Table of Contents

[1. Introduction 5](#_Toc206336787)

[1.1 Objectives 6](#_Toc206336788)

[1.2 Research Design 6](#_Toc206336789)

[1.2.1 Research Questions 6](#_Toc206336790)

[1.3 Structure of the Thesis 8](#_Toc206336791)

[2. Literature Review 10](#_Toc206336792)

[2.1 Software Development Lifecycle 10](#_Toc206336793)

[2.1.1 Definition and importance of SDLC 10](#_Toc206336794)

[2.1.2 Common SDLC models 11](#_Toc206336795)

[2.1.3 Overview of its stages: planning, design, development, testing, deployment, maintenance. 14](#_Toc206336796)

[2.2 Planning and Development Phases in SDLC 16](#_Toc206336797)

[2.2.1 Role of project planning: requirements gathering, scope definition, task estimation. 16](#_Toc206336798)

[2.2.2 Role of development: coding, integration, and initial testing. 18](#_Toc206336799)

[2.2.3 Challenges in these phases that could benefit from automation. 20](#_Toc206336800)

[2.3 Large Language Models (LLMs) 22](#_Toc206336801)

[2.3.1 Definition and architecture (transformers, attention mechanisms) 22](#_Toc206336802)

[2.3.2 Capabilities (text generation, summarization, reasoning, etc.) 23](#_Toc206336803)

[2.3.3 Limitations (context length, factual accuracy, hallucination) 23](#_Toc206336804)

[2.4 Leading LLMs in the Market 24](#_Toc206336805)

[2.4.1 OpenAI 24](#_Toc206336806)

[2.4.2 DeepSeek 24](#_Toc206336807)

[2.4.3 Claude 24](#_Toc206336808)

[2.4.4 Gemini 25](#_Toc206336809)

[2.5 Agentic Tools and AI Agents 25](#_Toc206336810)

[2.5.1 Definition of agentic tools and agent-based systems 25](#_Toc206336811)

[2.5.2 The idea of "tool use" in LLMs 25](#_Toc206336812)

[2.5.3 Why tool-use makes LLMs more interactive, context-aware, and capable 26](#_Toc206336813)

[2.6 Popular Agentic Frameworks 26](#_Toc206336814)

[2.6.1 CrewAI 26](#_Toc206336815)

[2.6.2 Auto-GPT 26](#_Toc206336816)

[2.6.3 IBM Watson 26](#_Toc206336817)

[2.6.4 Microsoft AutoGen 27](#_Toc206336818)

[2.6.5 Comparisons and use cases 27](#_Toc206336819)

[2.7 Reasoning Models and Chain-of-Thought Techniques 27](#_Toc206336820)

[2.7.1 Introduction to reasoning models and prompting techniques. 27](#_Toc206336821)

[2.7.2 Chain-of-thought, self-reflection, ReAct (Reason + Act). 27](#_Toc206336822)

[2.7.3 Role in decomposing tasks, planning, and decision-making. 27](#_Toc206336823)

[3. Implementation 28](#_Toc206336824)

[3.1 System Overview 28](#_Toc206336825)

[3.1.1 High-Level System Description 28](#_Toc206336826)

[3.1.2 User Interaction Flow 28](#_Toc206336827)

[3.1.3 Core System Components 28](#_Toc206336828)

[3.2 System Architecture 29](#_Toc206336829)

[3.2.1 High-Level Architecture Overview 29](#_Toc206336830)

[3.2.2 Data Flow and Component Interaction 30](#_Toc206336831)

[3.2.3 Architecture Diagram 31](#_Toc206336832)

[3.3 Technology Stack 31](#_Toc206336833)

[3.3.1 Frontend 32](#_Toc206336834)

[3.3.2 Backend 32](#_Toc206336835)

[3.3.3 Language Model Integration 32](#_Toc206336836)

[3.3.4 Agent Orchestration 33](#_Toc206336837)

[3.3.5 Utilities and Supporting Libraries 33](#_Toc206336838)

[3.3.6 Development Tools and External References 33](#_Toc206336839)

[3.4 Module 1: Project Planning 33](#_Toc206336840)

[3.4.1 Overview 33](#_Toc206336841)

[3.4.2 Input Methods 34](#_Toc206336842)

[3.4.3 Data Normalization 39](#_Toc206336843)

[3.4.4 Agentic Workflow (CrewAI Orchestration) 41](#_Toc206336844)

[3.4.5 LLM Prompting & Generation 44](#_Toc206336845)

[3.4.6 Plan Refinement and Ticketing Features 46](#_Toc206336846)

[3.4.7 Output Format 47](#_Toc206336847)

[3.4.8 Sample Input/Output 48](#_Toc206336848)

[3.5 Module 2: Code Generation 50](#_Toc206336849)

[3.5.1 Overview 50](#_Toc206336850)

[3.5.2 Task Extraction & Categorization (Agent Layer) 51](#_Toc206336851)

[3.5.3 Code Prompting & Generation (LLM Layer) 53](#_Toc206336852)

[3.5.4 Code Generation Workflow 56](#_Toc206336853)

[3.5.5 Output Format 58](#_Toc206336854)

[3.5.6 Sample Input/Output 58](#_Toc206336855)

[3.6 Integration Flow 58](#_Toc206336856)

[3.7 Challenges & Solutions 58](#_Toc206336857)

[3.8 Summary 58](#_Toc206336858)

[4. Evaluation 58](#_Toc206336859)

[5. Conclusion and Outlook 58](#_Toc206336860)

[6. Future Scope 58](#_Toc206336861)

[Publication bibliography 58](#_Toc206336862)

# 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.

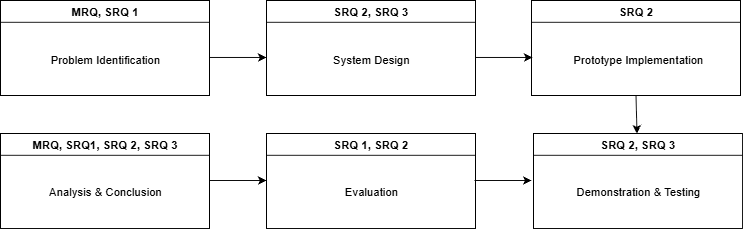
* + 1. Research Methodology

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 [1]. At its heart, SDLC presents what [2] 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 [1].
* 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 [1].
* 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% [2].
* 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 [3].

