Software Engineering Department

Braude College of Engineering

Capstone Project – Phase A (61998)

DeMaestro:

Autonomous Multi-Agent System for Full-Stack Web Application Synthesis

CODE: 26-1-D-25

By:

Owise Zoubi: owisezoubi@gmail.com

Mohamad Atamneh: Mohamad.atamneh@e.braude.ac.il

Advisor:

Dr. Natali Levi

Link to GitHub

https://github.com/owisezoubi/DeMaestro

Table of Content

Abstract

Developing a full-stack web application requires clear requirements and careful coordination between frontend, backend, and database components. In many cases, users describe their ideas in informal text or documents that are incomplete or unclear, which makes it difficult to turn them into a working application. Existing AI-based development tools mainly focus on code generation or visual building. These solutions usually assume that requirements are already well defined, provide limited validation, and do not clearly show how user input is translated into the final system. This project presents **DeMaestro**, a system that generates a complete and runnable full-stack web application directly from user requirements. The system first analyzes and clarifies the input, presents a structured summary for user approval, and only then generates the system design and source code. The final output is an executable project that can be run locally and used as a starting point for further development.

1.0 Introduction

Developing a modern web application is a complex process that involves several stages and different areas of knowledge, including requirements analysis, system design, implementation, and validation [2][15]. In order to build a full-stack web application, a developer must understand user requirements, design the system architecture, implement frontend and backend logic, design a database, and ensure that the application can run correctly [2][15]. For experienced development teams, this process is usually managed using established software engineering practices, methodologies, and lifecycle models [2][15]. However, for individuals, students, and small organizations, transforming an idea into a working application can be difficult, time-consuming, and costly [2][15].

One of the main challenges in software development is correctly understanding and translating user requirements into technical solutions. Users usually describe their needs using natural language, documents, or informal explanations, which are often incomplete, unclear, or ambiguous [2][8]. Traditional software engineering addresses this problem through structured requirements engineering processes, where requirements analysts manually convert user input into formal specifications before development begins [2][8]. When requirements are not clearly defined, the final software may not fully meet user expectations, even if the implementation itself is technically correct [2].In recent years, artificial intelligence tools have been introduced to support software development. AI-based coding assistants such as GitHub Copilot [3] and Amazon CodeWhisperer [4] assist developers by suggesting code during implementation. Other platforms, such as Replit [6], allow users to generate or modify applications using natural language prompts. In addition, no-code and low-code platforms like Bubble [9] enable users to create applications using visual tools without writing code. These tools aim to reduce development time and lower the technical barrier for building applications [5][10].

Although these solutions provide useful support, they mainly focus on code generation rather than requirements-first development. Studies show that most AI coding assistants

assume the user already understands what to build and how the system should be structured, and therefore do not address requirement analysis or validation [5][7]. Application generation platforms often generate code in a single step, without clearly explaining how user requirements are interpreted, validated, or confirmed [7][11]. No-code platforms simplify development but limit flexibility and usually do not provide full ownership or transparency of the produced codebase, which restricts further customization and scalability [9][10].

Another limitation of current solutions is the lack of validation and accountability throughout the development process. Research in requirements engineering and AI-assisted programming shows that there is often no clear traceability between the original user requirements and the generated implementation [8][11]. When generated applications fail to meet user expectations, existing tools rarely indicate which requirement was misunderstood or missing [5][11]. As a result, users are forced to manually review, debug, and extend the generated code, which requires technical expertise and reduces the overall benefit of automation [5][11].

To address these limitations, this project introduces **DeMaestro**, a role-based system designed to generate complete and runnable full-stack web applications directly from user requirements. Although DeMaestro uses a single large language model, the system is structured around multiple logical roles, such as requirements analysis, system design, and code generation, following principles from role-based and agent-oriented system design [12][14]. These roles operate sequentially within a phase-based workflow, where the output of each phase is finalized before the next phase begins.

Importantly, DeMaestro follows a requirements-first approach inspired by established requirements engineering practices [2][8]. User input is first transformed into structured requirements and presented back to the user as a **Requirements Summary Document** for explicit approval. Only after this approval does the system proceed to blueprint and code generation. This approach improves requirements consistency, traceability, and transparency, and reduces misunderstandings in later development stages [2][8].

Based on the approved requirements, the system generates a full source code project that includes frontend, backend, and database components. The generated application can be executed and tested locally by the user without requiring manual completion of missing parts [15].

The goal of DeMaestro is not to generate production-ready software, but to provide a solid and executable starting point for further development. By producing consistent architecture, complete source code, and intermediate artifacts that improve transparency and traceability, DeMaestro aims to reduce the gap between user ideas and initial software implementations while remaining feasible within the scope of a bachelor-level software engineering project [2][15].

This book is organized as follows. Chapter 1 presents the background and motivation for the project. Chapter 2 reviews existing solutions and related work in AI-assisted software development. Chapter 3 describes system requirements. Chapter 4 explains the

methodology and development process. Chapter 5 details tools and technologies. Chapter 6 presents evaluation with users. Chapter 7 describes the product and architecture. Chapter 8 presents verification and test planning. Chapter 9 presents evaluation criteria and measurements. Chapter 10 lists prompts used during development, followed by reference

2.0 Literature Review

2.1 Overview of AI-Assisted Software Development

Artificial intelligence has become increasingly involved in software engineering, especially with the development of Large Language Models (LLMs). These models are trained on large collections of source code and natural language text, allowing them to generate code, explain programming concepts, and assist developers during implementation [1][7]. As a result, AI-assisted software development tools are now widely used in both industry and academic environments.

Most existing AI-based development tools are designed to improve productivity rather than fully automate the software development lifecycle. They mainly support the implementation phase and rely on human developers to handle higher-level tasks such as requirements analysis, architectural design, validation, and integration [2][8]. While these tools demonstrate strong performance in code generation, their ability to manage end-to-end requirements-driven development remains limited.

