KING'S College LONDON

Faculty of Natural, Mathematical, and Engineering Sciences
Department of Informatics

Software Engineering Group Project

One-day Proof of Concept

Team Runtime Terrors
Enzo Bestetti
Isabella Mclean
Jacelyne Tan
Krystian Augustynowicz
Lucia Garces Gutierrez
Markus Meresma
Nishcal Gurung
Reza Samaei
Usman Khan
Wealthie Tjendera

9th April 2025

Contents

Chapter 1

Introduction

In an era where businesses are increasingly eager to explore the potential of **Generative AI**, the ability to quickly prototype and experiment with AI-driven solutions has become a valuable asset. This project, developed in collaboration with **Amazon Web Services**, aims to bridge this gap by creating a streamlined web portal that enables users to describe their business use cases and receive an automatically generated lightweight proof of concept (PoC) immediately. By lowering the technical barriers to AI adoption, this system empowers businesses to engage with cutting-edge AI technologies in a practical and hands-on manner, accelerating their innovation cycles.

Mathematically, let the space of all possible business use cases be represented as $\mathcal{U}$, and let $f : \mathcal{U} \to \mathcal{P}$ be a function mapping each business use case $u \in \mathcal{U}$ to a generated PoC $p \in \mathcal{P}$. The goal of this system is to approximate an optimal function f^* that maximizes *relevance* and *feasibility* of the PoC within practical constraints:

$$f^* = \arg\max_f \sum_{i=1}^{n} \left(R(f(u_i)) + F(f(u_i))\right)$$

where:

- $R(p)$ is the *relevance score of the PoC* p,
- $F(p)$ is the *feasibility score* of the PoC p,
- n is the number of use cases processed.

The system is a *web-based application* built using the **Kotlin** programming language and the **Ktor** framework, leveraging **AWS services** for authentication, scalability, integration, and deployment. It provides a *secure, cloud-enabled environment* where users can interact with AI-driven solutions tailored to their specific needs.

The key *technology stack* consists of:

- **Backend**: Developed in Kotlin using the Ktor framework to build an efficient server-side API.
- **Frontend**: Developed in TypeScript using the React framework and Shadcn to build a client application with pleasant and responsive UI/UX elements.

- **Cloud Infrastructure**: Hosted on AWS, leveraging its scalability for dynamic PoC generation.
- **AI-driven PoC Generation**: Utilising large language models (LLMs) to tailor PoCs to business needs.

This system showcases not only the feasibility of generating rapid PoCs but also serves as a case study in designing scalable AI-driven enterprise solutions. By aligning business objectives with technical innovation, this project exemplifies how organizations can efficiently prototype and experiment with generative AI in a structured manner.

Chapter 2

Objectives and stakeholders

2.1 Project objectives

The primary objective of the One-Day-PoC project is to enable AWS customers to rapidly experiment with and visualise potential solutions to their business challenges with generative AI. This objective directly addresses a critical business need: reducing the time and financial investment required to develop proof of concept prototypes, allowing customers to focus on fine-tuning *what* they want to build, rather than *how*.

Specifically, the project aims to:

1. **Accelerate innovation cycles** by compressing the traditional weeks-long PoC development process into a single day, allowing AWS customers to quickly validate or reject potential technology solutions.
2. **Lower the barrier to entry** for non-technical stakeholders to engage with advanced technology by eliminating the need for specialised technical knowledge in the initial prototyping phase.
3. **Enhance customer experience** by providing AWS customers with tangible, interactive demonstrations of how generative AI can address their specific business challenges.
4. **Drive AWS service adoption** by showcasing the capabilities of the AWS technology stack to solve real-world business problems, potentially leading to increased service utilisation.

These objectives align with Amazon's customer-obsessed philosophy by empowering customers to experiment with cutting-edge technology solutions with minimal risk and investment, ultimately helping them to invent and simplify with technology.

2.2 Stakeholders

Using the D.A.N.C.E. framework (Discover, Assess, Navigate, Communicate, Engage), we have identified and analyzed the stakeholders for the One-Day-PoC project.

2.2.1 Stakeholder analysis

2.2.1.1 Discover: Stakeholder Identification

Internal Stakeholders:

- AWS Solutions Architects and Sales Teams, represented by Nate Powell and Patrick Bradshaw
- Project Development Team (Team Runtime Terrors)

External Stakeholders:

- Non-technical AWS customers (primary users)
- End customers of AWS clients

2.2.1.2 Assess: Stakeholder Interests, Influence and Impact

AWS Solutions Architects and Sales Teams:

- **Interest**: High - They require efficient and effective tools to demonstrate AWS capabilities and close sales.
- **Influence**: High - As the client, their requirements and input drove the development of the solution.
- **Impact**: Will use the system as a sales enablement tool to showcase AWS services.

Non-technical AWS Customers:

- **Interest**: High - Seeking solutions to their business problems that could be quickly prototyped.
- **Influence**: High - As primary users, their adoption and feedback will drive the future development of the solution.
- **Impact**: Their feedback and usage patterns directly determine project success.

End Customers of AWS Clients:

- **Interest**: Low - Not directly involved but ultimate beneficiaries.
- **Influence**: Low - Indirect influence through AWS customers.
- **Impact**: Will experience the benefits of solutions developed using the system.

2.2.1.3 Navigate: Managing Stakeholder Dynamics

Throughout the project lifecycle, we experienced minimal conflicts among stakeholders. This harmonious dynamic can be attributed to several factors:

- **Clear Direction from AWS**: The requirements and objectives were clearly communicated by AWS representatives, providing our team with a well-defined scope and vision.
- **Aligned Priorities**: All stakeholders shared the common goal of creating a tool that would benefit both AWS and their customers, resulting in naturally aligned priorities.

- **Focused Stakeholder Group**: With a relatively streamlined set of stakeholders, the project avoided the complexity that can arise from managing numerous competing interests.

This alignment of stakeholder interests enabled our team to focus on development activities rather than managing conflicts, significantly enhancing our productivity and ability to meet project objectives.

2.2.1.4 Communicate and Engage: Stakeholder Communication and Engagement Strategies

Our communication and engagement approach was structured around regular, meaningful interactions with key stakeholders:

Fortnightly Meetings with AWS Representatives

- Scheduled at the end of each development sprint
- Conducted via Amazon Chime
- These meetings served several purposes:
  - Presenting sprint achievements and demonstrating newly implemented features
  - Gathering feedback and validating that development aligned with AWS requirements
  - Discussing plans for the upcoming sprint and adjusting priorities as needed

Although direct engagement with end-users (non-technical AWS customers) was not possible during development, we incorporated AWS representatives' insights about their customers' needs, and design decisions were consistently evaluated from the end-user perspective.

2.2.2 Key stakeholders

Based on our stakeholder analysis, we identified two key stakeholder groups whose needs and expectations were prioritized throughout the project:

1. **AWS Solutions Architects and Sales Teams (represented by Nate Powell and Patrick Bradshaw)**: As our direct client contact, they provided critical requirements, feedback, and domain expertise. Their influence was paramount as they represented AWS's interests and would ultimately be responsible for introducing the solution to end-users. Our fortnightly meetings ensured their continued engagement and alignment with project direction.
2. **Non-technical AWS Customers**: While not directly involved in the development process, these stakeholders represented the primary end-users of the system. Their needs - specifically, the ability to quickly generate functional prototypes without technical expertise - drove key design decisions, leading to a particularly user-friendly interface. Understanding their perspective was essential to creating a solution that would deliver genuine business value.

By maintaining clear communication with AWS representatives and consistently considering the needs of non-technical end-users, we ensured that the One-Day-PoC project remained focused on delivering meaningful value to both AWS and their customers. This targeted approach to stakeholder management contributed significantly to the project's successful execution and delivery.