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 [1]. 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 [1] 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 [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].

**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 [3]. 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 [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].

### 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 [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 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 JWT tokens, and responsive designs using CSS Flexbox/Grid [3]. 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 [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; 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. OWASP ZAP penetration tests find 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 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% [1]. 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 [3]. 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 [1] 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 [3] 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 [2]  
The CDP Model [2] 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 [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) 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 [2].  
Tools include discrete-event simulation (SimPy) for resource estimation and predicting bottlenecks [1], statistical analysis tools like Minitab for normality testing, control charting, ANOVA, and developing PPBs/PPMs, and MS Excel for recording data and calculating metrics [2].

**Comparison**: The Waterfall paper [1] presents a quantitative, simulation-based method for resource estimation in a rigid planning phase. The Web Engineering paper [3] contrasts this with the need for flexibility in web projects and supports iterative models. 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.

### 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].

### 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 [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.

#### 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 viewed as a risk-free and efficient means of experimenting and planning.
* **CI/CD Pipelines**: DevOps and Web Development practices [3] 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 [2].
* **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 [2, 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 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. [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 RNNs/CNNs [10]. Key components include positional embeddings (e.g., rotary embeddings) and layer normalization for stable training [10].

### 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].

### 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—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. [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].

## Leading LLMs in the Market

### 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, o4, o4-mini, o5 (open AI Website)

### 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 (DeepSeek Website)

### 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 (Claude Website)

### 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 (Gemini Website)

## **Agentic Tools and AI Agents**

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

AI Agents are self-operating programs built to tackle specific tasks alone [13]. Agentic AI goes further it’s 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" [14]. They operate independently through "planning, learning, and environmental data to make decisions without necessarily requiring direct human intervention," enabling complex task execution. Hierarchical architectures organize interactions across multiple agent levels (e.g., master, orchestrator, micro-agents).

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

LLMs stay limited until they’re given tools like search APIs or code executors [13]. 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 [14]. Systems like LangChain and AutoGen enable "language understanding, reasoning, planning, and decision-making," optimizing workflows. This tool integration facilitates "complex workflow" creation and automation.

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

Tools let LLMs act checking real-time info, scheduling, or analyzing data [13]. 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) [14]. It improves context-awareness by combining "text, images, and sound" for richer interactions. Capability expands through "workflow optimization," including resource allocation and automation opportunities.

## Popular Agentic Frameworks

### 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. [14]

### 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. 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. [14]

### 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. [14]

### 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. 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. [14]

### Comparisons and use cases

|  |  |  |  |
| --- | --- | --- | --- |
| 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 QA, legal doc analysis, domain-specific insights |

**Table 2‑1: Comparison of different AI Agent Frameworks [14]**

## 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:

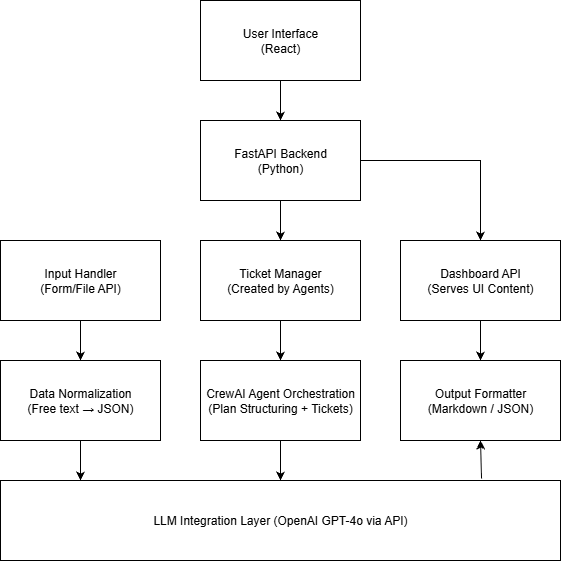
* **Frontend Interface (React):** Provides a modern user interface for entering requirements, viewing the project plan, refining the plan, generating tickets and interacting with generated code.
* **Backend Server (FastAPI - Python):** Acts as a main controlling layer for the system. It handles inputs 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

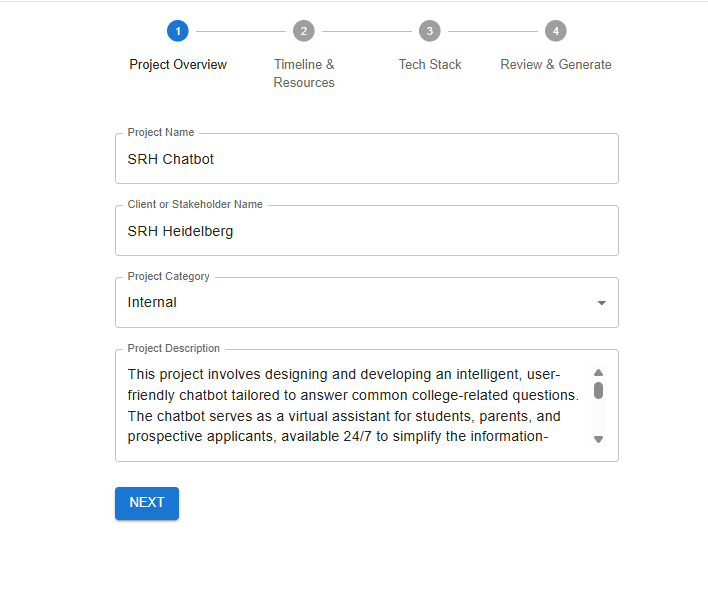
This module accepts two primary methods for collecting input from the users: **form-based structured** **input** and **file-based unstructured input** enabling flexibility to the users to define the project requirements using either the interactive UI or by uploading a specification document.

