Table of Contents

[1. Introduction 4](#_Toc205072767)

[1.1 Objectives 5](#_Toc205072768)

[1.2 Research Design 5](#_Toc205072769)

[1.2.1 Research Questions 5](#_Toc205072770)

[1.3 Structure of the Thesis 7](#_Toc205072771)

[2. Literature Review 9](#_Toc205072772)

[2.1 Software Development Lifecycle 9](#_Toc205072773)

[2.1.1 Definition and importance of SDLC 9](#_Toc205072774)

[2.1.2 Common SDLC models 9](#_Toc205072775)

[2.1.3 Overview of its stages: planning, design, development, testing, deployment, maintenance. 9](#_Toc205072776)

[2.2 Planning and Development Phases in SDLC 9](#_Toc205072777)

[Role of project planning: requirements gathering, scope definition, task estimation. 9](#_Toc205072778)

[Role of development: coding, integration, and initial testing. 9](#_Toc205072779)

[Challenges in these phases that could benefit from automation. 9](#_Toc205072780)

[2.3 Large Language Models (LLMs) 9](#_Toc205072781)

[2.3.1 Definition and architecture (transformers, attention mechanisms). 9](#_Toc205072782)

[2.3.2 Capabilities (text generation, summarization, reasoning, etc.). 9](#_Toc205072783)

[2.3.3 Limitations (context length, factual accuracy, hallucination). 9](#_Toc205072784)

[2.4 Leading LLMs in the Market 10](#_Toc205072785)

[2.4.1 Overview of current state-of-the-art models 10](#_Toc205072786)

[2.5 Agentic Tools and AI Agents 10](#_Toc205072787)

[2.5.1 Definition of agentic tools and agent-based systems 10](#_Toc205072788)

[2.5.2 The idea of "tool use" in LLMs 10](#_Toc205072789)

[2.5.3 Why tool-use makes LLMs more interactive, context-aware, and capable 10](#_Toc205072790)

[2.6 Popular Agentic Frameworks 10](#_Toc205072791)

[2.6.1 LangChain 10](#_Toc205072792)

[2.6.2 Auto-GPT 10](#_Toc205072793)

[2.6.3 OpenAI Function Calling 10](#_Toc205072794)

[2.6.4 Microsoft AutoGen 10](#_Toc205072795)

[2.6.5 Comparisons and use cases 10](#_Toc205072796)

[2.7 Reasoning Models and Chain-of-Thought Techniques 10](#_Toc205072797)

[2.7.1 Introduction to reasoning models and prompting techniques. 10](#_Toc205072798)

[2.7.2 Chain-of-thought, self-reflection, ReAct (Reason + Act). 10](#_Toc205072799)

[2.7.3 Role in decomposing tasks, planning, and decision-making. 10](#_Toc205072800)

[3. Implementation 11](#_Toc205072801)

[3.1 System Overview 11](#_Toc205072802)

[3.1.1 High-Level System Description 11](#_Toc205072803)

[3.1.2 User Interaction Flow 11](#_Toc205072804)

[3.1.3 Core System Components 11](#_Toc205072805)

[3.2 System Architecture 12](#_Toc205072806)

[3.2.1 High-Level Architecture Overview 12](#_Toc205072807)

[3.2.2 Data Flow and Component Interaction 13](#_Toc205072808)

[3.2.3 Architecture Diagram 14](#_Toc205072809)

[3.3 Technology Stack 14](#_Toc205072810)

[3.3.1 Frontend 15](#_Toc205072811)

[3.3.2 Backend 15](#_Toc205072812)

[3.3.3 Language Model Integration 15](#_Toc205072813)

[3.3.4 Agent Orchestration 16](#_Toc205072814)

[3.3.5 Utilities and Supporting Libraries 16](#_Toc205072815)

[3.3.6 Development Tools and External References 16](#_Toc205072816)

[3.4 Module 1: Project Planning 16](#_Toc205072817)

[3.4.1 Overview 16](#_Toc205072818)

[3.4.2 Input Methods 17](#_Toc205072819)

[3.4.3 Data Normalization 22](#_Toc205072820)

[3.4.4 Agentic Workflow (CrewAI Orchestration) 24](#_Toc205072821)

[3.4.5 LLM Prompting & Generation 27](#_Toc205072822)

[3.4.6 Plan Refinement and Ticketing Features 29](#_Toc205072823)

[3.4.7 Output Format 30](#_Toc205072824)

[3.4.8 Sample Input/Output 31](#_Toc205072825)

[3.5 Module 2: Code Generation 31](#_Toc205072826)

[3.6 Integration Flow 31](#_Toc205072827)

[3.7 Challenges & Solutions 31](#_Toc205072828)

[3.8 Summary 31](#_Toc205072829)

[4. Evaluation 31](#_Toc205072830)

[5. Conclusion and Outlook 31](#_Toc205072831)

[6. Future Scope 31](#_Toc205072832)

# 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 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 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 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 are accelerated by advancements in reasoning models, agentic tools, deep search etc., Models such as OpenAI, Google’s Gemini, DeepSeek demonstrate various techniques such as chain-of-thought reasoning, multi agent collaboration and many more which highlight the growth of AI in 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 together, they can make powerful systems that can manage end to end workflows with very minimum human intervention.

