

This chapter looked at how well the system's two main parts actually work: project planning and development. For the planning side, experts reviewed a system-generated project plan and gave high scores for clarity, structure, relevance, and overall satisfaction, showing the plan was well-organized and hit the main requirements. Written feedback confirmed the system did a solid job covering governance elements like stakeholder analysis, risk assessment, and quality assurance, but experts wanted more detailed task breakdowns, better parallel task handling, and more accurate effort estimates - basically, it works as a decent starting point but needs refinement before actual execution.

On the development side, practicing developers found the tasks and code snippets consistently positive - clear, relevant, actionable tasks with readable, logically sound code that made sense in context. They saw real practical value for cutting down repetitive work, faster onboarding, and providing solid foundations to build on. Suggested improvements included adding inline code comments, better context-awareness for specialized projects, and visual aids like workflow diagrams to boost usability and learning.

The evaluation shows the system provides real value for early-stage project work by mixing automation with human judgment. The planning module gives a clear foundation while the development module saves setup time so teams can focus on complex or creative aspects of software development. Everyone agreed it works well as a reliable assistant rather than replacing human decision-making, with solid

potential for speeding up workflows and keeping things organized, though improvements are needed in adding detail, context adaptation, and enhanced documentation.

5 Conclusion

This thesis aimed to explore how recent improvements in reasoning models and agentic tools can help automate the software development lifecycle (SDLC). Although automation has made significant strides in areas like code generation, the larger potential of combining reasoning and agentic systems across various development phases is still not fully explored. By concentrating on project planning and certain development aspects, this thesis sought to fill this gap by designing, implementing, and evaluating a prototype system.

The proposed system effectively showed how reasoning models and agentic tools can work together to automate structured project planning and create development tasks along with code snippets. The implementation made use of FastAPI for the backend, CrewAI for orchestration, and OpenAI o3 as the reasoning model, resulting in a functioning prototype that could take a project requirement and translate it into actionable outputs. Importantly, the system not only generated plans but also enabled refinement and downstream development tasks, illustrating a flow from planning to code.

The evaluation, carried out with domain experts and developers, provided evidence of the system's usefulness and potential. Experts rated the project plans as clear, well-structured, and largely complete, while developers found the generated tasks relevant and the code snippets readable and contextually aligned. At the same time, the evaluation highlighted important limitations, such as the lack of deeper architectural details in planning and the partial executability of code snippets. These findings confirm both the promise of the approach and the areas where future improvement is needed.

On a broader level, the work presented in this thesis contributes to the ongoing discussion about how artificial intelligence can move beyond isolated assistance towards more integrated, context-aware systems that support entire workflows. By

showing that reasoning models and agentic tools can be combined to automate multiple phases of software development, this thesis provides an early step in that direction.

Working on this project has been quite a journey - challenging in ways I didn't expect, but ultimately rewarding. Building and testing a system like this really pushed my technical skills in AI-driven development, but it also gave me a much deeper understanding of just how complex software engineering actually is. The prototype clearly isn't going to replace human developers or project managers anytime soon, but it does show how AI tools can become genuinely useful partners in the process, cutting down on tedious work and freeing people up for more important tasks.

This thesis set out to explore whether reasoning models and agent tools could automate planning and development tasks, and the results suggest there's real potential here, even if we're still far from perfect solutions. The findings point to some clear directions for future research and improvements that could make these systems even more useful. As AI keeps getting better, the work presented here might end up being a stepping stone toward more comprehensive automation in software development - though that's still a ways off.

6 Future Scope

The generated project plans cover several key elements such as objectives, scope, work breakdown structures, timelines, stakeholders, resource allocation, and risk analysis. These components form a strong foundation and show that AI-driven planning can effectively address many aspects of traditional project management.

But there is still room for improvement. One major area lies in enhancing the depth and precision of the planning outputs. While the current plans are well-structured, they lack features such as detailed cost estimation, effort estimates for individual tasks, and a clear mapping of task dependencies. Including these things would make the plans more practical for real-world use.