#### 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:**

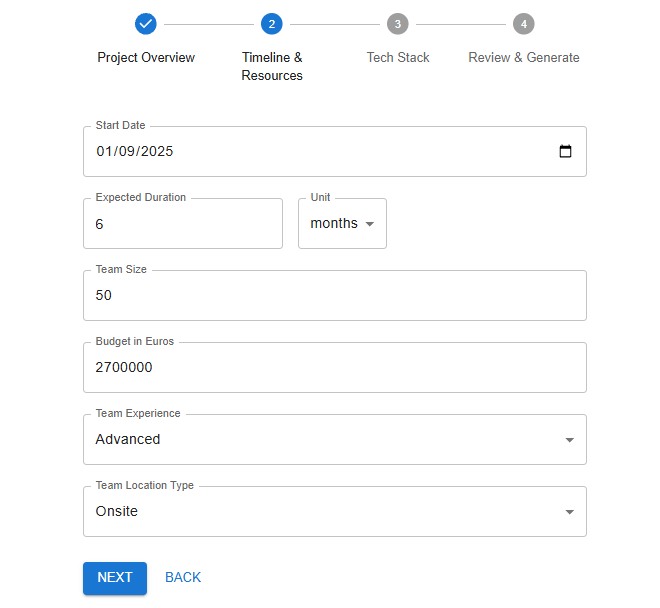
* **Project Name (\*):** As the name suggests it is a plain text field where the user must enter a name for their Project. This is a mandatory field.
* **Client or Stakeholder Name:** This field expects the name of the client or stakeholder the project is being developed for. It is a plain text field.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

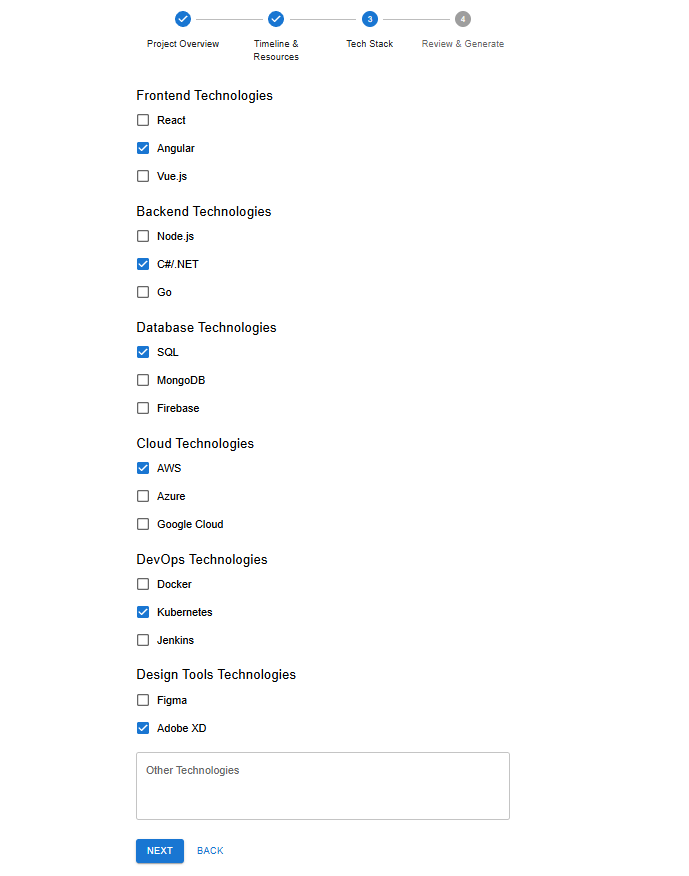
****

**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:

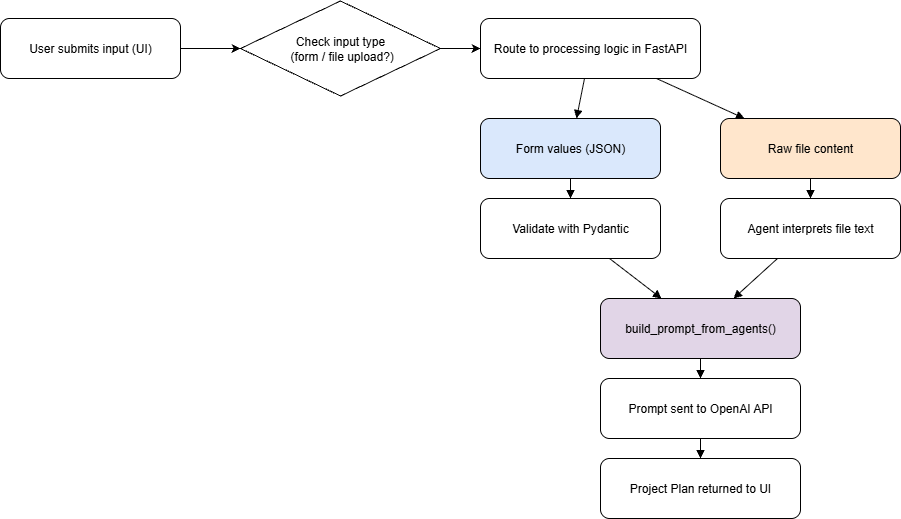
* **Structured Input Pathway (Form** → ***/api/generate-project-plan*):** 
  + Data arrives as a validated ***ProjectInput*** *Pydantic model*
  + This data is fed into ***build\_crew()*** a function used to construct a task-oriented CrewAI pipeline with relevant agents.
  + The outputs generated by 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*