Despite all these advancements AI, there still remains a gap using these improvements into creating 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 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 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 on how the system can be evaluated effectively using different metrics.

## 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 software development lifecycle which producing high quality and actionable outputs. In this thesis we are mainly focusing 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 agent tools like web search, code execution, and file analysis.
* 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.

## 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.

### 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 agentic tools like web search and code execution improve the ability of reasoning models in aiding automated software development?

* + 1. 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, 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.

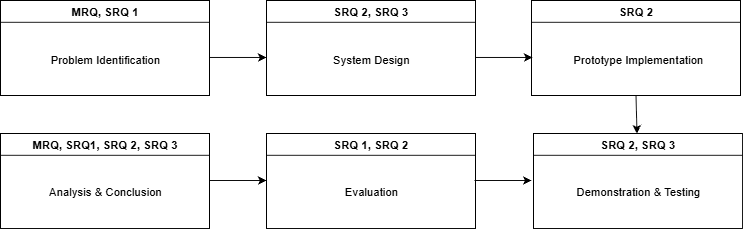
* + 1. Research Methodology

By combining the research questions with the design approach, we developed a research methodology to guide the creation and assessment of the proposed system. 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**

## 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 system design.

**Chapter 3: Implementation**. This chapter describes the architecture, components, and technologies used to develop the prototype system. It includes the integration of reasoning models and agentic tools.

**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 and Outlook**. This chapter summarizes key findings, reflects on limitations, and discusses potential future improvements.

**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**, and **List of Abbreviations** are included at the end of the document for clarity and completeness.

# Literature Review

In this chapter, we conduct a background study to establish the basic concepts and technologies relevant to this research. We start by discussing the software development lifecycle (SDLC), focusing on the planning and development phases. Next, we provide an overview of Large Language Models (LLMs), covering their architecture, capabilities, and the leading implementations in the AI market. The chapter also introduces the concept of agentic tools, which are systems that enhance the functionality of LLMs by enabling them to interact with external environments. We explore the prominent frameworks used to create agent-based systems. Finally, we look at recent advancements in reasoning models and AI agents, as well as the gaps in current literature that drive the goals of this thesis.

## Software Development Lifecycle

### Definition and importance of SDLC

The Software Development Life Cycle (SDLC) is a structured approach that guides the creation, implementation, and improvement of software systems. It originated in engineering and was developed in response to the mid-20th century software crisis, characterized by many project failures, budget overruns averaging 45%, and schedule delays exceeding 7% in large IT projects (Antonios Saravanos and Ma/hew X. Curinga 2 2023). At its heart, SDLC presents what (Lekh and Pooja 2015) describe as a step-by-step method for problem-solving through five interconnected 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 lower risks. Data from the Standish Group (2015-2020) shows that projects using SDLC methods have a 31% success rate, while those using ad-hoc methods have a 19% failure rate. This success comes from setting quality checkpoints that spot unclear requirements before implementation (Antonios Saravanos and Ma/hew X. Curinga 2 2023).
* Second, it improves resource use. Simulations by Bassil demonstrate that poorly assigned staff create major bottlenecks. Shortages of programmers can lead to a 247% increase in implementation wait times for large projects (Antonios Saravanos and Ma/hew X. Curinga 2 2023).
* Third, it ensures quality. By integrating with CMMI's five-tiered maturity model, organizations can enhance their processes in 22 areas, including Requirements Management (REQM) and Quantitative Project Management (QPM). Achieving Level 4 compliance cuts defect leaks by 42% (Lekh and Pooja 2015).
* Fourth, it fosters resilience. Modern versions of SDLC apply web engineering principles like stateless RESTful APIs and containerized deployment to address changing scalability needs. This allows e-commerce sites to handle over 10,000 concurrent users without sacrificing service quality (Chahar and Singh 2024).

This framework transforms software development from chaotic improvisation into a measurable engineering practice. As Humphrey and Kellner (1988) note, processes are essential for successful software development.

### Common SDLC models

**Waterfall Model**

The Waterfall model represents a linear approach to software development. It follows a sequence of phases: Requirements, Design, Implementation, Testing, and Maintenance. The model originated from Royce's 1970 work, which, while not using the term "waterfall," introduced the idea of a "downward flow" depicted in Figure 3 of his influential paper (Antonios Saravanos and Ma/hew X. Curinga 2 2023). This model is effective in situations with fixed requirements, such as safety-critical avionics systems. Here, detailed documentation is crucial for auditing and compliance (Sommerville, 2011). However, findings from (Antonios Saravanos and Ma/hew X. Curinga 2 2023) discrete-event simulations highlight significant weaknesses. 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. Design phases also lagged, reaching 18.1-time units without optimization. These issues prompted the development of hybrid models like "Scrumbanfall", which combines Waterfall's emphasis on documentation with Kanban's approach to workflow visualization. Recent PMI data (2020) shows that Waterfall remains important. About 56% of government and healthcare projects continue to use it for regulatory purposes, although only 28% meet their deadlines, compared to 65% for Agile hybrids (Antonios Saravanos and Ma/hew X. Curinga 2 2023).