Chapter 3

Specifications

3.1 Overview

This section outlines the specifications for the One-Day-PoC generator system, a web portal that enables customers to automatically generate proof-of-concept applications based on their business use cases. The software therefore requires to create a solution to, on a surface level, simple end-user interaction - allowing a user to submit a prompt and receive a working prototype which they can further tweak.

3.2 Functional Requirements

3.2.1 User Prompting

- **Description** The system provides an interface through which a user can enter prompts for PoC generation.
- **Implementation**
  - **Landing Page** The system rovides an input box where users can enter text to initiate a new conversation and trigger PoC Generation [found in Landing-Page.tsx and InputBox.tsx]
  - **Chat** The system provides a chat-like interface alongside a generated PoC to allow for a converation. Users can talk to the system in a natural way, tweaking the PoC with every new message [found in Chat.tsx and ChatScreen.tsx]
- **Contribution to Project Objectives** This functionality is the most basic feature of the system: to receive inputs from users. This input will subsequently be used in PoC generation.

3.2.2 User Prompt Processing

- The system accepts natural language prompts from users and processes them into a structured format suitable for AI processing.
- **Implementation**
  - **Prompt retrieval** The front end retrieves plain text input from the user [found in InputBox.tsx] and sends the input to the server [through the use of the Chat.tsx hook and FrontEndAPI.ts].
  - **Prompt Sanitisation** The system sanitises user inputs to remove potential security issues and standardise formatting [found in SanitsationTools.kt] allowing for better parsing of user input.
  - **Prompt storage** The system stores both user and LLM messages in a database [found in ChatRepository.kt], allowing a user to revisit previous conversations and view the messages.
- **Contribution to Project Objectives** This functionality forms the foundation of the system's ability to understand user intent, ensuring that inputs are properly processed before being sent to the AI model. Similarly, it provides users with the opportunity to generate multiple kinds of prototypes and revisit them all at a later date for comparison.

3.2.3 Functional Requirements Extraction

- **Description** The system uses one LLM call to extract a comprehensive list of formal functional requirements from the (potentially broad and vague) user's prompt.
- **Implementation**
  - **"Requirements Prompt"** The system uses specialised system prompts to guide the LLM in generating well-structured requirements [found in PromptingTools.kt]
  - **Standardised Response Format**: Requirements are returned in a standardised JSON format for consistent processing.
- **Contribution to Project Objectives** This functionality bridges the gap between natural language user requests and formal software specifications, ensuring that the generated prototype addresses all user needs.
- **Possible Extension** One possible *extension* here would be to display the functional requirements to the end user, allowing them to tweak the extracted requirements for more direct interactions with the system. **This, however, was deemed out of scope for this project.**

3.2.4 Template Retrieval

- **Description** The system is able to use semantic and keyword search to find relevant templates for a given user's prompt. These templates are used to provide more context to the LLM for code generation without requiring a full agentic workflow.
- **Implementation**
  - **Templates** A set of templates is predefined in the application. Each template consists of a highly reusable snippet of code written using some of the technologies we support.
  - **Template Annotation** Each template has an accompanying JSON-LD annotation, which includes semantic meaning to the templates. These annotations are the foundation for semantic search.
  - **Python Microservice** A Flask-based microservice is used to handle search operations. It uses text embeddings and cosine similarity to find the most semantically similar templates, and keyword similarity to find the most technically similar templates.
  - **Internal Database** Results from the search operations are IDs, which are used as primary keys into our database to locate the actual template code to be presented to the LLM
- **Contribution to Project Objectives** This approach allows more consistent generation of prototypes by providing the LLM with extra guidance without the performance hit of a full agentic workflow . This allows users to be confident they are receiving a working prototype which fulfills their requirements.

3.2.5 Self-Expanding Template Library

- **Description** The system extracts reusable code patterns from LLM responses, annotates them with semantic metadata, and stores them in a template library. This creates a self-expanding knowledge base that improves future code generation by providing relevant context to the LLM without requiring full agentic workflows.
- **Implementation**
  - **Component Detection** The system extracts potential React component templates from LLM responses using regex pattern matching that identifies common component declaration patterns.
  - **Component Name Identification** Extracts component names using sophisticated regex patterns that recognize export declarations, function components, class components, and prop-based components.
  - **JSON-LD Generation** A second LLM call with a specialized prompt generates Schema.org-compliant JSON-LD annotations for each extracted template.
  - **Metadata Structure** Annotations include component name, description, functionality, programming language, framework information, application category, and relevant keywords.

- **Contribution to Project Objectives** This implementation significantly enhances code generation consistency by providing the LLM with contextually relevant, reusable components. The system continuously learns from successful code generation, expanding its template library over time without manual intervention. This creates a positive feedback loop where each successful prototype potentially contributes to making future prototypes more robust and feature-complete, ultimately improving user experience and satisfaction.

3.2.6 Lightweight proof-of-concept Generation

- **Description** The system generates functioning code prototypes based on the extracted requirements and the user's prompt.
- **Implementation**
  - **Generation Prompt Generation** The system uses specialised system prompts [found in PromptingTools.kt] for more guided and reliable responses from the LLM. This includes giving a specific JSON schema to the LLM which it must create code in accordance to, and including the functional requirements generated.
  - **Multi-technology Support** The system generates code for various frontend and backend technologies, including "React" and "Tailwind".
  - **Template Incorporation** The system leverages pre-defined templates stored in a vector database to help with code generation, using the 'Retrieval Augmented Generation' technique, where the LLM is given more context to help generate a more accurate response.
- **Contribution to Project Objectives** This functionality delivers on the core promise of the system: transforming natural language descriptions into functioning prototype applications.

3.2.7 Template Library Expansion

- **Description** The system is able to detect snippets of highly reusable code in the LLM's responses. These are converted into templates themselves, dynamically expanding the template library.
- **Implementation**
  - **Template Tagging** The LLM tags potential templates in its response.
  - **Parsing** Each potential template is analysed, and JSON-LD annotation for it is generated via another LLM call.
  - **Storage** New templates and annotations are stored and database references are updated. JSON-LD annotations are then integrated into the Python microservice.
- **Contribution to Project Objectives** This approach allows the system to scale automatically, with improved accuracy over time, without the performance hit of a

full agentic workflow . This allows users to be confident they are receiving a working prototype which fulfills their requirements.

3.2.8 Prototype Rendering

- **Description** The system renders generated prototypes directly in the browser for immediate testing and feedback.
- **Implementation**
  - **WebContainer Integration** Uses WebContainer technology to execute code directly in the browser.
  - **Live Preview** Shows immediate results of code changes.
- **Contribution to Project Objectives** This functionality delivers the most key goal of the project: users immediately interact with their generated prototypes without requiring additional setup or deployment steps.

3.2.9 Prototype Reliability

- **Description** The system dynamically attempts to fix code generated to be displayed to the user.
- **Implementation**
  - **Automatic Dependencies** Code generated from an LLM can contain incomplete dependencies, therefore the system looks for unmentioned dependencies and attempts to install them dynamically as the prototype is attempted to be rendered.
  - **Patch work** Code generated from an LLM can miss critical files which are required for the prototype to be displayed, therefore the system initially looks for these files to ensure they are present, and if not found tries to create files which can display the generated code.
  - **Vite Configuration** Code generated from an LLM can miss configuration files, therefore the system attempts to automatically create these in accordance to the version of 'React' used to be able to display the generated code.
- **Contribution to Project Objectives** This functionality supports the key objective of this project, displaying generated prototypes. With limitations of the reliability of LLM generated code, this functionality helps bridge gaps and ensure as many prototypes can be rendered as possible.