Another area which is worth exploring is the connection between planning and development. Right now, the transition from project plans to development tasks works but it's loosely tied together. Strengthening this link could involve automatically mapping items from the work breakdown structure and milestones directly to development tasks. This would create a one-to-one relationship between high-level planning and actionable tickets, resulting in a more seamless and integrated workflow.

In the development module, future improvements could focus on making the generated code snippets more executable and reliable. The current system produces relevant and readable fragments, but they often cannot be run directly without adjustments. Adding automated validation, such as unit testing or sandbox execution of snippets, would increase developer trust and reduce manual effort.

The system could also benefit from richer output formats. Currently, the generated plans and tasks are presented in text form. Future versions could produce visual artifacts such as Gantt charts for timelines, WBS trees, or even Unified Modeling Language (UML) diagrams for system design. These multimodal outputs would align more closely with industry standards and make the results more accessible to stakeholders who prefer visual representations.

Finally, the evaluation process could be broadened. This thesis evaluated a single project case with a limited group of experts and developers. Expanding this to multiple domains and larger participant groups would provide more generalizable insights. In addition, comparative studies of different reasoning models and agentic frameworks could identify the best combinations for specific phases of the software development lifecycle.

In summary, the prototype presented in this thesis demonstrates the feasibility of using reasoning models and agentic tools for automating project planning and development. Future work can build upon this by deepening the planning detail, improving traceability to development, enhancing code executability, adding visual outputs, and expanding evaluation. Taken together, these improvements would bring the system closer to the vision of intelligent, end-to-end automation in software development.

References

[1] Antonios Saravanos and Ma/hew X. Curinga 2, “2308.03940v3,”, 2023.

[2] R. Lekh and Pooja, “Exhaustive study of SDLC phases and their best praxctices to create CDP model for process improvement,” in *2015 International Conference on Advances in Computer Engineering and Applications:* IEEE, 2015, pp. 997–1003.

[3] S. Chahar and S. Singh, “Analysis of SDLC Models with Web Engineering Principles,” in *2024 2nd International Conference on Advancements and Key Challenges in Green Energy and Computing (AKGEC):* IEEE, 2024, pp. 1–7.

[4] q. yas, A. Alazzawi, and B. Rahmatullah, “A Comprehensive Review of Software Development Life Cycle methodologies: Pros, Cons, and Future Directions,” *ijcsm*, pp. 173–190, 2023.

[5] Mohammad Ikbal Hossain, “Software Development Life Cycle (SDLC) Methodologies for Information Systems Project Management,” *IJFMR*, vol. 5, 2023.

[6] A. Agarwal, A. Agarwal, D. K. Verma, D. Tiwari, and R. Pandey, “A Review on Software Development Life Cycle,” *IJSRCSEIT*, pp. 384–388, 2023.

[7] Mithilesh Pandey, Preeti Kuniyal, and Govind Kumar, “A Comprehensive Study on Software Development Lifecycle and Systematic Analysis of It Challenges,”, 2022.

[8] S. Johnson and D. Hyland-Wood, *A Primer on Large Language Models and their Limitations*, Dec, 2024.

[9] M. A. K. Raiaan, M. S. H. Mukta, K. Fatema, N. M. Fahad, S. Sakib, and M. M. J. Mim, et al., “A Review on Large Language Models: Architectures, Applications, Taxonomies, Open Issues and Challenges,” *IEEE Access*, vol. 12, pp. 26839–26874, 2024.

[10] R Sahana Lokesh, “Open-Source Large Language Models: A Comprehensive Survey,” in *2025 International Conference on Machine Learning and Autonomous Systems (ICMLAS):* IEEE, 2025, pp. 1096–1101.

[11] A. Tamkin, M. Brundage, J. Clark, and D. Ganguli, *Understanding the Capabilities, Limitations, and Societal Impact of Large Language Models,* Feb, 2021.

[12] Shakudo, *Top 9 Large Language Models as of July 2025.* 2025, https://www.shakudo.io/blog/top-9-large-language-models, 2025.

[13] Open AI, *Documentation for Open AI Models*.

[14] DeepSeek, *Official Documentation of DeepSeek*.

[15] Claude, *Official documentation of Anthropic*.

[16] Google, *Documentation of Google Gemini*.