2.2 AI-Based Code Assistants

AI-based code assistants are one of the most common applications of LLMs in software engineering. Tools such as GitHub Copilot [3] and Amazon CodeWhisperer [4] are integrated into development environments and provide real-time code suggestions. These systems can generate functions, complete code blocks, and recommend common programming patterns.

The main advantage of code assistants is their ability to speed up the coding process and reduce repetitive tasks. Studies show that such tools can improve developer productivity, especially for routine programming work [5]. However, code assistants function primarily as assistive tools, not autonomous systems. They assume that the developer has already defined system requirements and architecture.

From a software engineering perspective, code assistants do not address the full challenge of translating user requirements into complete applications. They operate at the code level and do not provide mechanisms for requirement modeling, user confirmation, traceability, or full-stack integration guarantees [2][8].

2.3 Natural Language–Driven Application Generation

Some platforms attempt to move beyond code assistance by generating applications directly from natural language descriptions. These systems allow users to describe an

application in text and receive a working program as output. Platforms such as Replit provide features that support this form of interaction [6].

Although these systems demonstrate the potential of LLMs to automate higher-level development tasks, the literature highlights several limitations. Many application generation platforms rely on one-shot generation, where the application is generated in a single step without explicit requirements validation or user sign-off [7][8]. This approach struggles with complex or ambiguous requirements and often results in incomplete or inconsistent implementations.

In addition, these platforms provide limited transparency. Users are not shown how their requirements are interpreted, and there is usually no structured representation of requirements that can be reviewed, confirmed, and traced back to the generated code [8].

2.4 No-Code and Low-Code Development Platforms

No-code and low-code platforms aim to reduce the need for programming by allowing users to build applications using visual interfaces. Systems such as Bubble enable users to create applications without writing traditional code [9]. These platforms are effective for simple use cases and rapid prototyping.

However, the literature identifies several drawbacks. No-code platforms often limit customization and scalability, making them unsuitable for complex systems [10]. In many cases, users have limited access to the generated source code, which can lead to vendor lock-in. Furthermore, these platforms do not support formal requirement engineering processes, requirement confirmation, or systematic validation practices, which are essential for building reliable software systems [2][8].

2.5 Requirement Engineering and Semantic Challenges

Requirements engineering is a critical phase of software development. Users usually express their needs in natural language, which is often ambiguous or incomplete. Traditional software engineering addresses this issue through structured requirement elicitation, analysis, and validation techniques [2][8].

LLM-based systems face significant challenges in this area. While they can generate fluent text and syntactically correct code, they do not possess a true understanding of user intent. Research shows that AI-generated code may fail to meet actual requirements due to misinterpretation, even when the output appears correct [11]. This issue is commonly described as the semantic gap between human intent and executable software.

Most existing AI-assisted development tools do not explicitly address this problem. They generate code directly from user input without producing structured requirements and without requiring explicit user approval of the system’s interpretation, making validation and verification difficult [8][11].

2.6 Role-Based and Agent-Oriented Approaches

Recent research explores agent-oriented approaches to improve the reliability of AI-assisted software development. In these approaches, different agents or roles are responsible for specific tasks, such as planning, coding, and testing [12][14]. The goal is to decompose complex development processes into smaller and more manageable steps.

Although multi-agent systems offer theoretical advantages, practical implementations often suffer from coordination and reliability issues. Managing communication, shared state, and task dependencies between agents remains challenging [13]. As a result, fully autonomous multi-agent systems are still difficult to apply reliably to end-to-end full-stack application development.

Some studies suggest that a role-based, sequential approach, even when implemented using a single underlying model, can provide better control and stability while retaining the benefits of task decomposition [14]. This observation motivates the design choice adopted in the DeMaestro project.

2.7 Identified Research Gap

The reviewed literature shows that existing AI-assisted development tools focus mainly on code-level support and rapid prototyping. Code assistants require technical expertise, application generation platforms lack validation and transparency, and no-code platforms restrict flexibility and ownership. Furthermore, the semantic gap between user intent and executable software remains largely unresolved.

There is a clear need for a system that can transform unstructured user requirements into a complete and runnable full-stack application while following software engineering principles such as requirement modeling, explicit user confirmation, traceability, and validation [2][8]. This gap motivates the DeMaestro project, which adopts a role-based architecture using a single large language model to generate executable source code in a structured and transparent manner.

3.0 System Requirements and Methodology

3.1 Functional Requirements

- **FR1 – Text Input**
  The system shall allow the user to type application requirements in a chat input.
- **FR2 – Document Upload**
  The system shall allow the user to upload a requirements document (PDF).
- **FR3 – Project Association**
  The system shall associate each input (text or PDF) with the current user project.
- **FR4 – Requirement Structuring**
  The system shall convert the raw user input into structured requirements.
- **FR5 – Ambiguity Detection**
  The system shall detect missing or unclear requirement information.

- **FR6 – Clarification Questions**
  When ambiguity is detected, the system shall ask clarification questions via chat.
- **FR7 – Clarification Update**
  The system shall update the structured requirements using the user's answers.
- **FR8 – Requirements Summary**
  The system shall generate a human-readable Requirements Summary Document.
- **FR9 – User Approval**
  The system shall require user approval of the Requirements Summary before continuing.
- **FR10 – Blueprint Generation**
  After approval, the system shall generate an application blueprint.
- **FR11 – Code Generation**
  The system shall generate frontend, backend, and database code from the blueprint.
- **FR12 – Verification**
  The system shall verify basic consistency between frontend and backend interfaces.
- **FR13 – Project Download**
  The system shall allow the user to download the generated project as a ZIP file.
- **FR14 – Authentication & History**
  The system shall require login and allow users to access their project history.