3.2.10 Prototype Storage and Retrieval

- **Description** The system persists generated prototypes for future access and reference.
- **Implementation**
  - **Database Storage** Stores prototype metadata and content in a structured database.
  - **Prototype Versioning** Maintains historical records of generated prototypes [found in ChatRepository.kt] linked with the message that generated them. Furthermore, these prototypes can be revisited on demand by clicking the corresponding message in a conversation.
- **Contribution to Project Objectives** This functionality enhances the system's utility by allowing users to create better prototypes with every generation.

3.2.11 Authentication and User Management

- **Description** The system provides user authentication to personalise experiences and secure access to features.
- **Implementation**
  - **Cognito Authentication** Leverages cloud athentication for maximum security and reliability.
  - **User-specific Content** Associates prototypes with specific users for personal history and access control. The personal history includes both the chat messages and their prototypes.
- **Contribution to Project Objectives** This functionality enables personalised experiences and secure access to system features.

3.3 Non-Functional Requirements

3.3.1 Security

- **Description** The system ensures that all generated code and user interactions are secure.
- **Implementation**
  - **Safe Execution Environment** WebContainers provide isolated execution environments.
  - **Input Sanitisation** All user inputs are sanitised to prevent injection attacks, this prevents the system from being breached as it runs code in the WebContainer which could perform undesirable actions.
- **Contribution to Project Objectives** Security is paramount as the system generates and executes code based on user inputs.

3.3.2 Performance

- **Description** The system responds to user requests within acceptable timeframes.
- **Implementation**
  - **Efficient API Calls** Optimised communication between client and server using JSON, reducing overheads.
  - **Responsive UI** The application remains responsive during code generation and rendering.
  - **Resource Management** LLM calls are split up to allow for more efficient generation depending on what is required to be generated.
- **Contribution to Project Objectives** Performance directly impacts user satisfaction and the practical utility of generated prototypes.

3.3.3 Usability

- **Description** The system provides an intuitive and accessible interface.
- **Implementation**
  - **Responsive Design** Adapts to different screen sizes and device types, including mobile devices.
  - **Clear Feedback** Provides clear indications of system status during processing in the WebContainer. Furthermore, error messages and pages are set up to ensure the user is well informed of the current status within the application.
  - **Intuitive Navigation** Logical organisation of features and content.
  - **Accessible features** Webpage designs and colours are optimised to provide the best visual support for a diverse audience.
- **Contribution to Project Objectives** Usability ensures that users can effectively interact with the system regardless of technical expertise, which is especially important as the intended audience of the application includes non-technical customers.

3.3.4 Scalability

- **Description** The system can handle increasing workloads by adapting resources efficiently.
- **Implementation**
  - **Multi-Module Design** The codebase is organized into specialized modules (auth, database, routes, utils, etc)
    - **Benefit** Enables independent scaling and maintenance of different system components
  - **Kotlin Coroutines** Extensive use throughout the codebase (particularly in PromptingMain and repositories).
    - **Benefit** Efficient resource utilization through non-blocking I/O, allowing more concurrent requests.
  - **Connection Pooling** Using HikariCP in DatabaseManager
    - **Benefit** Efficient resource utilization through non-blocking I/O, allowing more concurrent requests.
- **Contribution to Project Objectives** Scalability is crucial for this system as the system must maintain responsiveness under increasing user loads and as the template library grows, efficient data management becomes critical

3.3.5 Compatibility

- **Description** The system works consistently across different browsers and devices.
- **Implementation**
  - **Cross-Browser Testing** Accessible across major browsers.
  - **Device Compatibility** Functions on both desktop and mobile devices.
- **Contribution to Project Objectives** Compatibility ensures accessibility to a wide range of users regardless of their preferred platforms.

3.3.6 Reliability

- **Description** The system's generation of prototypes is consistent and meets the user's prompt
- **Implementation**
  - **Vector Database** The vector database allows for the LLM to consistently fall back on templates already used leading to consistent prototypes where possible.
  - **Prompt Engineering** Through the use of system prompts the LLM has more predictable and consistent results which can be tweaked and improved through iterations.
  - **JSON Schema** The use of a JSON schema ensures the code generated will match the desired output and be usable for the front end to display both the chat message and the prototype itself.
- **Contribution to Project Objectives** The application needs to have a base level of quality to ensure the results are useful to the end user, and the application has to be reliable so that an end user can expect to receive a product when using the service.

3.3.7 Accessibility

The web application was designed with the Web Content Accessibility Guidelines (WCAG) in mind to ensure comfort and usability by a wide range of users. While designing the pages, care was taken into ensuring all text and interactive elements meet the minimum contrast ratio requirements specified by WCAG 2.1 AA standards. Buttons and alerts have high-contrast color schemes to improve visibility. Moreover, the application also employs labels, icons and tooltips where appropriate.

The web app is also fully tab-navigable, enabling users to access all interactive elements—such as tabs, buttons, form fields, and links—using only a keyboard. Focus styles are clearly defined using texttfocus-visible and textttring utilities from Tailwind CSS to ensure that users can easily track their position within the interface. Logical tab order and consistent visual feedback enhance the experience for users relying on assistive technologies or keyboard navigation.

These accessibility features not only align with WCAG recommendations but also contribute to a more inclusive, user-friendly experience for all users.

3.4 Out of Scope (Possible Extensions)

3.4.1 Agentic Workflows

- **Description** The system would use agentic workflows for the prototype generation, allowing the LLM to iteratively refine prototypes through multi-step interactions.
- **Contribution to Project Objectives** An agentic workflow would remove the strain on the LLM to produce a single, perfect response, but rather continually refine a response with distributed processing across multiple agents. This flow would further enhance the reliability of prototype generation.
- **Constraint** The primary limitation is technical resources, as adding more LLM calls would be infeasible for us to develop, given the lack of computing power our team has available.

3.4.2 Requirement editing

- **Description** The system should allow users to edit the functional requirements that are generated upon sending the very first message in a conversation. This might include a pop-up after the first LLM call, where users are presented with a list of requirements that can be edited and passed on to the second LLM call. After this initial iteration, the requirements would be shown in a window above the chat, displaying a constant list for refinement discussions in the chat.
- **Contribution to Project Objectives** The ability to fine-tune any LLM generated functional requirements would reduce future iterations of prototype generation.

3.4.3 Multimodal file uploads

- **Description** The system should allow users to upload varying kinds of files along with their text prompt. This might include: business documents, images and mockups.
- **Contribution to Project Objectives** Multimodal file uploads allow users to further enhance the data provided to the LLM, which will create responses that are much more tailored to the user at hand.

3.4.4 Typing indicators

- **Description** The system should display typing indicators in the message box, while the LLM is generating the functional requirements and code in the back-end.
- **Contribution to Project Objectives** Typing indicators would ensure that the user is constantly aware of the status of the LLM, while helping to align the system with common user expectations and experiences.

Chapter 4

Project management

4.1 Introduction

Managing a team of ten individuals in a dynamic project environment required us to make strategic decisions to ensure motivation, maintain quality, and achieve timely delivery. Drawing on agile methodologies and principles of self-organizing teams, we adopted a sub-group approach. A designated project leader was responsible for overseeing all sub-groups and maintaining clear continuous communication with the client. This structure was chosen not only to streamline decision-making but also to foster an environment where team members could both leverage their strengths and learn from new challenges.

4.2 Task Structure

To effectively manage the project, we decomposed the work into moderately large, vertical task slices encompassing:

- **Back-end Development**
- **Front-end Development**
- **Testing**
- **Documentation**

Each task was assigned to a sub-group, with further internal distribution based on individual skill sets. This vertical slicing strategy ensured that every sub-group could focus deeply on a particular component, while also promoting cross-functional understanding. Such an approach aligns with agile task management principles and the Theory of Constraints, as it enabled us to quickly identify bottlenecks and address them in real time.