[17] R. Sapkota, K. I. Roumeliotis, and M. Karkee, *AI Agents vs. Agentic AI: A Conceptual Taxonomy, Applications and Challenges,* May, 2025.

[18] S. Hosseini and H. Seilani, "The role of agentic AI in shaping a smart future: A systematic review," *Array,* vol. 26, p. 100399, 2025.

[19] N. Krishnan, *AI Agents: Evolution, Architecture, and Real-World Applications,* Mar, 2025.

[20] K.-T. Tran, D. Dao, M.-D. Nguyen, Q.-V. Pham, B. O'Sullivan, and H. D. Nguyen, *Multi-Agent Collaboration Mechanisms: A Survey of LLMs,* Jan, 2025.

[21] Crew AI, *Documentation of Crew AI*.

[22] H. Yang, S. Yue, and Y. He, *Auto-GPT for Online Decision Making: Benchmarks and Additional Opinions,* Jun, 2023.

[23] Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, and E. Zhu, et al., *AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation,* Aug, 2023.

[24] F. Xu, Q. Hao, Z. Zong, J. Wang, Y. Zhang, and J. Wang, et al., *Towards Large Reasoning Models: A Survey of Reinforced Reasoning with Large Language Models,* Jan, 2025.

[25] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, and F. Xia, et al., *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models,* Jan, 2022.

[26] S. Yao, J. Zhao, D. Yu, Du Nan, I. Shafran, and K. Narasimhan, et al., *ReAct: Synergizing Reasoning and Acting in Language Models,* Oct, 2022.

[27] W. Ma, J. He, C. Snell, T. Griggs, S. Min, and M. Zaharia, *Reasoning Models Can Be Effective Without Thinking,* Apr, 2025.

[28] Y. Chen, Y. Li, and J. Yang, “The design of prompts driven by Chain of Thought in multicultural teaching,” in *2024 IEEE 17th International Conference on Signal Processing (ICSP):* IEEE, 2024, pp. 258–262.

[29] P. Keshari, P. Maurya, P. Kumar, and A. Katiyar, “Web Development Using ReactJS,” in *2023 5th International Conference on Advances in Computing, Communication Control and Networking (ICAC3N):* IEEE, 2023, pp. 1571–1575.

[30] F. L. Soares, T. A. Falcão, T. B. F. Souza, J. S. Queiroz, P. M. Furtado, and J. L. R. Neto, et al., “A React Style Guide Library for MUI Web Apps,” in *2023 9th International HCI and UX Conference in Indonesia (CHIuXiD):* IEEE, 2023, pp. 77–82.

[31] FastAPI, *Documentation of FastAPI*.

List of Abbreviations

SL.No	Abbreviation	Full Form
1	AI	Artificial Intelligence
2	AJAX	Asynchronous JavaScript and XML
3	ANOVA	Analysis of Variance
4	API	Application Programming Interface
5	ASGI	Asynchronous Server Gateway Interface
6	AWS	Amazon Web Services
7	BERT	Bidirectional Encoder Representations from Transformers
8	CASE	Computer-Aided Software Engineering
9	CDP	Custom Development Process
10	CI/CD	Continuous Integration/Continuous Deployment
11	CMMI	Capability Maturity Model Integration
12	CNN	Convolutional Neural Network
13	CoT	Chain-of-Thought
14	CSS	Cascading Style Sheets
15	DoU	Document of Understanding
16	ELK	Elasticsearch, Logstash, and Kibana
17	ER	Entity-Relationship
18	GPT	Generative Pre-trained Transformer
19	HLD	High-Level Design
20	HTTP	Hypertext Transfer Protocol
21	I/O	Input/Output
22	IT	Information Technology
23	JAD	Joint Application Development
24	JWT	JSON Web Token
25	LLD	Low-Level Design
26	LLM	Large Language Model
27	MoSCoW	Must/Should/Could/Won't
28	MPL	Multicultural Prompt Learning
29	MRQ	Main Research Question
30	MVP	Minimum Viable Product
31	NPM	Node Package Manager
32	OWASP	Open Web Application Security Project
33	PDF	Portable Document Format