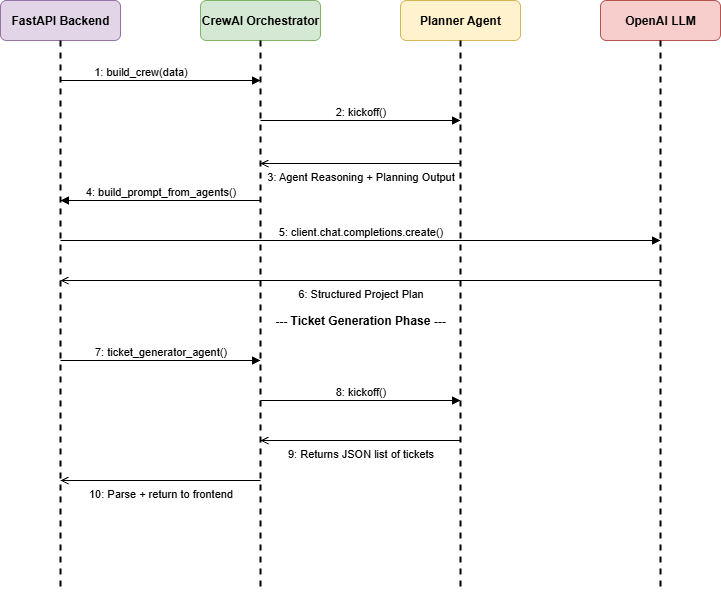
CrewAI internally manages context propagation between agents. This allows:

* 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

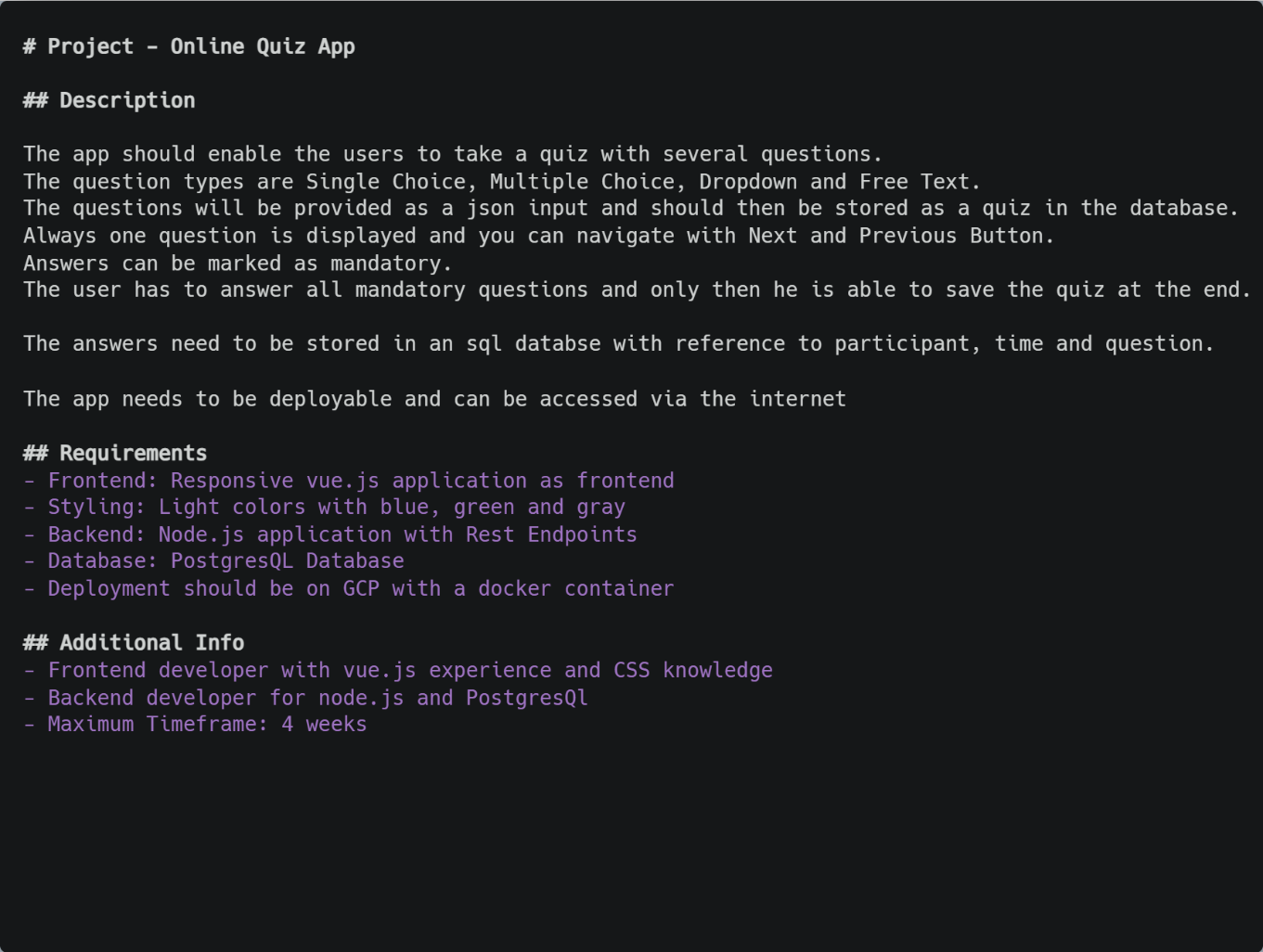
This section provides example of inputs that are accepted and outputs generated by the system during the planning module. These examples include how the system handles both structured and unstructured inputs and also shows in what form the project plan and the tickets are generated.