4.3 Sub-Group Formation and Collaboration

Sub-groups, typically consisting of 3 to 5 members, were formed through voluntary participation. This method encouraged team members to join groups where they felt most competent and motivated, leading to a natural distribution of skills and fostering peer-to-peer learning. For example, in a sub-group of three, two members might focus on back-end development while one handled front-end tasks, with additional responsibilities like testing and documentation shared as needed. This collaborative model not only facilitated parallel work streams—especially beneficial given our microservices-based architecture—but also minimized risks associated with single points of failure by preventing any one feature from being "owned" by a single individual.

However, members of the sub-groups would move around depending on the amount of backlog tasks where members would be allocated to sub-groups based on their experience and knowledge and less on their personal preference. This ensured the completion of the tasks in a timely manner.

4.4 Resource Constraints

The team encountered significant challenges due to critical resource constraints imposed by our clients and King's College London. Initially, our project was intended to leverage Amazon Bedrock, an advanced service for Large Language Model (LLM) integration, which was crucial for achieving the envisioned functionality. However, access to Amazon Bedrock was not granted, necessitating negotiations that ultimately led the client to approve the local deployment of models instead. While this compromise allowed progress, it simultaneously introduced substantial additional challenges.

LLMs inherently require extensive computational resources, and the absence of adequate infrastructure severely restricted our ability to deploy suitable models. Consequently, we resorted to utilizing smaller, locally runnable LLMs for tasks such as code generation. Unfortunately, these smaller models were neither optimized nor specifically trained for our project's specialized needs, leading to inconsistent and unreliable outputs. Specifically, we experienced frequent violations of the defined System Prompt instructions, incorrect character escaping, and unauthorized use of technologies incompatible with WebContainers. These issues directly contributed to significant technical debt, considerable delays, and ultimately impacted the overall refinement and polish of various application components.

The challenges described above should not be interpreted as deficiencies in the team's management approach or strategic planning. Rather, these were direct outcomes of the imposed resource limitations. The team confidently asserts that, given identical timelines, skill levels, and team availability, but with proper access to essential resources, the project would have achieved a significantly higher level of quality and reliability.

We kindly request that the marking team take these substantial constraints into account when assessing our project. Despite being compelled to operate under highly restrictive conditions, we believe we have demonstrated considerable adaptability and achieved commendable success. Crucially, all functionalities of the application, including the Proof-of-Concept (PoC) generation workflow, are operational, contingent upon adequate responses from the LLM.

With appropriate resources, specifically unrestricted access to Amazon Bedrock, we could have conducted comprehensive and consistent testing across the team, eliminating prolonged "under review" periods. Additionally, access to more powerful and specialized models would have enabled us to implement a robust agentic workflow using MetaGPTs integrated with the AFlow algorithm, ensuring consistently high-quality responses. Furthermore, this would have facilitated real-time PoC generation, significantly enhancing the application's usability and responsiveness.

4.5 Meetings and Communication

Two internal mandatory meetings (Mondays and Thursdays) were set each week with the team, while meeting with the client was set, more or less, every two weeks, based on their availability.

The first internal meeting, the brainstorming session, was generally for task allocation, progress updates from each of the members or sub-groups, and discussion of future feature implementations. While the second internal meeting, the cleanup session, was for code review, ensuring each member stuck to the coding standards as discussed before the start of the project and that necessary tests were implemented. And the client meeting involved showcasing the promised deliverables as discussed two weeks prior, receiving feedback from the client, clearing up any ambiguity with the project, and setting new deliverables for the next two weeks.

The internal meetings were held on a face-to-face first and then virtual second basis. Because in-person meetings creates an environment where communication between team members are clearer, more interactive, and more focused, as it encourages members to engage in the meetings. While meetings with the client were held entirely virtual.

Furthermore, meetings or asynchronous discussion between sub-groups were encouraged, as to prevent people from interfering with each others' work and to prevent 'surprises' when somebody implements a feature, or contributes code to a dead feature, that was not made known to everybody working on a particular task. This prevents wasted effort.

4.6 Critical Reflections and Lessons Learned

While our systematic approach generally produced high-quality code, robust tests, and comprehensive documentation, the project also presented several challenges that have provided valuable learning opportunities:

- **Over-Specialisation:** The vertical task structure sometimes led to over-specialisation. In instances where key members were unavailable, the sub-group faced bottlenecks. In future projects, incorporating structured cross-training sessions could mitigate this risk.
- **Integration Complexities:** Despite the modularity inherent in our sub-group and microservices approach, we experienced occasional integration challenges. These issues highlight the need for more frequent integration checkpoints and stronger inter-group communication.
- **Evolving Leadership Demands:** Initially, the project leader's role was focused on high-level oversight. However, as the project evolved, the role increasingly required hands-on problem solving. This evolution suggests that a more adaptive leadership

model—potentially including rotating leadership roles, or multiple leaders rather than one—might be more effective in managing similar projects in the future.

- **Miscommunication:** Despite our early philosophy of meetings or asynchronous communication within sub-groups, there were still occasional misunderstanding which left some people with outdated information regarding the task, or a sense of confusion regarding how to proceed. This could be mitigated by asking questions when in the slightest doubt, and double checking with everyone else before proceeding with implementations of features via messaging channels, so to have a reliable record of the past.
- **Sub-task dependencies:** Even with the decomposition project tasks into different categories (back-end, front-end, etc) and division of tasks within the sub-group, i.e. the sub-tasks, there were some instances within the sub-group where sub-tasks were inherently coupled, this caused bottlenecks when the task that was depended on was not complete in a timely manner. A possible mitigation would be an depth discussion among the peers within the sub-group on how to go about implementing the feature. This could highlight possible coupled sub-tasks, thus avoiding this issue. However, sometimes coupling could only be realised once coding has begun, so in these situations, the members could swap sub-tasks with each other, giving the most critical task to the person who can work on it quickly or reconvene and agree to another way to split the task.
- **Interpersonal Relations and Feedback Culture:** As a group composed of individuals who were already on good terms with each other, we often hesitated to provide direct critiques on each other's work to avoid potential discomfort. While this helped maintain a harmonious working environment, it also slowed down development and sometimes led to suboptimal results due to unaddressed issues. Additionally, the accumulation of unspoken frustrations led to moments of annoyance, which in some cases subtly soured relationships within the group. A key lesson from this experience is the importance of fostering a culture where constructive feedback is encouraged and accepted as a tool for improvement rather than personal criticism. Establishing clear expectations around feedback, perhaps through structured review sessions or anonymous peer evaluations, could help mitigate hesitations and create a more open and productive working dynamic while preventing the buildup of interpersonal strain.
- **Time Management and Workload Distribution:** Some tasks took longer than expected due to unforeseen complexities, underestimation of effort, or dependencies on other workstreams. This occasionally led to last-minute rushes, affecting code quality and increasing stress among team members. Additionally, uneven distribution of work resulted in some members being overloaded while others had less to do. Future projects could benefit from more realistic time estimations that take potential obstacles into account, the incorporation of buffer periods to accommodate unexpected delays, and periodic workload assessments to ensure an even and manageable task distribution. More frequent check-ins and progress tracking could also help identify bottlenecks earlier, allowing the team to reallocate resources as needed.
- **Documentation Challenges:** While documentation was emphasized as a crucial part of the project, there were instances where it was incomplete, inconsistent, or not

updated in a timely manner. This made it difficult for team members to understand the latest developments and onboard new contributors efficiently. Additionally, a lack of standardization in how documentation was written led to inconsistencies that sometimes caused confusion. To improve this, future projects could implement a structured documentation strategy, requiring regular updates after major feature implementations. Establishing clear guidelines and templates for documentation could also enhance clarity and consistency, making it easier for all team members to reference and contribute to the project effectively.