34	PMC	Project Monitoring and Control
35	PMI	Project Management Institute
36	PP	Project Planning
37	PPB	Process Performance Baseline
38	PPM	Process Performance Model
39	PPO	Process Performance Objective
40	QA	Quality Assurance
41	QC	Quality Control
42	RAD	Rapid Application Development
43	ReAct	Reason + Act
44	REQM	Requirements Management
45	REST	Representational State Transfer
46	RNN	Recurrent Neural Network
47	SDLC	Software Development Lifecycle
48	SOW	Scope of Work
49	SQL	Structured Query Language
50	SRQ	Sub Research Question
51	SRS	Software Requirements Specification
52	T5	Text-to-Text Transfer Transformer
53	TDD	Test-Driven Development
54	ToT	Tree-of-Thoughts
55	UAT	User Acceptance Testing
56	UI	User Interface
57	UML	Unified Modeling Language
58	UX	User Experience
59	WBS	Work Breakdown Structure
60	ZAP	Zed Attack Proxy

Index of Tables

Index of Figures

Appendix


```
{
  "projectName": "MediConnect",
  "projectDescription": "MediConnect is a healthcare management platform designed to
streamline patient-doctor interactions. It will provide features such as appointment
scheduling, electronic health record storage, prescription tracking, and automated reminders.
Patients will be able to book appointments online, while doctors can manage their schedules
and securely access patient records. The system will integrate with third-party health APIs
(e.g., wearables for vitals tracking) and comply with GDPR and HIPAA regulations to ensure
data privacy and security. Both mobile and web platforms will be developed for
accessibility.",
  "stakeholder": "Hospital Admin, Doctors, Patients, IT Team, Investors",
  "category": "External",
  "startDate": "2025-07-01",
  "expectedDuration": "12",
  "durationUnit": "months",
  "teamSize": "150",
  "budget": "3000000",
  "experience": "Advanced",
  "locationType": "Onsite",
  "frontend": [
    "Angular"
  ],
  "backend": [
    "C#/.NET"
  ],
  "database": [
    "SQL"
  ],
  "cloud": [
    "Azure"
  ],
  "devops": [
    "Kubernetes"
  ],
  "design": [
    "Adobe XD"
  ],
  "otherTech": "Cutting edge AI technologies."
}
```


Figure A.1: Input format sent from a structured form

```
# Project: MediConnect

## Requirements
MediConnect is a healthcare management platform aimed at optimizing patient–doctor
interactions. The system will offer:
- Online appointment scheduling
- Electronic health record (EHR) storage
- Prescription tracking
- Automated reminders for patients and doctors
- Integration with third-party health APIs (e.g., wearable devices for vitals tracking)
- Compliance with GDPR and HIPAA regulations for data privacy and security
- Accessibility via both mobile and web platforms

## Technologies
- the frontend is developed with Angular
- the backend is developed with C#/.NET
- the database is implemented with SQL
- the cloud infrastructure is hosted on Azure
- the devops pipeline is managed with Kubernetes
- the design prototypes are created using Adobe XD
- the platform leverages cutting edge AI technologies

## Notes
- stakeholders include Hospital Admin, Doctors, Patients, IT Team, and Investors
- the project is categorized as External
- the start date is July 1, 2025
- the expected duration is 12 months
- the team size is 150 members
- the allocated budget is $3,000,000
- the development team has advanced experience
- the work location is Onsite
```

Figure A.2: Input format sent from an unstructured file

Project Plan – MediConnect Unified Healthcare Platform

1. Executive Summary & Project Charter

Background & Justification

Hospitals are under pressure to improve patient experience, reduce administrative overhead, and comply with strict data-privacy regulations. Current legacy scheduling and EHR systems are siloed, creating manual hand-offs that impact care quality. MediConnect will be a cloud-native, web + mobile platform that unifies appointment scheduling, EHR access, and secure messaging for hospital administrators, doctors, and patients.

Vision – "A friction-free digital bridge between healthcare providers and patients."

Mission – Deliver a secure, omni-channel application that enables real-time collaboration, reduces no-show rates by 25 %, and shortens record-retrieval time to <10 s.