#### Sample: Structured Form Input



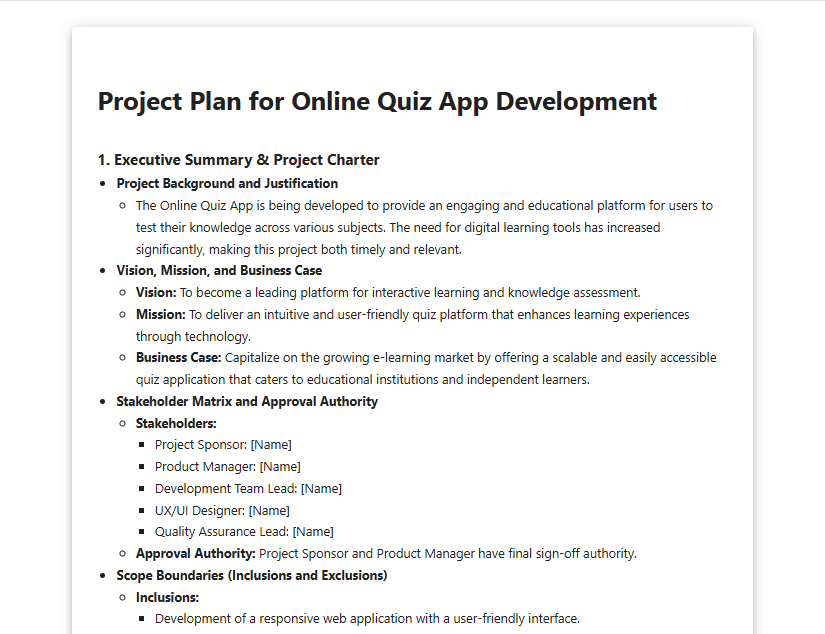
**Figure 3.7: Structured Form Input**

#### Sample: File Upload Input



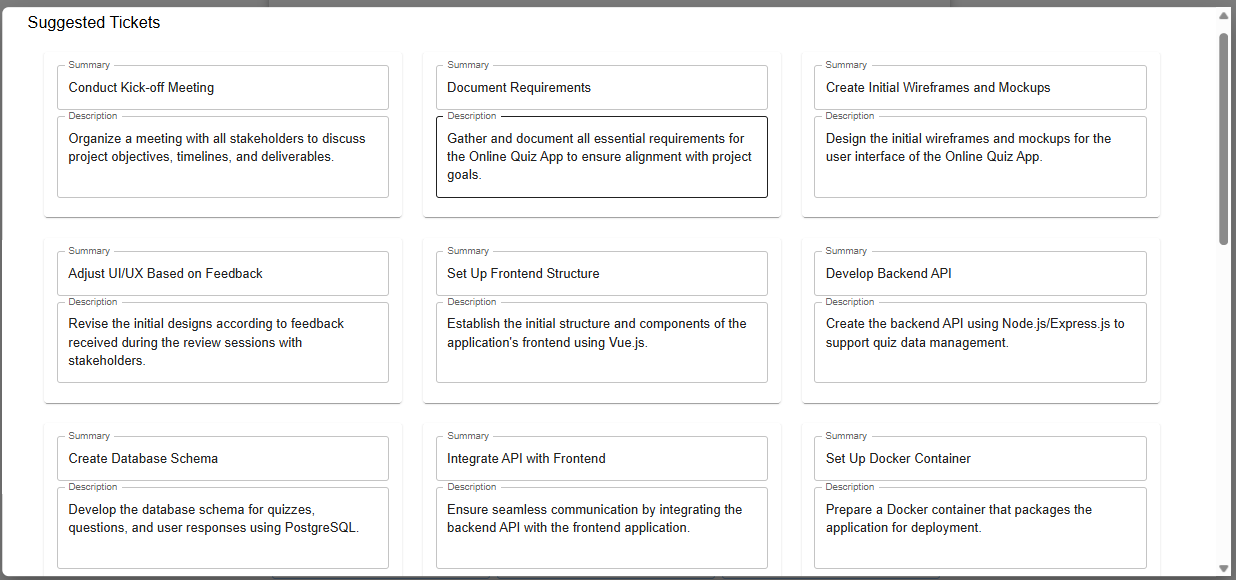
**Figure 3.8: File Upload Input**

#### Sample Project Plan Output (Excerpt)



**Figure 3.9: Project Plan Output (Excerpt)**

#### Sample Ticket Generated Output (Excerpt)



**Figure 3.10: Ticket Generated Output (Excerpt)**

## Module 2: Code Generation

### 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 to help 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 to generate 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 instruction for producing a code in a structured JSON format. The LLM then generates and returns code specific to the language that can be rendered in the dashboard.