Our reflections were informed by established frameworks such as the PMBOK guidelines and agile retrospectives, which underscored the importance of iterative improvement and systematic risk management.

4.7 Conclusion

In summary, our project management approach, anchored in agile theory and characterised by a flexible sub-group structure, vertical task slicing, and adaptive leadership, proved effective in meeting our project goals. Nevertheless, critical evaluation revealed areas for improvement, particularly in balancing specialisation with cross-functional capability and enhancing integration practices. The lessons learned from this project will serve as a foundation for refining our methodologies in future endeavours.

Chapter 5

Design and implementation

5.1 Architecture

The architecture of the system is delineated into two distinct applications: the **client** and the **server**.

The **client application** provides the primary interface for user interaction. It is responsible for presenting the user interface, including interactive components and visualisation of generated prototypes.

In contrast, the **server application** is not directly accessible by end-users and exclusively handles interactions mediated through the client. The server's responsibilities encompass user authentication via integration with Amazon Cognito, processing incoming requests (including prompt engineering and template retrieval), and managing the delegation of tasks related to proof-of-concept (PoC) generation to the appropriate subsystems. Furthermore, the server coordinates integration with Ollama for generative tasks, handling computationally intensive operations, and manages user session states securely.

Communication between the client and server occurs exclusively via defined APIs, ensuring robust separation of concerns and encapsulation.

Moreover, the architecture incorporates Amazon Cognito for secure user authentication, leveraging OAuth2 and JWT standards to maintain session security and integrity. Session management utilizes HTTP-only and secure cookies, reinforcing the application's security posture.

Finally, specialised operations, such as semantic embedding of templates and retrieval via similarity searches, are delegated to separate Python-based components. These functions utilise advanced techniques, including semantic embeddings and vectorized search methodologies, enhancing the system's responsiveness and accuracy in prototype generation tasks.

5.2 The Client Application

5.2.1 Overview

The client application is developed using React with TypeScript and JavaScript, and built with Vite. This solution provides us with native ESM development server, hot reload, lazy compilation and a Rollup-based production bundler, which contributes to a faster and more efficient development workflow. TailwindCSS was chosen for its utility-first CSS approach, which ensures consistent styling and reduces the need for custom CSS files.

These components are logically organised within the codebase under the `components` directory, categorised by function (e.g., general UI components and function-specific components). This component-based structure improves code discoverability as well as ensuring scalability and maintainability. The ui components employ the ShadCN UI component library, which provides a cohesive design language as well as a modern and visually appealing interface.

5.2.2 Internal High-Level Architecture

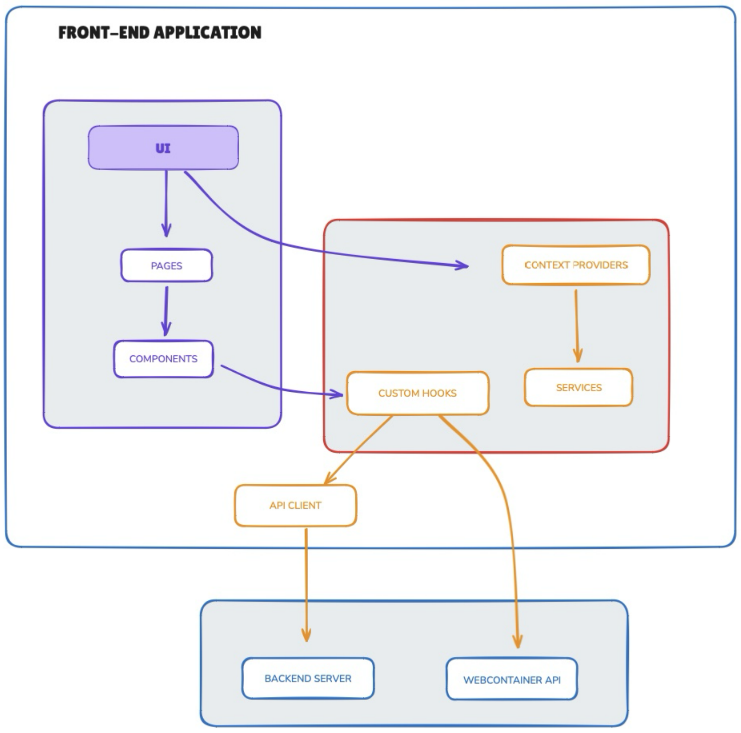


Figure 5.1: Architectural diagram of frontend application

The application follows a modular, component-based architecture, with a clear separation between functional domains. The client application intends to fulfill the following functional requirements: receiving requirements, editing requirements, viewing the generated prototype. This structure is reflected in two main user-facing pages: the Landing Page and the Generate Page.

The Landing page (`/`) serves as the user's first point of interaction and is responsible for capturing initial requirements. It includes the `HeroSection`, `InputBox`, `OldPrompts` and `GeneratedPrompts` components.

The Generate page (`/generate`) acts as the root view for the main interactive experience which encompasses editing requirements and prototype viewing. It contains the following:

- `Chat` module which includes the `ChatScreen`, `ChatBox`, `MessagesBox`, and `MessageBubble` components
- `Prototype` which includes the `PrototypeFrame` and `PrototypeWindow`
- and `Sidebar` module which includes `SidebarWrapper`, `AppSidebar`, `NavMain`, and `NavUser` components.

React Router is utilised for efficient client-side navigation which enables seamless transitions between the different views. It enables the application to function as a Single Page Application (SPA), ie; the user can navigate between routes without triggering full page reloads. This is particularly beneficial in a chat context, where maintaining state such as message history, user input, or prototype previews is needed for a smooth user experience. Programmatic navigation is also supported through hooks like `useNavigate()`, which are used within interactive components (e.g., clicking the logo to return to the homepage). To further enhance user experience and prevent application crashes, React Error Boundary is implemented for handling runtime errors elegantly.

5.2.3 Architectural Pattern

The internal design aligns with the Model–View–Controller (MVC) pattern, promoting separation of concerns across the system:

- **Model (M)**: manages data
  - `AuthContext`: Manages authentication state and exposes login/logout functionality to child components through React Context.
  - `ConversationContext`: Manages the full lifecycle of user conversations, including fetching historical data, managing the active conversation, and updating metadata. This context is consumed across the chat and prototype modules to ensure synchronised state.
- **View (V)**: renders data and user interface
  - `MessagesBox` and `MessageBubble`: Display the conversational history and individual chat messages, respectively.
  - `PrototypeFrame` amd `PrototypeWindow`: Renders the output of the generated prototype in a structured format.
  - Other presentational components like `HeroSection`, `Sidebar`, and prompt display modules also fall within the view layer.
- **Controller (C)**: handles user input and updates the model.
  - `ChatMessage` Hook: A custom hook that controls chat interactions, manages message state, sends and retrieves data from the backend, and coordinates

prototype generation. It acts as the intermediary between user input (via `ChatBox`) and the underlying model (`ConversationContext`), reflecting controller responsibilities within the architecture.

  – `usePrototypeFrame` and `useWebContainer` Hooks: Custom hooks that that manage the lifecycle of a WebContainer instance, including mounting prototype files, listening for server readiness, and exposing a reset function. They act as the controller between the backend prototype responses and the UI component (`PrototypeFrame`). This aligns with the controller role of abstracting side effects and providing a clean interface for the view layer.

This MVC-driven internal architecture ensures that logic, presentation, and data responsibilities are cleanly separated, facilitating a more maintainable, scalable, and testable client application. This architecture supports the core React principle of unidirectional data flow as data flows from Model to View and updates are triggered by user interaction, which flow back from View to Controller to Model.

The frontend interacts with the backend through API requests, allowing real-time updates and data retrieval. The combination of these technologies ensures a scalable, maintainable, and user-friendly application.