**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 (Chahar and Singh 2024). Research by (yas et al. 2023) 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 (Mohammad Ikbal Hossain 2023). 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% (Chahar and Singh 2024).

**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 (Agarwal et al. 2023). 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 (Lekh and Pooja 2015). 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 (Chahar and Singh 2024).

**DevOps Model**

DevOps brings together development and operations through automated tools that support Continuous Integration (CI) and Continuous Deployment (CD). This approach is crucial for web applications that need daily updates, such as SaaS platforms. Jenkins pipelines automate regression testing, which cuts validation cycles by 40%. Docker containers also help create standardized environments across development, staging, and production (Chahar and Singh 2024). Kubernetes manages rolling updates with blue-green deployments, reducing downtime by 68% compared to traditional release methods. The integration of security (DevSecOps) incorporates vulnerability scanning into CI/CD processes, identifying SQL injection threats 53% quicker than manual reviews. Despite its benefits, the complexity of these tools can hinder onboarding. Teams may take 12 to 18 weeks to become proficient with Kubernetes, which delays the return on investment (Chahar and Singh 2024).

**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 (Agarwal et al. 2023). 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 (Chahar and Singh 2024).

### Overview of its stages: planning, design, development, testing, deployment, maintenance.

**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 (Lekh and Pooja 2015). Web engineering uses Balsamiq wireframes to prototype user interface flows and MoSCoW prioritization (Must/Should/Could/Won't) to manage changes in scope (Chahar and Singh 2024). Impact analysis matrices help track requirement changes against configurable items, which prevents 27% of downstream defects (Lekh and Pooja 2015). Simulations by (Antonios Saravanos and Ma/hew X. Curinga 2 2023) 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 ER models, outlining server clusters, API gateways, and database sharding strategies for scalability (Lekh and Pooja 2015). 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 (Lekh and Pooja 2015). Web engineering requires RESTful API contracts with OpenAPI specifications, not stateful sessions via JWT tokens, and responsive designs using CSS Flexbox/Grid (Chahar and Singh 2024). Simulation data indicates 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 (Antonios Saravanos and Ma/hew X. Curinga 2 2023).

**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 (Lekh and Pooja 2015). 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% (Chahar and Singh 2024). This phase tends to be the most prone to bottlenecks. Resource modeling by (Antonios Saravanos and Ma/hew X. Curinga 2 2023) 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; NPM library vulnerabilities account for 31% of security breaches in JavaScript applications (Chahar and Singh 2024)

**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 (Lekh and Pooja 2015). Web engineering expands on this with specific checks. Cross-browser testing (BrowserStack) ensures consistency across Chrome, Firefox, and Safari. OWASP ZAP penetration tests find SQL injection and cross-site scripting vulnerabilities. Lighthouse audits check performance budgets (Chahar and Singh 2024). 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 (Antonios Saravanos and Ma/hew X. Curinga 2 2023). 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) (Mohammad Ikbal Hossain 2023).

**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 (Chahar and Singh 2024). 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) (Antonios Saravanos and Ma/hew X. Curinga 2 2023). 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 (Lekh and Pooja 2015). For long-term web applications, "recycle phases" restart development cycles to change monolithic architectures into microservices, extending system viability by 4–7 years (Lekh and Pooja 2015).

**Critical Analysis and Emerging Trends**

The SDLC landscape faces growing pressure from AI disruptions. Machine Learning models now predict requirement changes with 89% accuracy using past project data. Automated code generators like GitHub Copilot cut development efforts by 35% (Antonios Saravanos and Ma/hew X. Curinga 2 2023). Quantum computing brings significant changes, requiring SDLC adjustments for qubit-based algorithms. Sustainability needs drive "Green SDLC" practices. Energy-efficient coding lowers AWS carbon footprints by 18%, while serverless architectures reduce wasted resources (Chahar and Singh 2024). These innovations coexist with ongoing issues, including hybrid model governance, technical debt management, and DevSecOps integration. These challenges highlight that SDLC is evolving as a dynamic discipline rather than remaining a fixed methodology.

## Planning and Development Phases in SDLC

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

Project planning lays the groundwork for software development. It involves gathering requirements, defining the scope, and estimating tasks. The research highlights different methods and challenges:  
**Requirements Gathering and Scope Definition**:  
The Waterfall Model (Antonios Saravanos and Ma/hew X. Curinga 2 2023) emphasizes careful upfront requirements analysis. This approach produces a formal Software Requirements Specification (SRS) document, which aims to finalize requirements before proceeding. This model assumes that requirements will change rarely, but it can lead to problems if the initial analysis is inadequate.  
Web Engineering principles (Chahar and Singh 2024) note the difficulties in adjusting requirements for web applications due to fast development cycles and changing security needs. While they acknowledge that Waterfall is suitable for well-defined projects, they advocate for using iterative models like Agile, Spiral, or Prototype. These models are better equipped to handle shifting user needs and feedback during planning. Techniques like wireframing and creating Documents of Understanding (DoU) with screens are recommended as useful practices in requirement analysis (Lekh and Pooja 2015)  
The CDP Model (Lekh and Pooja 2015) breaks down sub-practices in requirement gathering. It lists "Wire-framing," "Document of Understanding (with screens)," and "Scope of Work (SOW)" as Method 1, 2, and 3 for the "Requirement Gathering or Analysis (SRS Preparation)" phase. It also highlights SRS Review through "Intra Team Review" and "Inter Team Review."  
  