Business Case

- €2.1 M annual savings from lower admin FTEs and reduced paper processes.
- +12 % revenue uplift via higher appointment utilization.
- Compliance cost avoidance (GDPR/HIPAA fines) estimated €3 M over 5 yrs.

Stakeholder Matrix

Stakeholder	Role	Interest	Power	Approval Authority
Hospital Board	Sponsor	ROI, compliance	High	Final budget & go-live sign-off
CIO & IT Dept.	Owner	Architecture, supportability	High	Technical go/no-go
Chief Medical Officer	Key User	Workflow fit	Medium	UAT sign-off
Doctors & Nurses	Users	UX, reliability	Medium	Advisory
Patients Council	Users	Accessibility	Low	Advisory
Data Protection Officer	Regulator	GDPR/HIPAA	High	Compliance sign-off

Scope Boundaries

In-Scope: Appointment booking, EHR viewer, secure messaging, wearable API ingestion, real-time notifications, training, 3-month hyper-care.

Out-of-Scope: Insurance billing integration, AI diagnosis engine, legacy system decommissioning.

Success Criteria & KPIs

- ≥98 % uptime in first 6 months.
- ≤5 % appointment no-show rate within 12 months.
- Mean Time to Resolution (MTTR) <2 hrs post-launch.
- 100 % GDPR/HIPAA compliance audit pass.
- Net Promoter Score (patient) ≥60.

2. Business Goals and Objectives

- **Strategic Goals** – Digitalize 80 % of patient interactions by 2025; cut admin costs 20 %.
- **Technical Objectives** – Microservice architecture on AWS; ≤250 ms P95 API latency; horizontal scalability to

Figure A.3: Sample Project Plan generated by the application (Executive Summary & Project Charter Section)

3. Work Breakdown Structure (WBS) & Effort Estimate

WBS	Task	Effort (wd)	Role(s)	Notes
1.0	Project Management & Scrum	60	PM, PO	Sprint planning, reviews
2.0	UX/UI Design	60	UX Designer	Wireframes, Hi-fi, Design System
3.0	Front-end – Web (Angular)	**180**	2 FE Devs	Component library, booking, EHR views
3.1	Angular setup & CI hooks	15	FE Dev, DevOps	
3.2	Design system implementation	20	FE Dev, UX	
3.3	Booking & calendar module	45	FE Devs	High effort ▲
3.4	EHR module	50	FE Devs	High effort ▲
3.5	Profile & settings	25	FE Dev	
3.6	Unit tests & Storybook	25	FE Dev, QA	
4.0	Front-end – Mobile (Flutter)	**180**	2 Flutter Devs	Mirrored features + native push
5.0	Backend API (Node.js, GraphQL)	**220**	3 BE Devs	CRUD, GraphQL resolvers, middleware
6.0	Database & Realtime (MongoDB/Firebase)	60	BE Dev, DevOps	Schema, indices, Firebase topics
7.0	Authentication & Security layer	40	BE Dev, Security	OAuth 2.0, MFA, JWT, RBAC
8.0	Third-party API Integrations	50	BE Devs, FE Devs	Fitbit, Apple HealthKit, Google Fit
9.0	Compliance Implementation & Audit	40	Security, PM	Policies, DPIA, audit evidence
10.0	DevOps & CI/CD (AWS)	70	DevOps, BE Dev	IaC, pipelines, monitoring
11.0	QA – Automation & Performance	80	QA Eng, DevOps	Cypress, JMeter, Load test 10 k
12.0	UAT & Bug Fix Hardening	40	All Devs, QA, Pilot users	
13.0	Launch & Hypercare	20	DevOps, QA, PM	Blue/green, smoke tests
14.0	Post-Launch Retrospective	5	All leads	Lessons learned

Total Effort ≈ **1 105 working-days** (development & delivery) + 60 wd PM + contingency built in timeline.
High-effort tasks marked ▲ ; resource constraints likely on BE (220 wd) and FE synchronisation.