5.2.4 WebContainers

The primary functionality of our application necessitates the delivery of functioning, interactive prototypes to end-users. This requirement presents the challenge of rendering and displaying code generated by the Large Language Model (LLM). While prompt engineering and template retrieval are managed by the server-side components, the client application must effectively render and display the generated prototype code.

Several technical considerations informed our approach to this challenge. These include the secure execution of AI-generated code, determining the optimal location for prototype rendering (server-side versus client-side), and implementation methodology. After comprehensive analysis of available solutions, WebContainers emerged as the optimal technology, providing the most advantageous balance between our non-functional requirements of security, compatibility, and scalability.

5.2.4.1 Available Approaches

The challenge of rendering arbitrary code and delivering it to users has been extensively researched and developed within the industry, particularly by organisations offering web-based integrated development environments (IDEs) such as StackBlitz and CodeSandbox. These entities have developed proprietary technologies, WebContainers and Nodebox respectively, to address similar requirements.

Historically, web IDE implementations would instantiate lightweight web servers on their own infrastructure (typically utilising virtual machines), bundle the application, and transmit the rendered code to the client's browser. This methodology is documented in CodeSandbox's technical literature, which elucidates their performance optimisation strategy through Firecracker VMs (https://codesandbox.io/blog/how-we-clone-a-running-vm-in-2

5.2.4.2 Server vs. Client Rendering

Our architectural decision process concerning prototype rendering location was primarily guided by considerations of compatibility and scalability. Given that code is generated by an LLM, we cannot predetermine which libraries, frameworks, or features will be utilised in the output.

Server-side rendering offers robust compatibility assurances, as server environments typically provide modern, frequently-updated computational resources. These environments also deliver superior performance in application rendering, being engineered for concurrent request processing. However, server-side solutions present significant limitations: they are ephemeral, incur substantial scaling costs, pose security challenges, and are susceptible to network latency and initialisation delays. Furthermore, project resource constraints precluded the server-side approach, as computational resource limitations would be rapidly encountered.

Analysis revealed that the client's browser represents the optimal environment for executing arbitrary code. This approach offers several advantages: it is secure, persistent, theoretically infinitely scalable, cost-effective, and provides rapid initialisation, thus enhancing user experience. Client-side execution reduces potential attack surfaces and enables offline functionality of generated web applications, a valuable capability in bandwidth-constrained environments. The primary limitation of this approach concerns compatibility, as both the rendering technology and frameworks employed in generated code must be supported by the user's browser.

Nodebox demonstrates comprehensive browser compatibility, albeit with performance disadvantages compared to WebContainers and continued dependence on backend servers for certain execution aspects, introducing latency. Conversely, WebContainers present a more suitable solution for our use case, implementing a complete Node.js server within the browser via WebAssembly (WASM). This implementation eliminates server dependencies, thereby removing network round-trip latency. According to StackBlitz documentation, WebContainers can achieve performance superior to localhost environments. Additionally, they provide a more developer-friendly API. The primary limitation of WebContainers is reduced browser compatibility compared to Nodebox, due to their utilisation of SharedArrayBuffer, an experimental JavaScript feature in Safari. This limitation extends to Private Windows in Firefox, although support is robust across modern Chromium-based browsers. [1]

Following comprehensive evaluation, WebContainers was selected as the implementation technology, as the client indicated that support for mobile and older browsers was not a priority requirement, and the solution satisfied all other technical and functional criteria.

5.3 The Server Application

5.3.1 Overview

The server component has been developed in Kotlin, employing the Ktor framework for robust API implementation. This combination capitalises on the Java Virtual Machine's

[1] https://webcontainers.io/guides/browser-support

(JVM) efficient runtime environment and leverages Kotlin's coroutine-based concurrency model, thereby ensuring scalability and high performance in handling concurrent user interactions.

The system architecture adheres to modular design principles, comprising distinct modules that encapsulate specific functionality. Each module is designed to offer clear interfaces, enabling seamless interaction and reuse across the application. Notably, the `authentication` (`auth`) module provides comprehensive user authentication and authorisation via Amazon Cognito, whereas the `prompting` module coordinates communication between the `chat` module (responsible for processing user requests) and the proof-of-concept (PoC) generation workflow, ensuring efficient orchestration of tasks within the prototype generation process.

5.3.2 Internal High-Level Architecture

The high-level architecture of the server application is illustrated in Figure 5.2. The diagram highlights the primary components and interactions within the system.

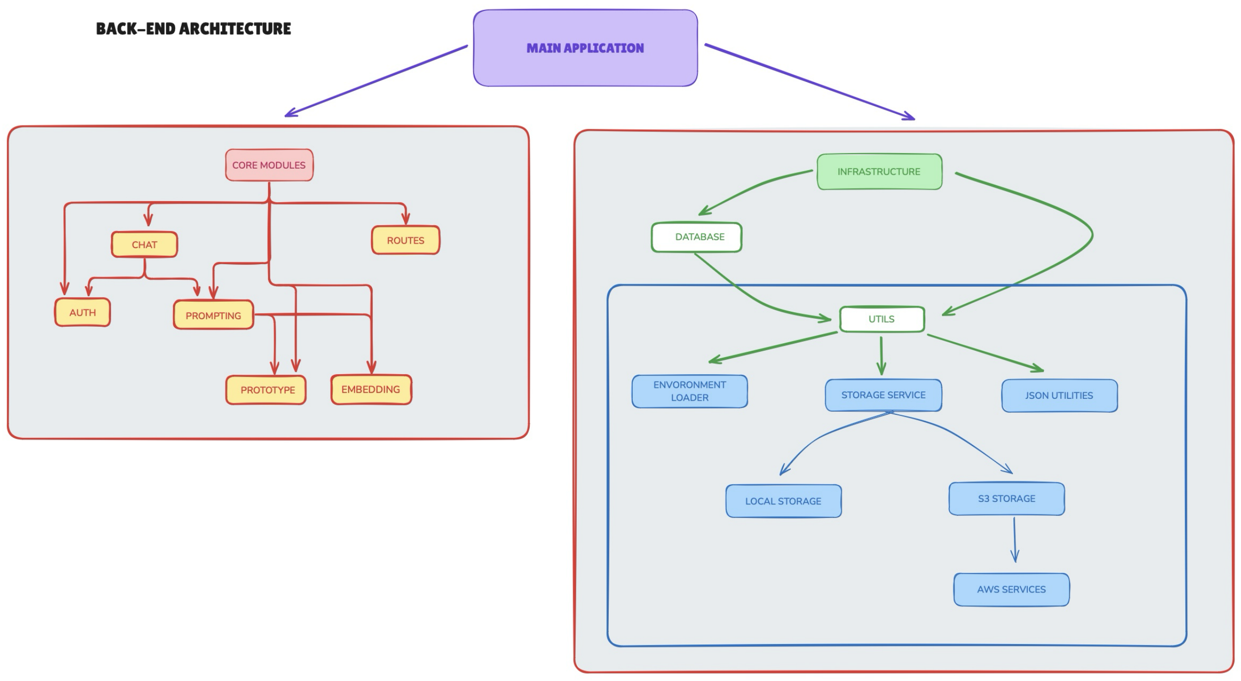


Figure 5.2: High-level Architecture of the Server Application

As shown in Figure 5.2, the architecture comprises distinct modules, adhering to the microservices architectural paradigm, promoting modularity and separation of concerns. The system enforces strict encapsulation, ensuring that interactions occur exclusively through predefined interfaces and are limited solely to the client application. Direct user interaction with server components is explicitly disallowed, preserving security and modular integrity.

User interactions with the server fall into two categories:

- **Authentication Requests**: These requests are routed directly to the dedicated authentication module, which handles all authentication and authorization processes.