**Task Estimation and Resource Planning**:  
A significant finding from the Waterfall simulation study (Antonios Saravanos and Ma/hew X. Curinga 2 2023) 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 (Lekh and Pooja 2015) 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 (Chahar and Singh 2024).  
  
**Frameworks**: CMMI (Capability Maturity Model Integration) provides a systematic approach with defined process areas, such as Requirements Management (REQM), Project Planning (PP), and Project Monitoring and Control (PMC). These assist in improving planning maturity and establishing best practices like PPBs and PPMs (Lekh and Pooja 2015).  
Tools include discrete-event simulation (SimPy) for resource estimation and predicting bottlenecks (Antonios Saravanos and Ma/hew X. Curinga 2 2023), statistical analysis tools like Minitab for normality testing, control charting, ANOVA, and developing PPBs/PPMs, and MS Excel for recording data and calculating metrics (Lekh and Pooja 2015).

**Comparison**: The Waterfall paper (Antonios Saravanos and Ma/hew X. Curinga 2 2023) presents a quantitative, simulation-based method for resource estimation in a rigid planning phase. The Web Engineering paper (Chahar and Singh 2024) contrasts this with the need for flexibility in web projects and supports iterative models. The CDP paper (Lekh and Pooja 2015) 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.

### 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 (Antonios Saravanos and Ma/hew X. Curinga 2 2023).

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 (Chahar and Singh 2024).

The CDP Model (Lekh and Pooja 2015) 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 (Antonios Saravanos and Ma/hew X. Curinga 2 2023; Mithilesh Pandey et al. 2022)

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 (Chahar and Singh 2024).

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 (Chahar and Singh 2024).

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 (Lekh and Pooja 2015).

**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 (Chahar and Singh 2024).

Web-specific principles, such as REST, statelessness, and asynchronous communication, represent a specialization within development practices (Chahar and Singh 2024).

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 (Lekh and Pooja 2015).

### 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.

#### Challenges Identified

* **Resource Bottlenecks and Idle Time**: The Waterfall simulation example (Antonios Saravanos and Ma/hew X. Curinga 2 2023) 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 (Chahar and Singh 2024). 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 (Lekh and Pooja 2015) and the requirement for simulation in Waterfall (Antonios Saravanos and Ma/hew X. Curinga 2 2023) reflect this.
* **Delays in Defect Detection**: The Waterfall model delays rigorous testing, which will be more expensive to fix later (Antonios Saravanos and Ma/hew X. Curinga 2 2023; Mithilesh Pandey et al. 2022).
* **Lack of Efficient Testing Processes**: Timely test case preparation, testing, and test case audit is cumbersome (Lekh and Pooja 2015).
* **Challenges in Integration**: Manual integration in sequential models can be complex and prone to errors (Mithilesh Pandey et al. 2022).
* **Maintaining Consistency and Standards**: Maintaining coding standards and design specifications is difficult to ensure manually.

#### Automation Strategies and Technologies

* **Simulation for Planning Resources**: The Waterfall paper by (Antonios Saravanos and Ma/hew X. Curinga 2 2023) 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 viewed as a risk-free and efficient means of experimenting and planning.
* **CI/CD Pipelines**: DevOps and Web Development practices (Chahar and Singh 2024) support automating the deployment, integration, unit testing and integration tests—and build processes. Jenkins style tools do this automation, providing rapid feedback and low-friction integration and deployment.
* **Automated Testing**: Although not specifically outlined in the papers, the focus on testing issues and CI/CD environment suggests the use of automated unit test packages such as JUnit and pytest, and potentially automated functional and regression testing tools to preclude inefficiencies with manual testing.
* **Automated Code Review and Analysis**: Static code analysis tools can automatically review coding standards, security vulnerabilities, and potential bugs and assist with the said manual code reviews in the CDP model (Lekh and Pooja 2015).
* **Requirements Management Software**: Specialist software may be able to automate requirement tracing, impact analysis, and change control, though not mentioned. This would help overcome the volatility issues (Lekh and Pooja 2015; Chahar and Singh 2024).

#### Case Study Example:

Our primary case study is from the Waterfall simulation paper (Antonios Saravanos and Ma/hew X. Curinga 2 2023). 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 whole-heartedly 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.

## Large Language Models (LLMs)

### 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. (Johnson and Hyland-Wood 2024)

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. (Raiaan et al. 2024)

LLMs are based on the Transformer architecture, which uses self-attention mechanisms to process sequences in parallel, replacing sequential RNNs/CNNs (R Sahana Lokesh 2025). Key components include positional embeddings (e.g., rotary embeddings) and layer normalization for stable training (R Sahana Lokesh 2025).