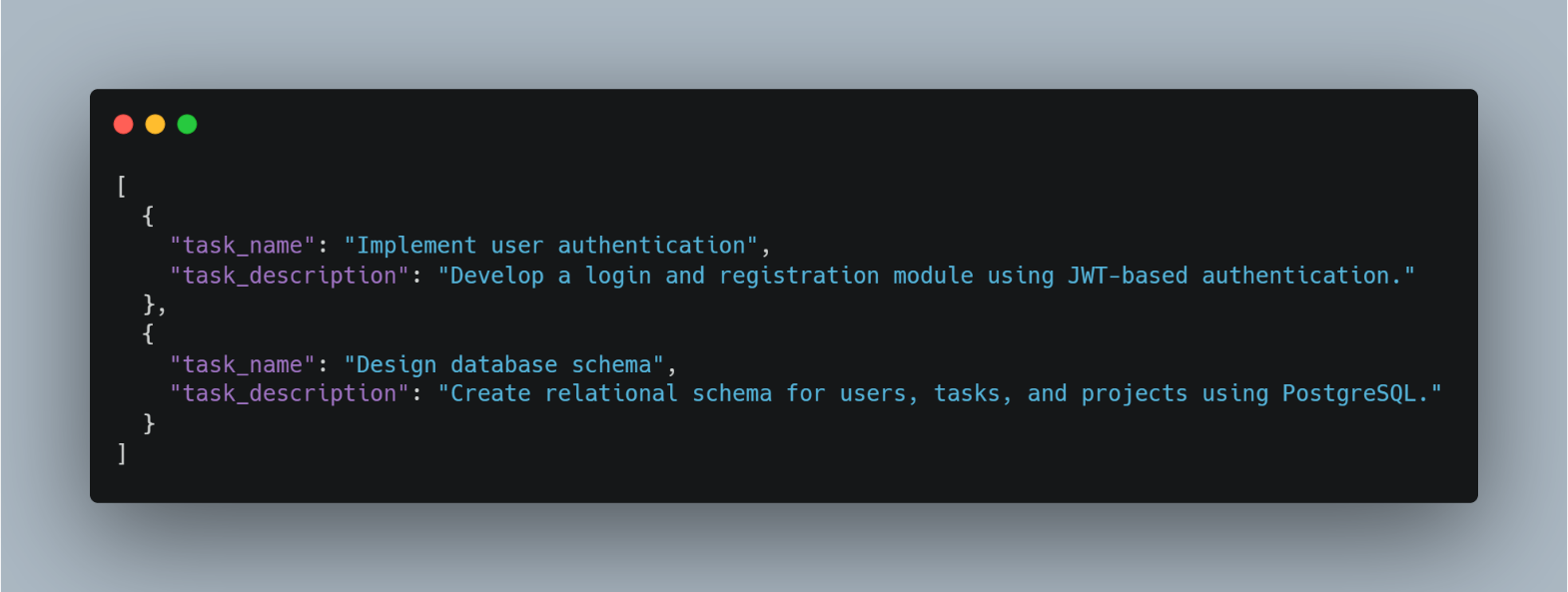
This combination of agentic task decomposition and LLM based code generation in this module ensures that the generated code snippets are not only relevant to the context but also aligned properly with the project plan. This represents a very important step towards bridging the gap between automated planning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

### 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.

#### Development Task Extraction

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

These extracted tasks are then structured in a JSON form to ensure compatibility with downstream processing. The below image represents a small sample of how the tasks are structured.

**Figure 3.11: Task Extraction Agent Output**

#### 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)

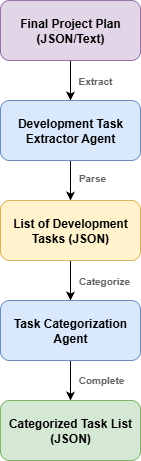
This step makes sure that the structured prompts for code generation are specialized, accurate and relevant. The below is a small sample on how the extracted tasks are categorized.



**Figure 3.12: Task Categorization Agent Output**

**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.



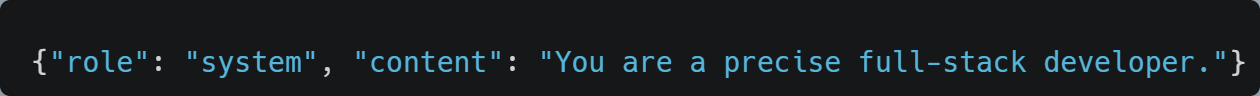
**Figure 3.13: Task Extraction and Categorization Pipeline**

### 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 which aligns with the architecture and technology stack of the project. The backend construct a prompt with two parts (system + user) and calls OpenAI Gpt-4o and instructs it to generate an output with a particular JSON schema so that the frontend can easily render the code.