Further technical specifics of this module are detailed in Section **5.3.3**.

- **Prompt Submission Events**: Handled by the chat module, these interactions trigger the core Proof-of-Concept (PoC) generation workflow. Upon receiving a user's prompt, the chat module initiates a sequence of interactions between various internal modules, abstracting the complexity of prompt engineering, template retrieval, and delegation to external services (e.g., Ollama). The user receives a fully processed response without needing awareness of underlying server operations or inter-module communications.

This architectural design simplifies client-side implementation by ensuring that clients rely solely on a stable API contract provided by the server, reinforcing a clear separation of concerns and promoting maintainability. Further elaboration on the specific mechanisms of authentication and internal request handling is detailed in subsequent sections.

5.3.3 Authentication Flow: An Example

The authentication module, depicted in Figure 5.3, provides a concrete example of internal module interactions within the server.

The authentication workflow begins when the client sends a request containing an endpoint within the server API. Upon receiving this request, the authentication module searches for the presence of an `AuthenticatedSession` cookie, which signifies an existing authenticated session. If this cookie is not found, the user is considered unauthenticated and is redirected to initiate the OAuth 2.0 authentication flow with Amazon Cognito.

This redirection first takes users to the `/api/auth` endpoint, initiating the authentication process. Successful completion of the authentication sequence results in Amazon Cognito providing a JWT token, from which an `AuthenticatedSession` cookie is generated, securely containing the user's ID, token, and permission level (e.g., admin privileges). This cookie is designated as secure and HTTP-only, preventing visibility and manipulation via JavaScript, thus mitigating potential vulnerabilities.

In scenarios where the cookie is already present, the authentication module verifies its validity through the `JWTValidator` object. This entity communicates with Amazon Cognito, retrieves the signing key via JSON Web Key Sets (JWKS), and ensures the JWT token includes mandatory claims (such as the user's ID), verifies its expiration, and validates its cryptographic signature. Only tokens passing these checks are considered valid, maintaining stringent security.

The client application responds to the server's authentication results by updating the user interface appropriately: indicating successful authentication through UI adjustments such as displaying hidden content and changing the *Sign In* button to *Log Out*, or presenting custom error notifications in case of authentication failure.

5.3.4 Proof-of-Concept Generation Workflow

The process of generating a proof-of-concept (PoC) prototype involves considerable complexity due to the multiple potential interaction strategies with large language models (LLMs) required to achieve optimal results. Our implementation faces additional constraints arising from limited access to these models and the substantial computational resources necessary for operating large-scale LLMs, far exceeding typical personal computing capacities. Consequently, we developed a workflow that emulates agent-based methodologies while significantly minimising the number of direct calls to the LLM by positioning the server itself to function analogously to an agent.

Instead of utilising multiple LLMs interacting through a dedicated orchestrator model, our approach configures the server to simulate a model by means of predefined templates presented to the LLM. The full workflow is visualized in Figure 5.4.

The workflow initiates when a user submits a prompt to the chat interface via the client application, which targets the endpoint `/api/chat/json`. The `chat` module, serving as an entry point, solely accepts user prompts, invokes the subsequent workflow stages, and ultimately transmits the generated response back to the client.

The process continues through the `prompting` module, which fulfills the orchestrator role within this architecture. Initially, the `prompting` module applies prompt engineering techniques to the user's input, partly involving an LLM call designed to identify functional requirements and pertinent keywords from potentially non-technical initial prompts. Leveraging these extracted requirements along with the original input, the system generates an intermediate prompt for template retrieval.

Template retrieval is conducted by the `embeddings` module, which forwards the processed information to a dedicated Python service responsible for executing both semantic searches (employing embeddings of the intermediate prompt) and keyword searches. This service returns identifiers for the top ten most relevant templates. Subsequently, the `prompting` module queries the internal database, via the `database` module's ORM (DAO notation used throughout), to obtain file paths corresponding to these template identifiers and retrieves the associated template code.

Next, the `prompting` module constructs a comprehensive system prompt that integrates explicit instructions for the LLM, the engineered requirements, the original user prompt, and the retrieved templates. A subsequent LLM call with this system prompt generates the PoC code.

The LLM responds with JSON-formatted content adhering to a predefined structure, facilitating straightforward parsing by the server. The server separates the response into two main components: the textual segment displayed as a conversational message within the client's chat interface and the generated code, which is directed to a WebContainer for visual rendering.

Additionally, the workflow incorporates mechanisms supporting the dynamic expansion of our template library. Whenever the LLM identifies a segment of the generated code as a candidate template, an additional LLM call is executed to produce a JSON-LD annotation describing the template. Both the template code and its corresponding JSON-LD annotation are stored in dedicated files, references are updated in the database, and the JSON-LD annotation is further integrated into the vector database of the `embeddings` module, enhancing capabilities for semantic and keyword-based searches.