Figure A.4: Sample Project Plan generated by the application (Work Breakdown Structure (WBS) & Effort Estimate Section)

Appendix

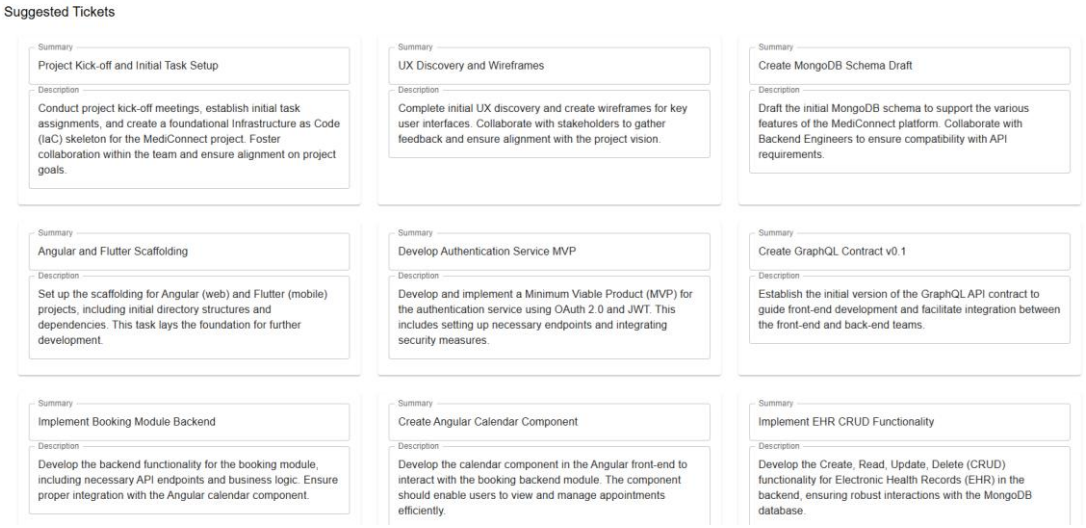

Figure A.5: Jira Ticket suggestions based on the project plan

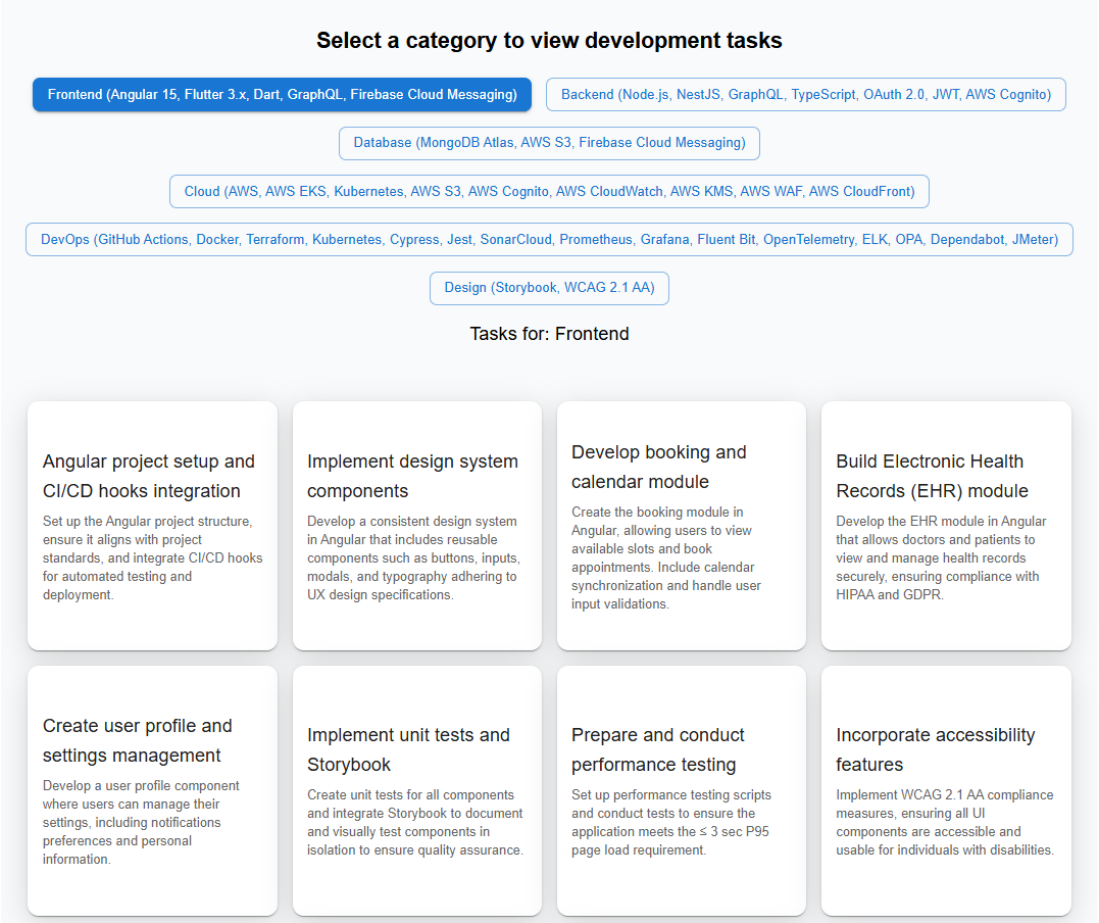

Figure A.6: Tasks generated for Frontend category

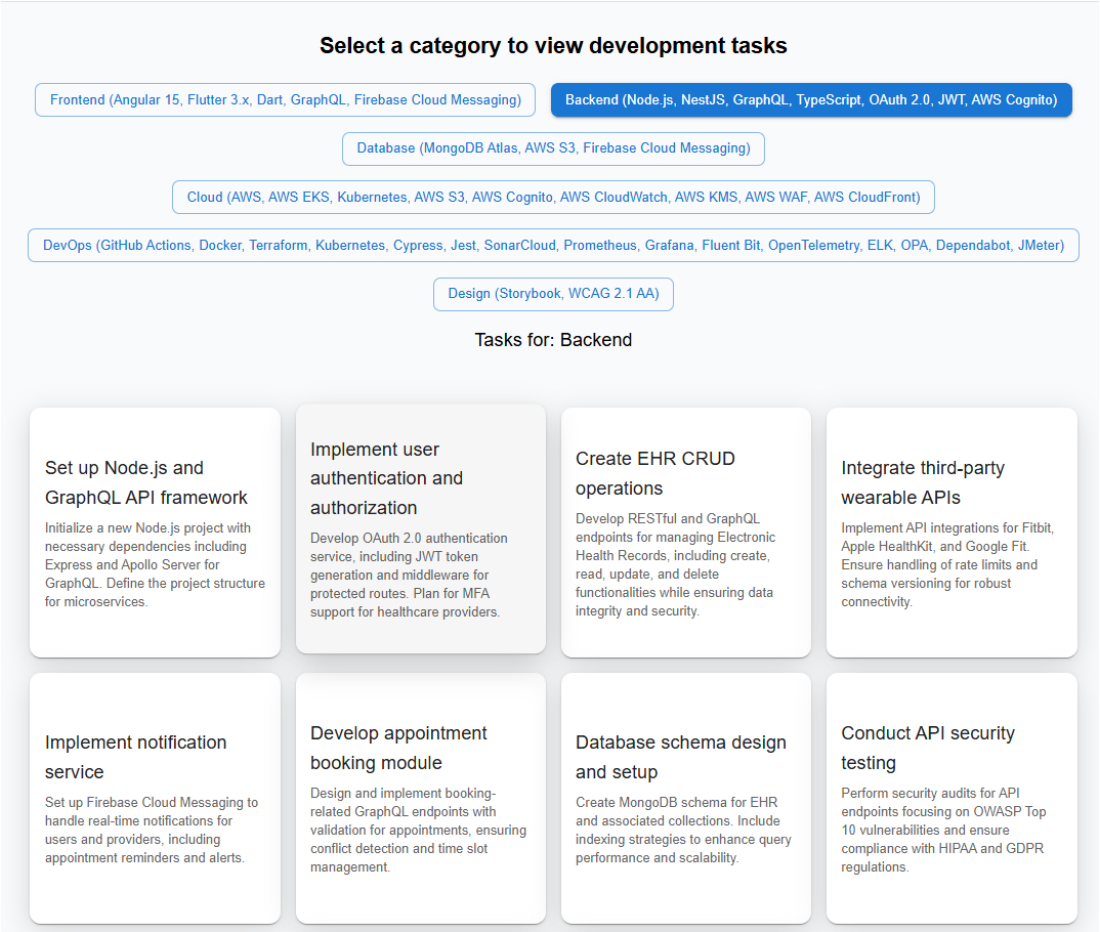

Figure A.7: Tasks generated for Backend category

Generated Code Snippet

```
// ─────────────────────────────────────────────
// libs/ui/src/lib/design-tokens.ts
// Centralised design-tokens (can be imported by FE & exported to JSON for Flutter)
// ─────────────────────────────────────────────
export const Colors = {
  primary500: '#0066FF',
  primary600: '#0052CC',
  secondary500: '#00C8A6',
  secondary600: '#009E84',
  neutral100: '#FFFFFF',
  neutral900: '#1A1A1A',
  danger500: '#E74C3C'
} as const;

export const Spacing = {
  0: '0px',
  1: '4px',
  2: '8px',
  3: '12px',
  4: '16px',
  5: '24px',
  6: '32px'
} as const;

export const Typography = {
  fontFamilyBase: `'Inter', sans-serif`,
  fontSizeSm: '0.875rem',
  fontSizeMd: '1rem',
  fontSizeLg: '1.25rem',
  fontWeightRegular: 400,
  fontWeightBold: 700
} as const;

/* Mount tokens to :root as CSS custom properties (runtime theme switch) */
export function mountTokens() {