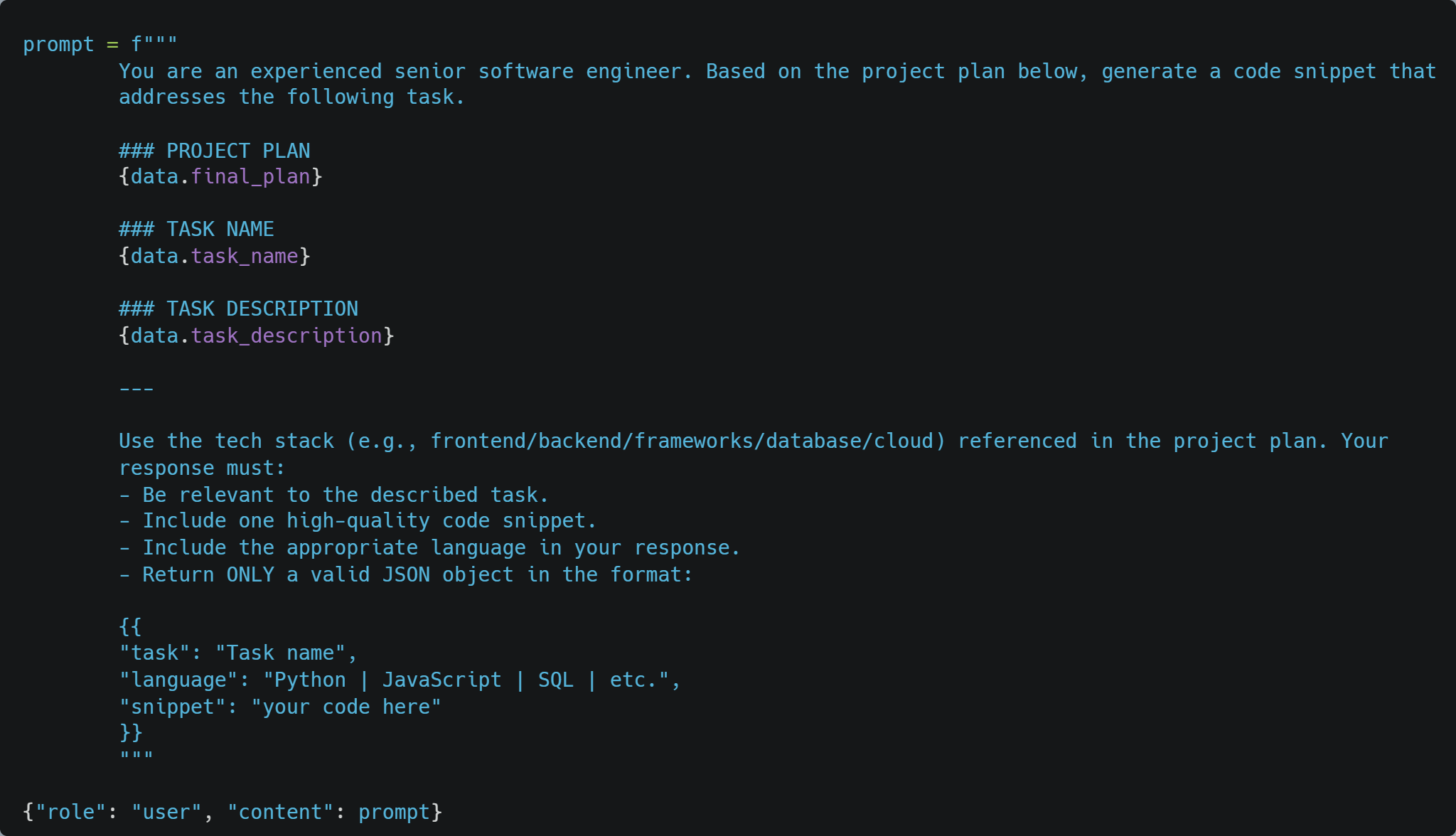
#### Prompt Design

* **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.



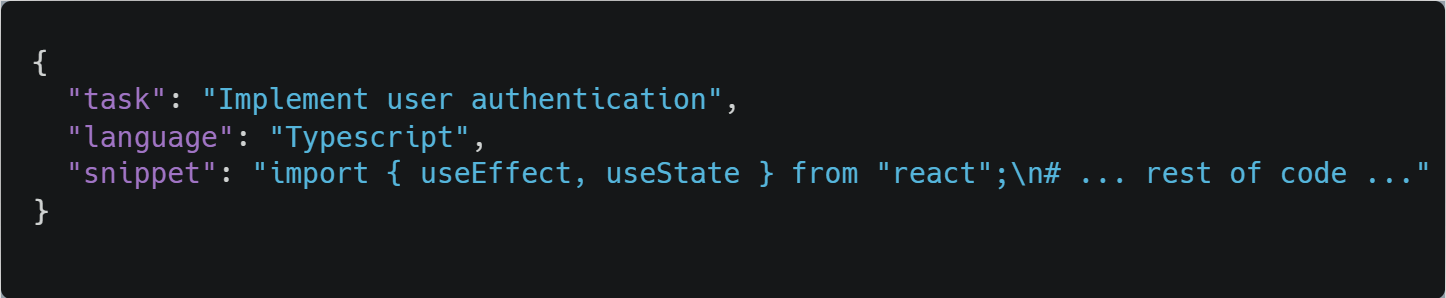
**Figure 3.14: System Role**

* **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 3.15: User Role**

* **Output Contract:** The model returns its response in a strict JSON format. The below figure shows a sample of the model response.

****

**Figure 3.16: Output contract**

#### Model & Parameters

The system uses **OpenAI-4o** 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 **low temperature** of **0.3** 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 ensure that the response is always aligned with project’s plan, task description and technology stack. This configuration minimizes hallucinations and makes sure that the code generated is trustworthy.

#### Context Narrowing

Hallucinations are often seen in the LLM responses due to passing of data that are irrelevant to the task that you need the LLM to do. Hence, in order 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.

#### Safety & Guardrails

The safety is insured 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 which 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 the LLM that it is a “precise full-stack developer”, which encourages focused and accurate output. No other security checks are applied apart from these prompt based constraints.

#### Error Handling & Retries

Once the LLM model sends the response the backed attempts to parse it directly as a 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 HTTP 500 and returns the details of the error to the client. Currently the implementation does not include any retries or repair prompts or schema validation.

### Code Generation Workflow

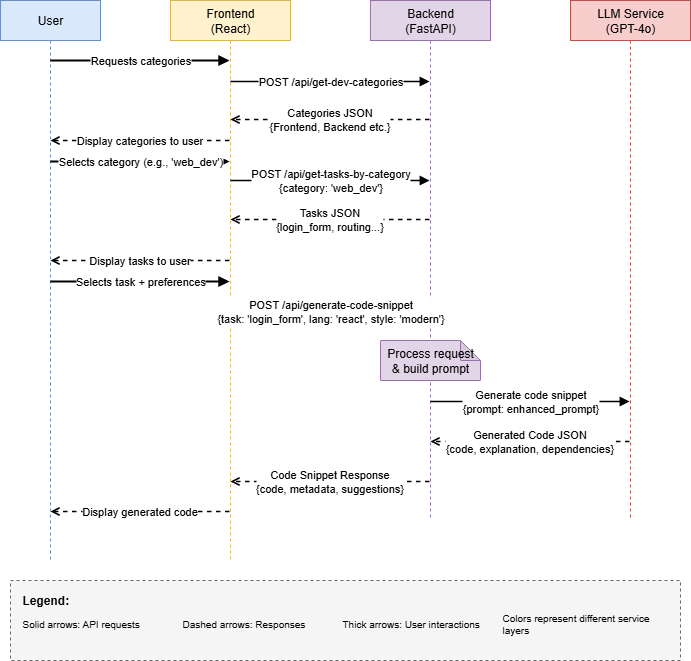
The code generation workflow in the development module is structured in such a way that it helps user to view the tasks in a very organized manner. The implementation is a sequence of API calls that narrow down the user selection from categories such a Frontend, Backend etc. to specific tasks related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

#### 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 return 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.

#### 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 GPT-4o endpoint.
* The response received is parsed as a JSON.
* If the parsing is successful the backend returns the JSON object else throws a HTTP 500 error.



**Figure 3.17: Code Generation Workflow - Sequence Diagram**

### Output Format

### Sample Input/Output

## Integration Flow

### Overview

The integration flow shows how the gap between the **Project Planning Module** and **Project Development Module** is bridged. Both modules serve distinct purposes. The planning module focuses on the 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 requirement specified at the beginning directly influence the code snippets produced at the end.