3.2 Non-Functional Requirements

- **NFR1 – Input Format**
  User requirements shall be submitted only as text or PDF documents.
- **NFR2 – Reliability**
  Most generated projects shall run locally without syntax errors.
- **NFR3 – Interface Correctness**
  Frontend API calls shall match backend routes.
- **NFR4 – Maintainability**
  Generated code shall follow a clear and simple folder structure.
- **NFR5 – Usability**
  A new user shall submit requirements within approximately 60 seconds.
- **NFR6 – Performance**
  Project generation shall complete within five minutes on average.
- **NFR7 – Traceability**
  Each project shall include a Requirements Summary Document.
- **NFR8 – Security**
  Users shall only access their own projects.
- **NFR9 – Verifiability**
  The system shall generate logs for each generation stage.

4.0 Methodology & Development Process

In this project, we followed a design-and-build development approach, where the primary contribution is a working and executable system rather than a purely theoretical model. This approach is suitable for a bachelor-level software engineering project in which the outcome is evaluated through implementation, verification activities, and user-facing evaluation.

At the start of the project, we investigated recurring challenges in AI-assisted software development, especially for generating complete full-stack applications from natural-language requirements. Prior work and existing tools report common issues such as ambiguous input, weak architectural consistency, limited transparency, and incomplete end-to-end integration. These observations guided our methodology toward a controlled pipeline that emphasizes traceability, staged outputs, and pre-generation validation.

Based on this analysis, we designed a structured development process that constrains the AI component through explicit stages. Each stage produces a concrete artifact (e.g., structured requirements, a blueprint, verification logs) that is stored and can be reviewed before proceeding. This reduces uncontrolled “one-shot” generation and improves maintainability and accountability of the produced output.

4.1 Research and Design Decisions

Before implementation, we reviewed existing AI-based code generation solutions and relevant software engineering literature, focusing on the role of requirements engineering and the risks of generating code from incomplete or ambiguous input. This informed two key design decisions:

1. **Multi-stage pipeline:** The system must separate *requirements analysis*, *system design*, and *code generation* into distinct phases, each producing explicit intermediate artifacts that guide later phases.

2. **Clarification + confirmation before generation:** The system must include a mechanism to detect ambiguity and collect missing details, and it must provide a “requirements sign-off” step so the user can confirm the system’s interpretation before code generation begins.

Together, these decisions enforce a requirements-first workflow and reduce error propagation from early misunderstanding into later implementation.

4.2 Use of Artificial Intelligence

DeMaestro uses a single large language model, Google Gemini, as the core AI component for analysis and generation. The model is not used as a free-form chatbot; instead, it is invoked programmatically within a controlled workflow. Although only one model is used, it performs multiple *logical roles* (e.g., requirements analyst, architect, code generator)

across sequential phases. This role-based structure improves predictability and reduces inconsistency by constraining the model to produce specific artifacts per stage.

To further improve stability, the AI is restricted to a predefined technology stack and expected output formats (e.g., structured JSON for requirements and blueprints). This reduces variability and simplifies downstream verification.

4.3 System-Level Development Process

Alongside the AI workflow, we implemented a web platform that acts as the interface between users and the generation pipeline. The platform is responsible for user interaction, authentication, process orchestration, and persistent storage of project artifacts, while the AI component focuses on reasoning and generation tasks.

The system supports user signup/login via Firebase Authentication to enable personal workspaces and access control. After authentication, each user can create multiple projects. Each project stores all artifacts produced throughout the pipeline, including raw inputs, structured requirements, blueprint outputs, verification logs, and the final generated source code package. Project data is stored using Firebase Firestore in a per-user, per-project structure.

This separation of concerns improves maintainability: the web platform manages users, projects, and execution flow, while the AI pipeline focuses exclusively on requirements interpretation and software artifact generation.

4.4 AI Agent Workflow and Code Generation Process

The AI workflow is implemented as a staged pipeline that transforms unstructured user input into executable source code. The stages are designed to produce explicit artifacts that can be reviewed and validated before the system proceeds.**This workflow is illustrated in Figure 1.**

4.4.1 Raw Requirements Handling

The workflow begins when the user submits application requirements in natural language (chat) or via an uploaded document. These inputs are stored as *raw requirements*, representing the initial and unprocessed form of user intent. At this stage, the system does not assume correctness or completeness.

4.4.2 Requirements Structuring and Clarification

The AI analyzes the raw requirements and converts them into a structured requirements representation in JSON format. This structured form acts as a requirements contract and supports controlled updates. Because natural-language input is frequently ambiguous or incomplete, the system runs a clarification loop: when missing or unclear details are detected, the system asks targeted clarification questions. The user's answers are used to update and finalize the structured requirements.

4.4.3 Requirements Review and User Sign-Off

After the structured requirements are finalized, DeMaestro generates a human-readable *Requirements Summary Document* derived from the final requirements representation. This document is shown to the user before design or code generation begins. The user explicitly confirms that the requirements reflect their intent (or requests corrections), and only then the workflow continues. This “sign-off” step maintains requirements consistency by validating the system’s understanding prior to implementation and reducing downstream misinterpretations.

4.4.4 Design and Blueprint Generation

Once requirements are confirmed, the AI produces a high-level application blueprint. The blueprint defines the database model, backend API routes, and frontend pages/components, along with their responsibilities and interface expectations. This stage ensures architectural consistency and creates an explicit bridge between requirements and implementation.

4.4.5 Source Code Generation

Using the blueprint as a strict guide, the AI generates the full source code for the frontend, backend, and application database layer. Code is produced using predefined templates and a consistent folder structure to improve maintainability.