### 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. (Johnson and Hyland-Wood 2024)

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. (Raiaan et al. 2024)

LLMs enable text generation (e.g., essays, code), summarization, and translation across 46 languages (e.g., BLOOM) (R Sahana Lokesh 2025). 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 (Tamkin et al. 2021).

### 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. (Johnson and Hyland-Wood 2024)

They sometimes make up believable but false facts—what researchers call hallucination. Their memory is limited too, struggling with super long documents or chats. Plus, they inherit biases from training data, need heavy computing power, and trip up on words with multiple meanings. (Raiaan et al. 2024).

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 (Tamkin et al. 2021). They also exhibit hallucinations (inaccurate outputs) and scalability challenges (R Sahana Lokesh 2025).

## Leading LLMs in the Market

### OpenAI

### DeepSeek

### Qwen

### Grok

### Llama

### Claude

### Mistral

### Gemini

### Command R

## **Agentic Tools and AI Agents**

### Definition of agentic tools and agent-based systems

### The idea of "tool use" in LLMs

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

## Popular Agentic Frameworks

### LangChain

### Auto-GPT

### OpenAI Function Calling

### Microsoft AutoGen

### Comparisons and use cases

## Reasoning Models and Chain-of-Thought Techniques

### Introduction to reasoning models and prompting techniques.

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

### Role in decomposing tasks, planning, and decision-making.

# Implementation

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

## System Overview

### 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 deliver structured project plan
* Allows refinement of the plan
* Converts plans into actionable JIRA tickets
* Generates modular code snippets based on tickets or user-requirement

### 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
* User can refine the generated plan multiple times until they are satisfied
* User can use “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
* User selects a ticket or specifies requirement for code generation
* The code generated can be refined or enhanced using simple prompts from the user

### 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 GPT-4o, and an agent orchestration engine based on CrewAI. These components work together to allow for dynamic task handling, agent coordination, and interactive output generation.

## 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 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 structed 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.

### High-Level Architecture Overview

At high level, the system consists of the following omponents:

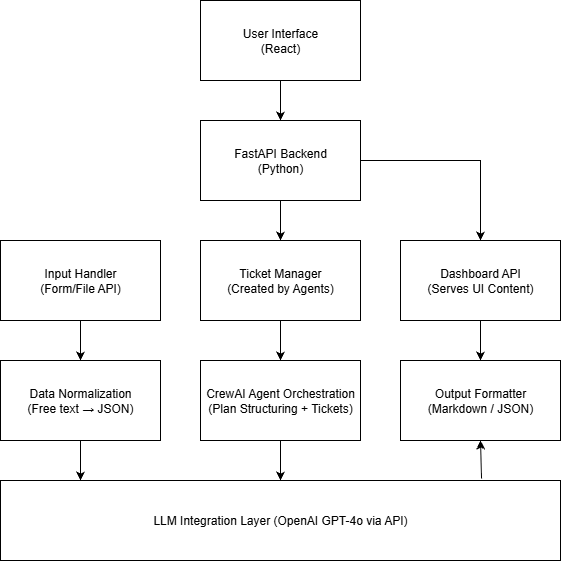
* **Frontend Interface (React):** Provides a modern user interface for entering requirements, viewing the project plan, refining the plan, generating tickets and interacting with generated code.
* **Backend Server (FastAPI - Python):** Acts as a main controlling layer for the system. It handles inputs from the frontend, manages API routing, triggers agent workflows and supports interactions with the LLM.
* **LLM Integration Layer (OpenAI GPT-4o):** Handles all the language model interactions such as project planning, refinement of the plans based of user feedback and code generation using OpenAI GPT-4o reasoning model. This layer is responsible for sending the requirements given by the agent modules to the OpenAI API and present them to the user. GPT-4o is chosen for its reasoning capability and low-latency performance.
* **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.

### 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 are 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 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 to 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.
* **Code Refinement:** The user can ask the system to modify the code logic using simple prompts.

### Architecture Diagram



**Figure 3.1 High Level Architecture Diagram**

## 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.

### 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. It has a complete ecosystem with tools such as router, redux and 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 by its ready to use elements and the built-in responsiveness in them.

### 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 is having 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.

It’s capability of handling requests asynchronously makes it a perfect choice for managing long-running tasks like LLM queries and agent orchestration.

**Uvicorn**: An ASGI server used to run the FastAPI application, offering support for asynchronous, non-blocking I/O.

### Language Model Integration

**OpenAI GPT-4o:** This is the primary LLM used in the system. It’s reasoning capabilities, tool-use potential and low-latency performance makes it the most well-suited model to use for this thesis. It’s main role in this thesis is to power project generation and plan refinement.

### 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. In this thesis a few tasks assigned to the agents are Project Intake Analyst, Business Object Mapper, Risk Identifier etc.

### 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.

### Development Tools and External References

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

## Module 1: Project Planning

### Overview

The project planning module server as the most foundational component of the system. The primary purpose of this module is to generate a detailed and well structed 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 it’s support to iterative interaction. Users can refine the generated project plan by submitting feedback about the changes or 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, the 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.