### 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.

#### 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.

#### 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.

### 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.

#### 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.

#### 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 GPT-4o.

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.

#### 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.

The following table give a quick overview of all APIs used in the systems one place. With the input it takes and output it provides.

|  |  |  |
| --- | --- | --- |
| Endpoint | Input | Output |
| /api/generate-plan | Form/File input (requirements) | 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) |

**Table 3‑1: API Summary Table**

### 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.

#### 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.

#### Transition to Development

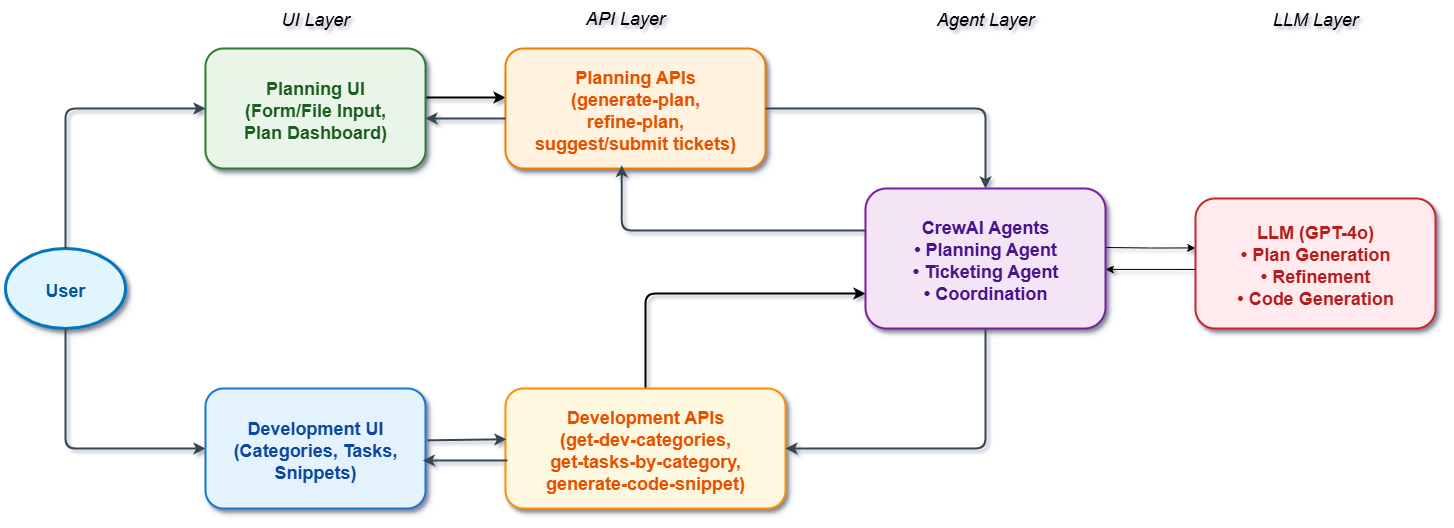
* After finishing the planning phase, the user moves to the Development screen.
* Upon entry, the frontend automatically calls the backend to retrieve categories from the stored plan using the POST ***/api/get-dev-categories*** endpoint.
* The categories are listed as selectable items, ensuring a direct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

#### 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.

### Detailed System Integration



**Figure 3.18: Detailed System Integration Flow**

## Challenges & Solutions

## Summary

# Evaluation

# Conclusion and Outlook

# Future Scope

References

1. Antonios Saravanos, Ma/hew X. Curinga 2. 2308.03940v3. 2023.

2. Lekh R, 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 (ICACEA); 19/03/2015 - 20/03/2015; Ghaziabad, India: IEEE; 2015. p. 997–1003. doi:10.1109/ICACEA.2015.7164852.

3. Chahar S, Singh S. Analysis of SDLC Models with Web Engineering Principles. In: 2024 2nd International Conference on Advancements and Key Challenges in Green Energy and Computing (AKGEC); 21/11/2024 - 23/11/2024; Ghaziabad, India: IEEE; 2024. p. 1–7. doi:10.1109/AKGEC62572.2024.10868694.

4. yas q, Alazzawi A, Rahmatullah B. A Comprehensive Review of Software Development Life Cycle methodologies: Pros, Cons, and Future Directions. ijcsm. 2023:173–90. doi:10.52866/ijcsm.2023.04.04.014.

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

6. Agarwal A, Agarwal A, Verma DK, Tiwari D, Pandey R. A Review on Software Development Life Cycle. IJSRCSEIT. 2023:384–8. doi:10.32628/CSEIT2390387.

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

8. Johnson S, Hyland-Wood D. A Primer on Large Language Models and their Limitations; 03/12/2024.

9. Raiaan MAK, Mukta MSH, Fatema K, Fahad NM, Sakib S, Mim MMJ, et al. A Review on Large Language Models: Architectures, Applications, Taxonomies, Open Issues and Challenges. IEEE Access. 2024;12:26839–74. doi:10.1109/ACCESS.2024.3365742.

10. R Sahana Lokesh. Open-Source Large Language Models: A Comprehensive Survey. In: 2025 International Conference on Machine Learning and Autonomous Systems (ICMLAS); 10/03/2025 - 12/03/2025; Prawet, Thailand: IEEE; 2025. p. 1096–1101. doi:10.1109/ICMLAS64557.2025.10967790.

11. Tamkin A, Brundage M, Clark J, Ganguli D. Understanding the Capabilities, Limitations, and Societal Impact of Large Language Models; 04/02/2021.

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

13. Sapkota R, Roumeliotis KI, Karkee M. AI Agents vs. Agentic AI: A Conceptual Taxonomy, Applications and Challenges; 15/05/2025.

14. Hosseini S, Seilani H. The role of agentic AI in shaping a smart future: A systematic review. Array. 2025;26:100399. doi:10.1016/j.array.2025.100399.