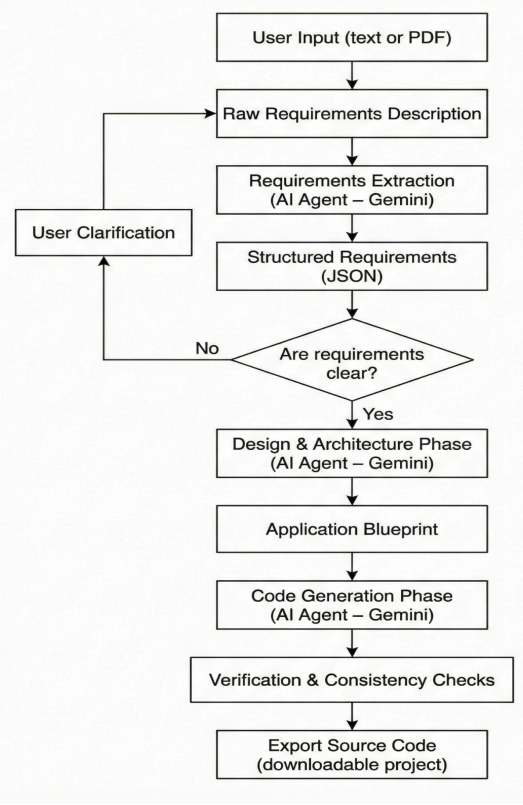

Figure 1: DeMaestro Workflow and Code Generation Process

The result is a complete project intended to run locally without requiring the user to manually “fill in missing parts.”

4.5 Verification and Source Code Export Mechanism

Before delivering the generated project, the system performs basic verification checks on the produced artifacts. These checks focus on common integration failures such as missing files and mismatches between frontend API calls and backend route definitions. The results are recorded in a verification log stored under the user’s project.

After verification, the system packages the generated application into a structured project folder and exports it as a downloadable ZIP archive. The final output is therefore not only code snippets, but an executable starting point that users can run locally and extend, aligned with the project’s goal of producing a practical baseline rather than production-ready software.

5.0 Tools and Technologies

This chapter presents the tools and technologies used to design and implement the DeMaestro system. The selected technologies were chosen to support rapid development, integration with artificial intelligence services, user authentication, and persistent storage of user projects. Preference was given to widely used and well-documented technologies to ensure reliability and feasibility within a bachelor-level software engineering project.

5.1 Artificial Intelligence Technology

Google Gemini API

The core artificial intelligence component of the system is Google Gemini, accessed through its public API. Gemini is used as a large language model to perform all AI-related tasks in the system, including requirements analysis, clarification question generation, Requirements Summary Document generation (for user approval), system design/blueprint generation, and full-stack source code generation.

Gemini is integrated as a controlled component that operates within a predefined workflow. Instead of free-form generation, the backend invokes the model programmatically at specific pipeline stages and expects structured outputs (e.g., JSON requirements and blueprint artifacts). This staged integration improves consistency and traceability while reducing the risk of uncontrolled one-shot generation.

5.2 Frontend Technologies

5.2.1 Web Client (Browser-Based Application)

The frontend of the system is implemented as a browser-based web application. It serves as the main interface between the user and DeMaestro. Through this interface, users can:

- register and sign up to the system,
- log in to their personal account,
- create and manage projects,
- submit requirements using a chat interface,
- upload requirement documents such as PDF files,
- review and approve the Requirements Summary Document before generation,
- view project history,
- and download generated source code.

All AI logic is handled on the server side. The frontend is responsible for user interaction, stage visibility, and presentation.

5.2.2 React

The user interface is built using React, a popular JavaScript library for building interactive web applications. React was chosen because it supports a component-based architecture, which is suitable for building and maintaining interfaces such as chat views, approval dialogs, and project dashboards. React also supports dynamic UI updates in response to clarification prompts and generation progress events.

5.2.3 Tailwind CSS

For styling, the system uses Tailwind CSS, a utility-first CSS framework. Tailwind allows fast development of clean and responsive interfaces without maintaining large custom CSS files. Using utility classes also helps maintain a consistent visual design across the application.

5.3 Backend Technologies

5.3.1 Backend Orchestration Server (Python)

The backend of the system is implemented using Python and acts as the central controller of the DeMaestro platform. It is responsible for:

- handling requests from the frontend,
- managing authentication and authorization,
- controlling the AI workflow stages,
- storing and retrieving project artifacts,
- performing verification routines,
- and packaging the final source code export.

Python was chosen because it supports rapid backend development, integrates well with AI-driven workflows, and is effective for text processing and pipeline orchestration. The backend exposes APIs that allow the frontend to communicate with the system and ensures that pipeline stages execute in the correct order.

5.3.2 Authentication and User Management (Firebase Authentication)

User authentication is implemented using Firebase Authentication. It provides secure account management, signup, login, and session handling without requiring custom authentication logic. This enables personal user workspaces and ensures that each user can access only their own projects.

5.4 Data Storage and Project Persistence

5.4.1 Firebase Firestore (Platform Storage)

The system uses Firebase Firestore as its primary platform database for storing user and project data. Firestore is used to store:

- user account identifiers and project ownership metadata,
- raw requirements input (text and document references),
- structured requirements (JSON),
- the Requirements Summary Document (user-approved),
- application blueprints,
- verification logs,
- and project status information.

Firestore is selected because its document-based model aligns well with JSON artifacts produced by the AI pipeline and supports flexible project organization per user.

5.4.2 Generated Application Database vs Platform Storage

Firestore is used for DeMaestro platform persistence (users, projects, and artifacts). The generated applications themselves use a separate application database (e.g., SQLite or PostgreSQL), defined during blueprint generation. The blueprint includes the schema for the generated application database, independent of Firestore.

5.5 File Packaging and Source Code Export

After the AI pipeline completes code generation and verification, the system packages the generated full-stack application source code into a downloadable project folder. This folder includes all files required to run the application locally. The system exports the folder as a ZIP archive to ensure users receive a complete runnable baseline rather than individual code snippets.

5.6 Development and Version Control Tools

5.6.1 Visual Studio Code

Visual Studio Code is used as the primary development environment. It supports both frontend and backend development and provides strong extension support for JavaScript/React and Python.