### Input Methods

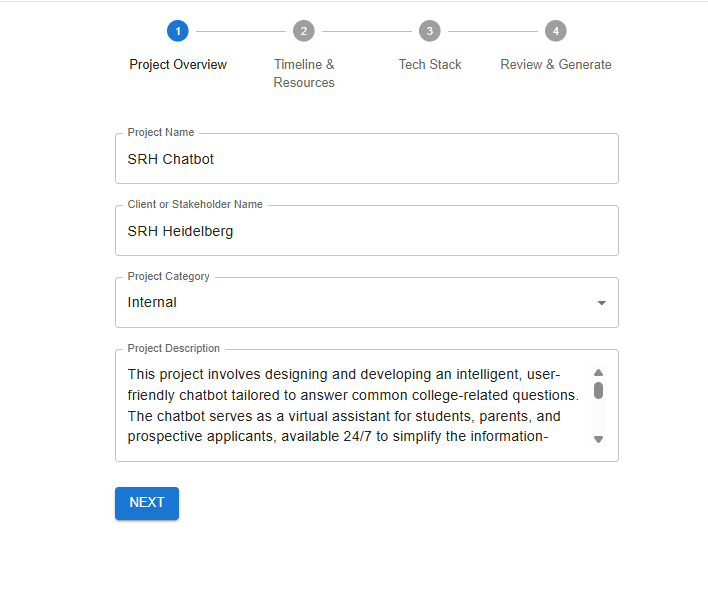
This module accepts two primary methods for collecting input from the users: **form-based structured** **input** and **file-based unstructured input** enabling flexibility to the users to define the project requirements using either the interactive UI or by uploading a specification document.

#### 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:**

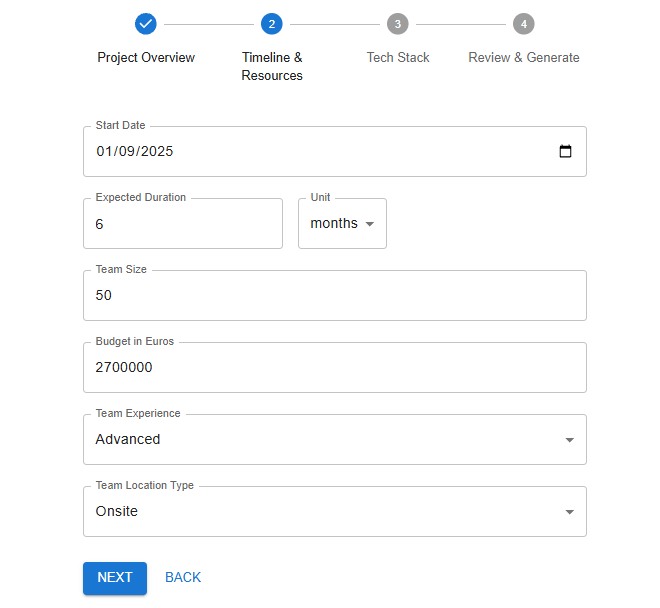
* **Project Name (\*):** As the name suggests it is a plain text field where the user must enter a name for their Project. This is a mandatory field.
* **Client or Stakeholder Name:** This field expects the name of the client or stakeholder the project is being developed for. It is a plain text field. 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: Project Overview Section**

**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 user’s expectation about when the project must be completed. This is a plain text which accepts only numbers with a dropdown indicating options such a 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.

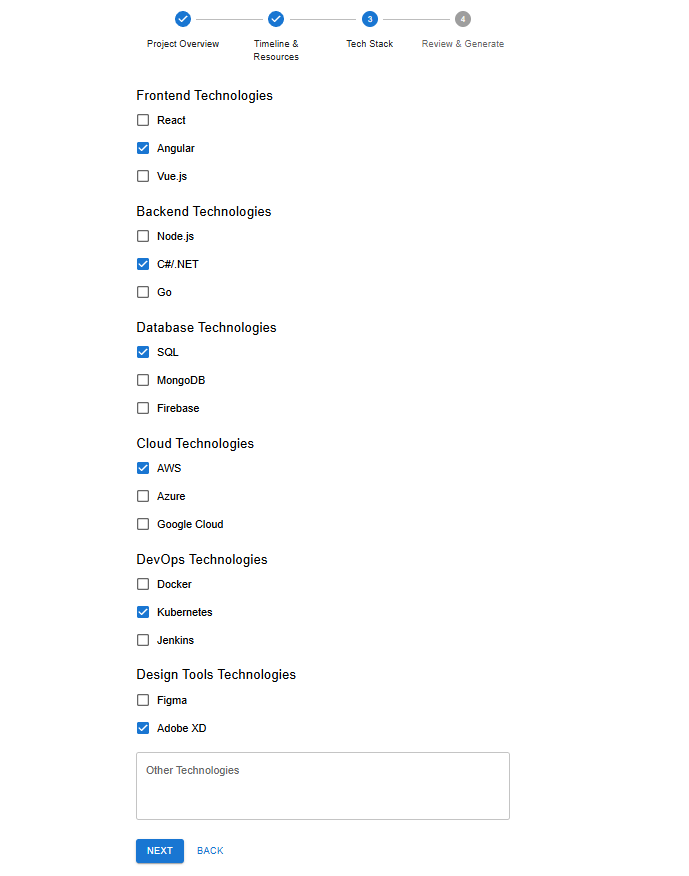
****

**Figure 3.3: Timeline & Resources Section**

**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 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: Technology Stack Section**

#### File-based Unstructured Input

As an alternate to the pre-defined form, the 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 expectation or natural language description of the requirements they have for their desired project. Once uploaded and submitted the following process is carried 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 a form. Also note that the current system does not take images into consideration. Only files through which text can be extracted are parsed.