  const root = document.documentElement;
  const record: Record<string, string | number> = { ...Colors, ...Spacing, ...Typography };
  Object.entries(record).forEach(([k, v]) => {
    root.style.setProperty(`--mc-${toKebab(k)}`, String(v));
  });
}
function toKebab(str: string) {
  return str.replace(/[A-Z]/g, m => `-${m.toLowerCase()}`);
```

Figure A.8: Code snippet example for one of the frontend tasks

```
// index-setup.js – run once per deployment (e.g. in CI/CD step)
import { MongoClient } from 'mongodb';

(async () => {
  const uri = process.env.MONGODB_URI || 'mongodb://localhost:27017/mediconnect';
  const client = new MongoClient(uri, { useNewUrlParser: true, useUnifiedTopology: true });

  try {
    await client.connect();
    const db = client.db();

    /* ====================
       PATIENTS COLLECTION
       ==================== */
    await db.collection('patients').createIndexes([
      // 1️⃣ Equality look-ups by patientId (UUID) → hashed index
      { key: { patientId: 'hashed' }, name: 'idx_patientId_hashed' },

      // 2️⃣ Unique constraint for login / contact purposes
      { key: { email: 1 }, name: 'uniq_email', unique: true },

      // 3️⃣ Compound index for alphabetical search (lastName → firstName)
      { key: { lastName: 1, firstName: 1 }, name: 'idx_name_compound' }