5.6.2 Git and GitHub

Git is used for version control throughout the project, and GitHub is used to host the project repository. These tools support collaboration, track changes over time, and provide a reliable backup of code and documentation.

6.0 User Evaluation (SUS)

To evaluate the usability of the DeMaestro platform, we use the System Usability Scale (SUS). SUS is a widely used standardized questionnaire for measuring perceived usability of interactive systems. It provides a comparable usability score based on ten items rated on a five-point Likert scale (1 = strongly disagree, 5 = strongly agree).

6.1 Why We Chose SUS

Our choice of SUS is aligned with the research gaps discussed in Chapter 2. Existing AI development tools often require technical expertise and provide limited transparency regarding requirement interpretation [2][8]. DeMaestro addresses these issues through a structured workflow, clarification loop, and an explicit user approval gate via the Requirements Summary Document. SUS includes items that reflect ease of use, confidence, perceived complexity, and integration quality, making it appropriate for evaluating whether DeMaestro achieves its usability goals.

6.2 The Questionnaire

1. I think that I would like to use DeMaestro frequently.
2. I found DeMaestro unnecessarily complex.
3. I thought DeMaestro was easy to use.
4. I think that I would need the support of a technical person to be able to use DeMaestro.
5. I found the various functions in DeMaestro were well integrated.
6. I thought there was too much inconsistency in DeMaestro.
7. I would imagine that most people would learn to use DeMaestro very quickly.
8. I found DeMaestro very cumbersome to use.
9. I felt very confident using DeMaestro.
10. I needed to learn a lot of things before I could get going with DeMaestro.

7.0 Product

7.1 Project Architecture

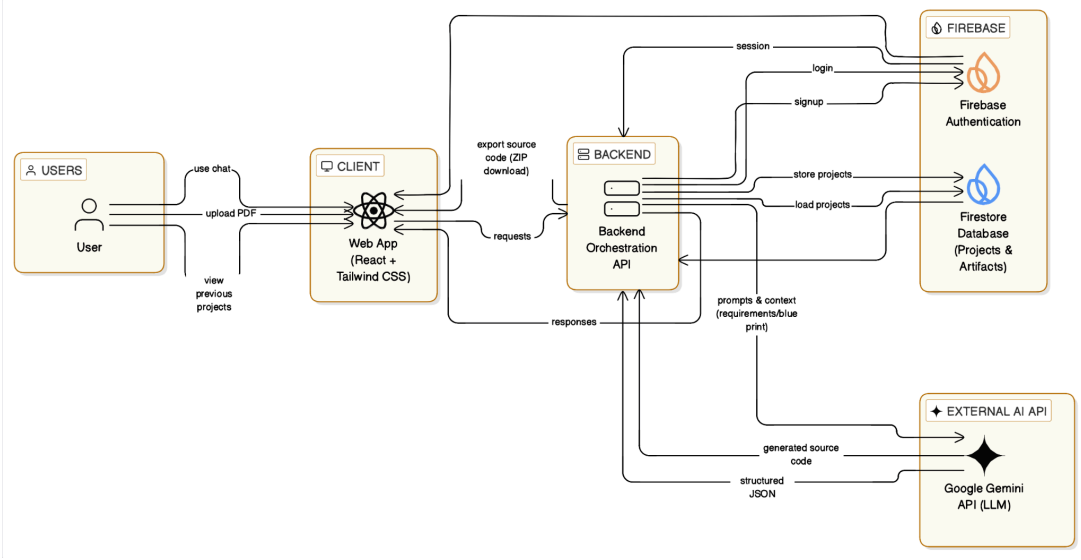

Figure 2: DeMaestro Project Architecture

The high-level architecture of the DeMaestro system is presented in **(Figure 2)**.

- **User Interface Layer:** Web app where users log in, chat with the AI, upload requirement files, approve the Requirements Summary, view project history, and download the generated code.
- **AI Chat Interaction:** Users describe needs in chat; the system asks clarification questions when details are missing or unclear.
- **Requirements Confirmation:** After structuring/clarifying, the system shows a Requirements Summary Document for user sign-off before continuing to blueprint and code generation.
- **Web Client (React + Tailwind):** Handles UI, progress display, clarification flow, summary approval, and export screens.
- **Backend Orchestration Server:** Controls authentication, project lifecycle, pipeline stages, artifact management, verification, and ZIP export.
- **AI Agent (Gemini API):** Produces structured requirements, the requirements summary, the blueprint, and the full-stack source code.
- **Project Storage (Firebase Auth + Firestore):** Stores users, projects, and all generated artifacts per project.
- **Verification & Export:** Checks completeness and API consistency, then packages the project into a downloadable ZIP.

7.2 Use Case Diagram

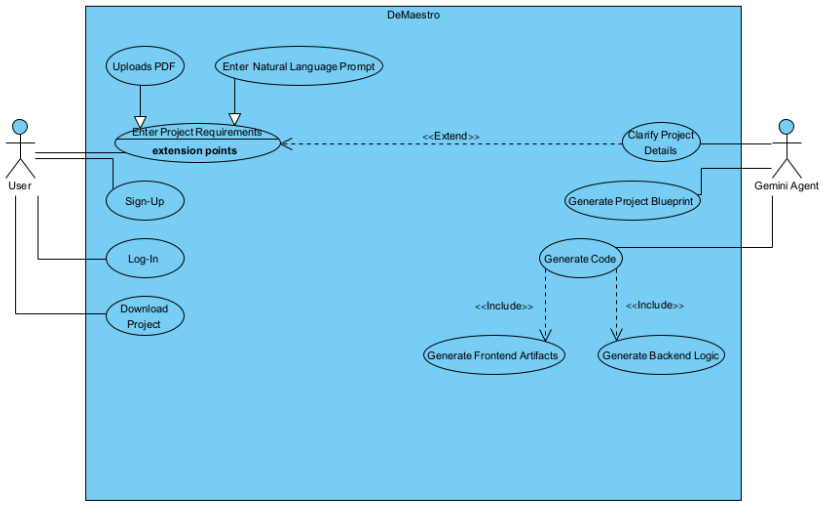

Figure 3: DeMaestro Use Case Diagram

The use case diagram (**Figure 3**) summarizes how the user interacts with DeMaestro:

- **Enter Project Requirements:** The user starts by typing requirements or uploading a PDF.
- **Clarify Project Details (extends):** If the input is unclear, the system asks clarification questions and updates the requirements.
- **Review & Approve Requirements Summary:** The user reviews the generated requirements summary and approves it (or requests fixes) before generation continues.
- **Generate Project Blueprint:** The system creates a blueprint with the database schema, API routes, and frontend pages.
- **Generate Code:** Full-stack source code is generated based on the blueprint.
- **Download Project:** The system packages everything into a ZIP file for the user to download.

7.3 Activity Diagram Description

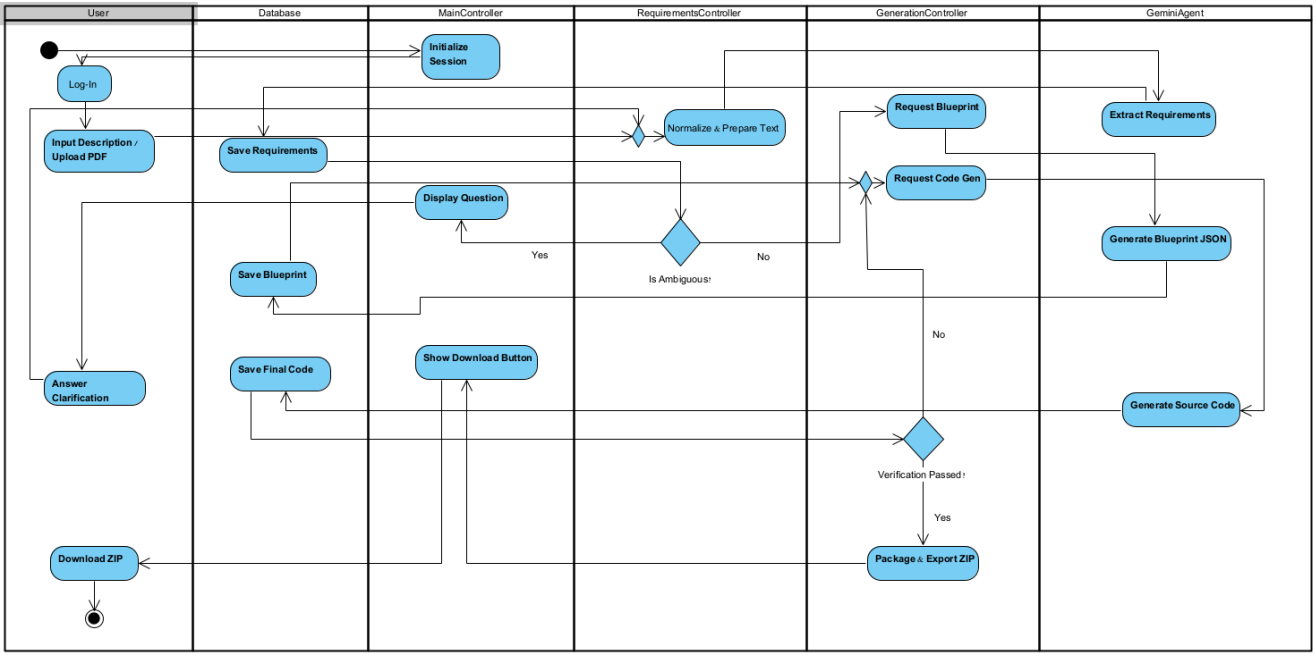

Figure 4: DeMaestro Activity Diagram and Workflow

The activity diagram (**Figure 4**) illustrates the DeMaestro workflow across the user, system controllers, AI agent, and database:

- The user creates a new project and submits requirements (prompt or PDF).
- The RequirementsController stores the raw input and requests requirement structuring from the AI.
- If ambiguity is detected, the system enters a clarification loop and updates structured requirements based on user answers.
- The system generates a Requirements Summary Document and presents it to the user.
- **Decision: User Approved Requirements?** If not approved, the workflow returns to clarification and updates the requirements. If approved, the workflow continues.
- The GenerationController requests blueprint generation and stores the blueprint artifact.
- The AI generates full-stack source code according to the blueprint.
- A verification routine runs to check interface consistency and file completeness.
- If verification passes, the system packages the project into a ZIP archive for download.

7.4 Package Diagram

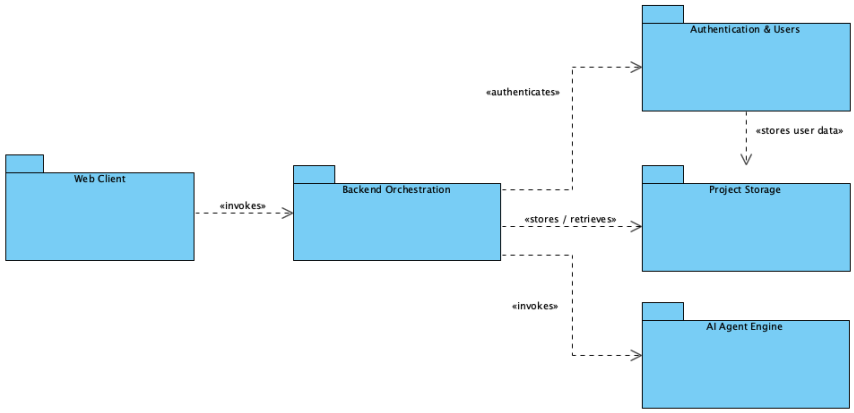

Figure 5: DeMaestro Package Diagram

The package diagram (**Figure 5**) presents the system as a set of major components: Web Client, Backend Orchestration, AI Agent Engine, Authentication & Users, and Project Storage. The Backend Orchestration component coordinates the stage-based workflow and enforces that requirements approval occurs before blueprint and code generation. The AI Agent Engine is separated from user interface and storage to maintain control and improve maintainability.