#### Input Validation and Schema Enforcement

All inputs irrespective or 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 returned to the frontend so that any normal user might understand what went wrong and how to fix it.

### Data Normalization

The data normalization is one of the most important stages which is responsible to 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 the further module work with a consistent and semantically rich representation of the user’s requirements, regardless of the original input source.

#### Input Sources

User inputs are acquired using 2 channels in the frontend and submitted to separate backend routes:

* **Structured Form Input:**

Data is collected through a multiple form UI form and is mapped directly to a FastAPI ***ProjectInput*** model. These fields include project name, duration, tech stack selection and stakeholder information.

* **Unstructured File Input**

The text content is extracted from the files uploaded by the user in the form of ***.txt, .dox*** or ***.pdf*** files on the frontend and then submitted to the backend as a raw text payload (e.g., {“text”: “…”}), preserving the natural language format written by the user.

#### Normalization Process

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

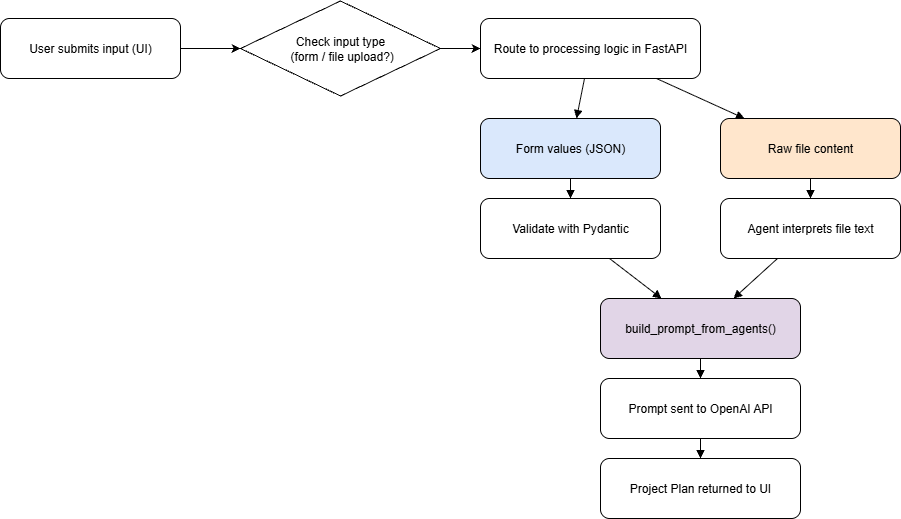
* **Structured Input Pathway (Form** → ***/api/generate-project-plan*):** 
  + Data arrives as a validated ***ProjectInput*** *Pydantic model*
  + This data is fed into ***build\_crew()*** a function used to construct a task-oriented CrewAI pipeline with relevant agents.
  + The outputs generated by the agents is combined into a high-level prompt using ***build\_prompt\_from\_agents()****.*
* **Unstructured Input Pathway (File → text input, not yet formalized in backend):** 
  + Currently the raw text is treated as it is and inserted into a generalized planning prompt.
  + No formal schema extraction occurs yet in the backend of the system.
  + Normalization is left to the internal capabilities of the LLM’s reasoning models guided by a prompt structure.
* **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 though 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 which 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: Input Normalization Flow**

### 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 it well aligned with the user’s expectation.

#### *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 together 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 ticket to suggest for the user.

#### *Agentic Workflow: Project Planning Phase*

In the project planning phase:

* The system constructs a CrewAI instance via *build\_crew(data)* using the structured *ProjectInput* coming from the frontend.
* The agents defined in the crew are assigned different roles such as:
  + Map business goals
  + Flag project risks
  + Suggest optimal architecture
  + Break project into sprints
  + Inject modern best practices
* These agents operate in a sequential mode using *crew.kickoff(),* 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 the OpenAI’s GPT model to generate a project plan.

#### *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.

#### *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.

#### *Context and Memory Sharing*

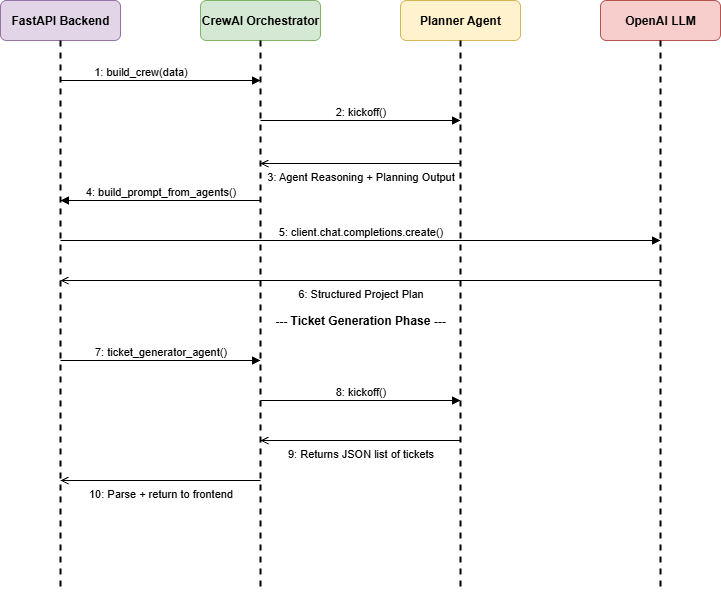
CrewAI internally manages context propagation between agents. This allows:

* Each agent 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.