    ]);

    /* ====================
       APPOINTMENTS COLLECTION
       ==================== */
    await db.collection('appointments').createIndexes([
      // 1️⃣ Core filter: patientId + date range queries (desc to optimise latest-first sort)
      { key: { patientId: 1, appointmentDate: -1 }, name: 'idx_patient_date' },

      // 2️⃣ Doctor schedule queries
      { key: { doctorId: 1, appointmentDate: -1 }, name: 'idx_doctor_date' },

      // 3️⃣ Partial index for quick access to future, cancellable slots only
      {
        key: { status: 1, appointmentDate: 1 },
        name: 'idx_upcoming',
        partialFilterExpression: { status: { $in: ['BOOKED', 'RESCHEDULED'] } }
      }
    ]);

    /* ====================
       AUDIT LOGS COLLECTION (TTL)
       ==================== */
    await db.collection('auditLogs').createIndex(
      { createdAt: 1 },
      {
        name: 'ttl_audit_1y',
        expireAfterSeconds: 60 * 60 * 24 * 365 // 1 year retention (HIPAA min requirement met via backups)
      }
    );

    console.log('✅ MongoDB indexes ensured');
  } catch (err) {
    console.error('❌ Index creation failed', err);
    process.exit(1);
  } finally {
    await client.close();
  }
})();
```

Figure A.9: Code snippet example for one of the database tasks