7.5 Prototype

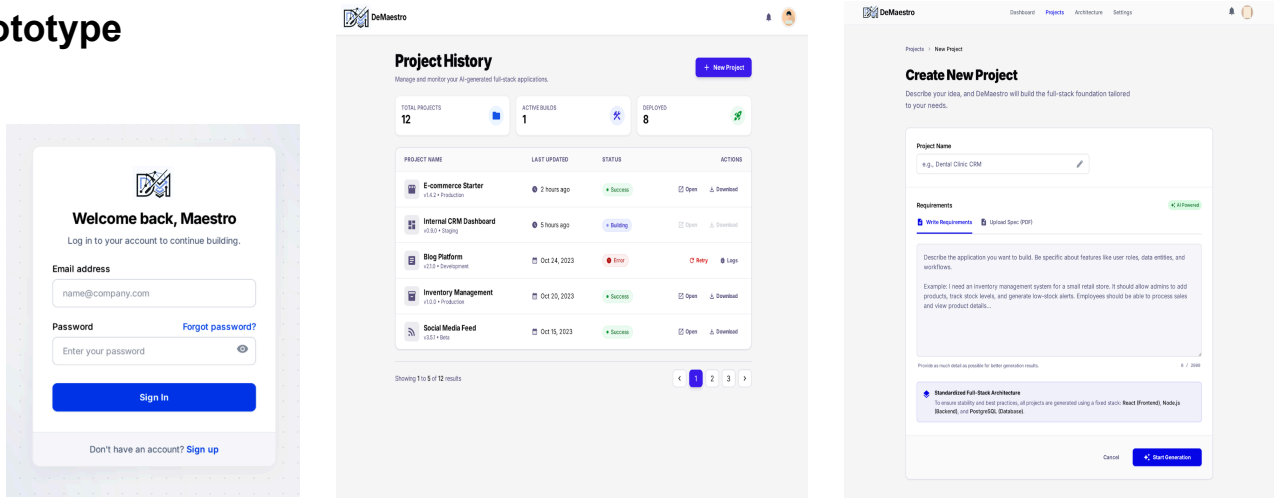

Figure 6: DeMaestro Prototype Interfaces

The user interface of the DeMaestro platform consists of three main screens (**Figure 6**). The login screen allows users to securely access the system using their personal credentials. After authentication, users are directed to the project history dashboard, where they can view, manage, and download previously generated projects or create new ones. When creating a new project, users are presented with a screen that allows them to define a project name and submit application requirements either through a chat based text input

or by uploading a PDF document, which then initiates the AI-driven full-stack code generation process.

7.6 Expected Challenges

7.6.1 Ambiguity in User Requirements

A major challenge is handling ambiguous or incomplete user requirements. Users often describe their ideas informally, omit critical details, or use terms that may have multiple interpretations. This makes accurate requirement extraction difficult. Research in requirements engineering shows that ambiguity is a fundamental problem even in traditional software projects and requires explicit validation before implementation [2,8].

DeMaestro mitigates this risk using a clarification loop and a **Requirements Summary Document** that the user must review and approve before blueprint and code generation. However, there remains a risk that the final structured requirements and approved summary may still not fully reflect the user's original intent if assumptions are misunderstood, if the user provides insufficient clarifications, or if requirements change after approval.

7.6.2 Consistency of AI-Generated Output

Large language models may produce variable outputs for similar inputs, which can lead to inconsistencies across generated components. This issue is documented in studies evaluating code generation models [7].

In the context of DeMaestro, inconsistencies may appear as mismatched API routes, inconsistent naming, or incompatible data models. In addition, inconsistencies can occur between pipeline artifacts (structured requirements, the approved requirements summary, the blueprint, and generated code) if details drift across stages. While the system mitigates this risk by enforcing a fixed technology stack and staged generation process, maintaining consistency across multiple generation stages remains a challenge.

7.6.3 Full-Stack Integration Challenges

Generating a complete application requires coordination between frontend, backend, and application database layers. Ensuring that these layers integrate correctly is significantly more complex than generating isolated code snippets. Existing studies on AI-assisted programming indicate that maintaining architectural coherence across multiple files and components is a persistent limitation of current tools [5,11].

Although DeMaestro includes blueprint constraints and verification checks to detect common integration issues, some errors may still occur at runtime.

7.6.4 Limited Verification and Testing Capabilities

The verification stage focuses on basic structural and integration checks, such as file completeness and interface alignment. However, full functional testing of generated

applications is difficult to automate. Software engineering literature emphasizes that verification and validation are critical but resource-intensive activities [15,17].

As a result, some logical errors—such as incorrect business rules, missing edge cases, or misinterpreted semantics—may only be identified through manual testing by the user.

7.6.5 Performance and Scalability Constraints

Full-stack code generation is computationally intensive. As the number of users and concurrent projects increases, system performance may degrade, leading to longer generation times. This challenge is amplified by reliance on external AI services, which introduce latency and usage limits [16].

Designing the system to manage concurrent requests while maintaining acceptable response times is therefore a key scalability concern.

7.6.6 Dependency on External AI Services

The system relies on the Google Gemini API for all reasoning and generation tasks. Any changes in API behavior, pricing, availability, or usage limits may directly impact system functionality. Such dependency on external AI infrastructure is a known limitation in AI-based systems and reduces the level of control available to system designers [13,16].

This dependency must be carefully managed at the architectural level.

7.6.7 User Account and Project Management Complexity

Supporting user signup, login, and project history introduces challenges related to data organization, access control, and long-term storage. Each project must be securely associated with the correct user and protected from unauthorized access. Managing persistent project artifacts and generated source code while maintaining performance and security requires careful system design, especially as the number of stored projects grows [2].