#### *Orchestration Logic in Backend*

The initiation of the orchestration takes place without 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**

### 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 GPT-4o 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 it’s reasoning capabilities.

#### 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.

#### 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, risks assessment, WBS, architecture recommendation, etc.,
* The response of LLM is parsed and displayed in the frontend dashboard

#### Model Selection and Performance

* OpenAI’s GPT-4o was selected as the main model because of it’s 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 GPT-4o selectively. It only applies to tasks that need open-ended language generation or adjustment.

### Plan Refinement and Ticketing Features

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

#### 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 add 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 GPT-4o 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 GPT-4o is directly responsible it.

#### 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 to use that plan and create 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 them 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.

### Output Format

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

#### 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

#### Development Ticket Output

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

### Sample Input/Output

This section provides example of inputs that are accepted and outputs generated by the system during the planning module. These examples include how the system handles both structured and unstructured inputs and also shows in what form the project plan and the tickets are generated.

#### Sample: Structured Form Input (Excerpt)

(To be added)

#### Sample: File Upload Input (Excerpt)

(To be added)

#### Sample Project Plan Output (Excerpt)

(To be added)

#### Sample Ticket Generated Output (Excerpt)

(To be added)

## Module 2: Code Generation

## Integration Flow

## Challenges & Solutions

## Summary

# Evaluation

# Conclusion and Outlook

# Future Scope

Publication bibliography

Agarwal, Archi; Agarwal, Avni; Verma, Deepak Kumar; Tiwari, Deepti; Pandey, Rashmi (2023): A Review on Software Development Life Cycle. In *IJSRCSEIT*, Article CSEIT2390387, pp. 384–388. DOI: 10.32628/CSEIT2390387.

Antonios Saravanos; Ma/hew X. Curinga 2 (2023): 2308.03940v3.

Chahar, Suman; Singh, Satyaveer (2024): Analysis of SDLC Models with Web Engineering Principles. In : 2024 2nd International Conference on Advancements and Key Challenges in Green Energy and Computing (AKGEC). 2024 2nd International Conference on Advancements and Key Challenges in Green Energy and Computing (AKGEC). Ghaziabad, India, 21/11/2024 - 23/11/2024: IEEE, pp. 1–7.

Johnson, Sandra; Hyland-Wood, David (2024): A Primer on Large Language Models and their Limitations. Available online at http://arxiv.org/pdf/2412.04503v1.

Lekh, Rachna; Pooja (2015): 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. 2015 International Conference on Advances in Computer Engineering and Applications (ICACEA). Ghaziabad, India, 19/03/2015 - 20/03/2015: IEEE, pp. 997–1003.

Mithilesh Pandey; Preeti Kuniyal; Govind Kumar (2022): A Comprehensive Study on Software Development Lifecycle and Systematic Analysis of It Challenges.

Mohammad Ikbal Hossain (2023): Software Development Life Cycle (SDLC) Methodologies for Information Systems Project Management. In *IJFMR* 5 (5), Article 6223. DOI: 10.36948/ijfmr.2023.v05i05.6223.

R Sahana Lokesh (2025): Open-Source Large Language Models: A Comprehensive Survey. In : 2025 International Conference on Machine Learning and Autonomous Systems (ICMLAS). 2025 International Conference on Machine Learning and Autonomous Systems (ICMLAS). Prawet, Thailand, 10/03/2025 - 12/03/2025: IEEE, pp. 1096–1101.

Raiaan, Mohaimenul Azam Khan; Mukta, Md. Saddam Hossain; Fatema, Kaniz; Fahad, Nur Mohammad; Sakib, Sadman; Mim, Most Marufatul Jannat et al. (2024): A Review on Large Language Models: Architectures, Applications, Taxonomies, Open Issues and Challenges. In *IEEE Access* 12, pp. 26839–26874. DOI: 10.1109/ACCESS.2024.3365742.

Tamkin, Alex; Brundage, Miles; Clark, Jack; Ganguli, Deep (2021): Understanding the Capabilities, Limitations, and Societal Impact of Large Language Models. Available online at http://arxiv.org/pdf/2102.02503v1.

yas, qahtan; Alazzawi, Abdulbasit; Rahmatullah, Bahbibi (2023): A Comprehensive Review of Software Development Life Cycle methodologies: Pros, Cons, and Future Directions. In *ijcsm*, pp. 173–190. DOI: 10.52866/ijcsm.2023.04.04.014.