7.6.8 Balancing Usability and System Control

A key challenge is balancing user-friendly interaction with the need for structured and precise input. Allowing free-form natural language input improves usability but increases ambiguity and processing complexity. Research on AI-assisted development tools highlights the difficulty of balancing flexibility and control in human–AI interaction [5,11].

DeMaestro addresses this through a clarification loop and a user approval gate (Requirements Summary Document), but achieving the optimal balance without making the process cumbersome remains an ongoing challenge.

8.0 Verification Plan

Our testing plan focuses on four primary components of the system:

ID	Requirement	Acceptance Test	Expected Result
AT1	FR1 – Requirement Input	User enters text or uploads a PDF	Input is accepted and displayed in the chat
AT2	FR3 – Requirement Clarification	User submits a vague request	System asks clarification questions
AT3	FR7/FR8 – Backend & Frontend Generation	User completes clarification	Frontend and backend code are generated
AT4	FR9 – Integration Verification	Code generation completes	Frontend API calls match backend routes
AT5	FR12 – Source Code Download	User clicks “Download Project”	ZIP file is downloaded successfully
AT6	FR13 – Project Management	User revisits dashboard	Previous projects are visible
AT7	FR14 – User Authentication	User logs in	User accesses only their own projects
AT8	NFR5 – Usability	New user submits requirements	Done within ~60 seconds
AT9	NFR1 – Reliability	Generated project is run locally	Application starts without syntax errors
AT10	FR14 – User Authentication (Sign Up)	New user signs up with email/password	Account is created successfully and user is redirected to the dashboard
AT11	FR14 – User Authentication (Log In)	Existing user logs in	Login succeeds and the user sees only their own projects
AT12	NFR9 – Security (Access Control)	User tries to open a project that belongs to another user (by changing URL/id)	Access is denied (error/redirect) and no project data is exposed
AT13	NFR5 – Usability	First-time user creates a project and submits requirements	User completes submission within ~60 seconds (excluding waiting for AI generation)
AT14	NFR1 – Reliability	User downloads a generated project and runs it locally using the provided steps	App starts without syntax/build errors

9.0 Evaluation

The evaluation criteria focus on:

9.1 Usability

How easily users can describe requirements using natural language or PDF uploads and how intuitively they can complete the clarification and approval steps (including Requirements Summary Document sign-off) before generation.

9.2 Performance

Ensuring the end-to-end generation time (from approved requirements to downloadable ZIP) remains under five minutes on average for standard-size projects, while maintaining responsive UI interaction.

9.3 Accuracy

Measuring reliability of generated outputs, including a high “Code Validity Rate” (projects running without syntax/build errors) and verified consistency between frontend API calls and backend routes.

9.4 Adaptability

Measuring the system’s ability to handle vague or incomplete inputs by successfully triggering the clarification loop and reaching an approved requirements state prior to generation.

9.5 Efficiency

Measuring time saved compared to manual initialization of a full-stack environment by comparing typical setup time versus automated generation time.

9.6 Functionality

Ensuring the export feature consistently produces a valid ZIP archive containing all necessary artifacts (frontend, backend, and database configuration) ready for local execution.

Prompts Used (Gemini + ChatGPT)

- “Expand the introduction to include more detail about how manual full-stack setup is a major bottleneck for junior developers and students.”
- “Write a background section focusing on how LLMs are transforming software engineering, specifically automated code synthesis.”
- “Rewrite the architecture section to focus more on controller separation and staged orchestration.”
- “Enhance the clarification loop description by adding steps on how the system detects ambiguous requirements before asking questions.”
- “Write a brief description of how DeMaestro will incorporate a verification routine to check API routes before download.”
- “Generate a Requirements Summary Document from finalized structured requirements JSON, preserving requirement IDs and listing assumptions for user approval.”
- “Give use case diagram ideas with include/extend relationships regarding blueprint generation.”
- “Simplify the activity diagram explanation and align it with the controller swimlanes.”
- “Enhance the academic language of the project scope and methodology.”

References

- [1] Brown, T. et al., “Language Models are Few-Shot Learners,” NeurIPS, 2020.
- [2] Sommerville, I., *Software Engineering*, 10th Edition, Pearson, 2015.
- [3] GitHub Copilot, product page.
- [4] Amazon CodeWhisperer, product page.
- [5] Vaithilingam, P. et al., “Expectations, Outcomes, and Challenges of Using AI-Based Code Generation Tools,” CHI, 2022.
- [6] Replit, product website.
- [7] Chen, M. et al., “Evaluating Large Language Models Trained on Code,” arXiv:2107.03374, 2021.

  [8] Nuseibeh, B., and Easterbrook, S., “Requirements Engineering: A Roadmap,” 2000.
- [9] Bubble, product website.
- [10] Poppendieck, M., and Poppendieck, T., *Lean Software Development*, Addison-Wesley, 2003.
- [11] Kulkarni, C. et al., “Semantic Gaps in AI-Assisted Programming,” ACM TOSEM, 2021.
- [12] Wooldridge, M., *An Introduction to Multi-Agent Systems*, 2nd Ed., Wiley, 2009.
- [13] Stone, P. et al., “Artificial Intelligence and Life in 2030,” Stanford, 2016.
- [14] Xi, Z. et al., “The Rise and Potential of Large Language Model Based Agents,” arXiv:2309.07864, 2023.
- [15] Pressman, R. S., and Maxim, B. R., *Software Engineering: A Practitioner’s Approach*, 8th Ed., McGraw-Hill, 2014.
- [16] Google, “Gemini: A Family of Highly Capable Multimodal Models,” Google AI Blog, 2023.
- [17] Humble, J., and Farley, D., *Continuous Delivery*, Addison-Wesley, 2010.