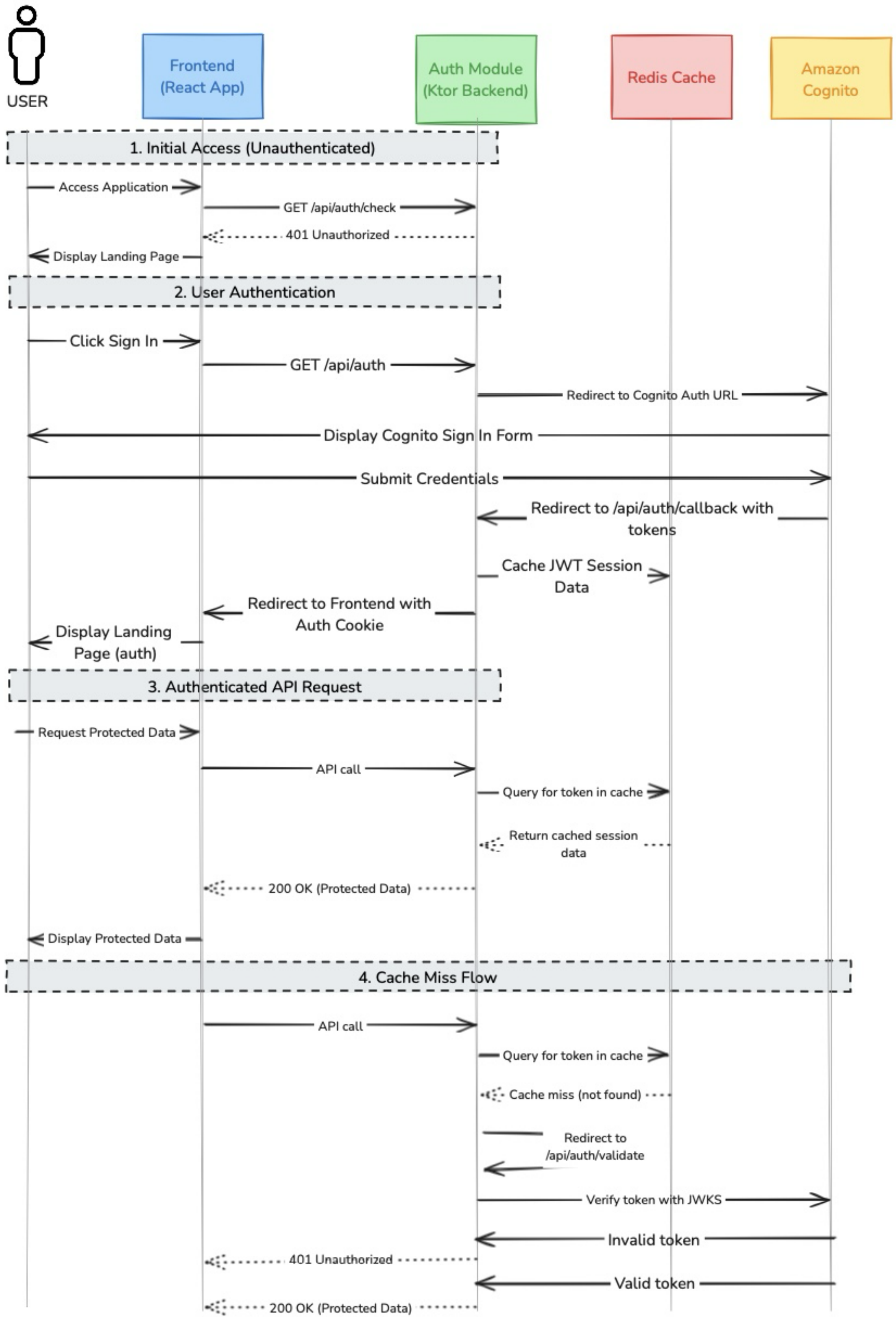


Figure 5.3: Authentication Module v1.0

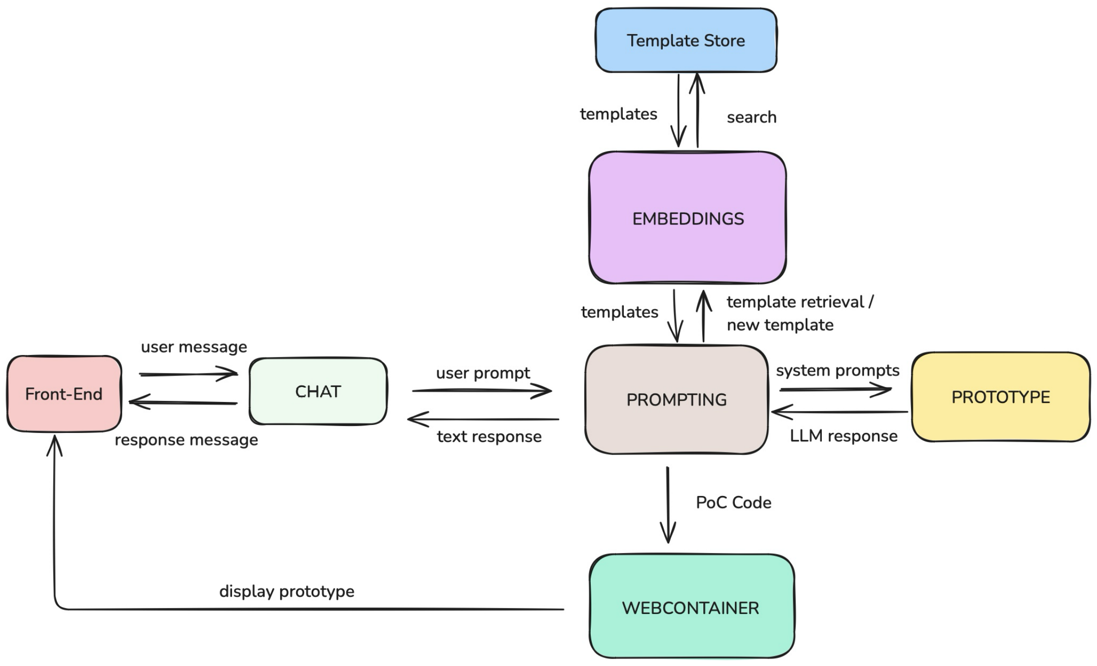


Figure 5.4: Prototype Generation Workflow v1.0

Chapter 6

Testing

6.1 Approach and tools

Throughout the software development lifecycle, rigorous testing methodologies were employed to ensure the verification and validation of the software under development. Additionally, testing served as a guiding framework for the development process, providing a shared understanding among team members regarding the functional and non-functional requirements of the application.

A combination of automated and manual testing strategies was utilised to maximise coverage and reliability. Automated testing was implemented wherever feasible to validate the correctness of individual software components and ensure adherence to expected behavior. In contrast, manual testing was conducted to assess the usability, consistency, and overall user experience, ensuring an intuitive and visually coherent interface.

Several automated testing frameworks were integrated into the development pipeline. The Kotlin server leveraged Ktor's testing framework for unit and integration testing, allowing for comprehensive validation of server-side functionalities. The Python microservice was tested using Pytest. To analyse code coverage and maintain high testing standards, three distinct tools were employed: **JetBrains' Coverage tool**, **JaCoCo**, and **SonarQube**. Each tool provides unique insights into test coverage, maintainability issues, code duplication, and potential vulnerabilities, justifying their combined usage.

For the frontend, the **Vitest** framework was used to conduct unit tests on React components, ensuring that the rendering logic met the expected output under various conditions.

Furthermore, functional and end-to-end (E2E) testing was implemented using **Playwright**, enabling automated validation of user interactions across different workflows. These tests simulate real-world user behavior within the application, ensuring that the system operates correctly under various scenarios and edge cases.

By integrating these diverse testing methodologies, the development process was systematically structured to enhance software reliability, maintainability, and user satisfaction.

6.2 Quality assurance processes

The development team implemented a comprehensive quality assurance (QA) process to ensure the delivered software met all functional and non-functional requirements while maintaining high standards of reliability, performance, and usability. This section details our systematic approach to quality management throughout the project lifecycle.

6.2.0.1 QA Strategy and Methodology

Our quality assurance strategy was founded on a proactive, prevention-oriented approach rather than focusing solely on defect detection. We adopted a modified Agile QA methodology that integrated quality considerations at every stage of development. This involved continuous validation against requirements and regular quality checkpoints throughout each sprint.

The team followed a risk-based testing approach, which prioritised test efforts on components with:

- Higher technical complexity
- Greater business impact
- More frequent historical defect rates
- Significant changes from previous iterations

This strategic allocation of testing resources enabled us to maximise defect detection in critical areas while maintaining efficient use of project resources.

6.2.0.2 Quality Standards and Compliance

The project adhered to industry-standard quality frameworks and internal standards including:

- ISO/IEC 25010:2011 for software quality characteristics
- OWASP security guidelines for web applications
- Accessibility compliance with WCAG 2.1 Level AA standards

Compliance with these standards was verified through dedicated testing activities and automated code analysis.

6.2.0.3 QA Activities Throughout the Development Lifecycle

Quality assurance was integrated throughout all phases of the development lifecycle.

Requirements Phase: During requirements analysis, each requirement underwent validation against our SMART criteria (Specific, Measurable, Achievable, Relevant, Time-bound) to ensure it could be effectively tested.

Design Phase: Design reviews incorporated quality considerations through formal inspections and architecture evaluation against quality attributes. Testability was explicitly considered during architectural decisions, resulting in design modifications to improve observability and controllability of system components.

Implementation Phase: During implementation, quality was assured through several mechanisms:

- Pair programming for complex components
- Code reviews(Appendix **??**)
- Static code analysis with SonarQube
- Continuous integration with automated unit and integration tests

Testing Phase: Our testing activities followed a multi-layered approach:

- **Unit Testing**: Verified individual components in isolation
- **Integration Testing**: Validated interactions between components
- **System Testing**: Evaluated the complete system against functional requirements
- **Acceptance Testing**: Validated the system against business requirements with stakeholders

6.2.0.4 Defect Management Process

The project implemented a structured defect management process, as described below.

All identified defects were:

1. Reported in our Discord "bugs" channel.
2. Classified by severity and priority (as per Table 6.1.)
3. Assigned to responsible developers with target resolution deadlines
4. Verified after resolution by the original reporter
5. Analysed for root causes to prevent recurrence

Severity	**Definition**
Critical	System crash, data corruption, security breach, or complete failure of core functionality
Major	Significant impact on functionality, but with workarounds available
Minor	Limited impact on non-core functionality or cosmetic issues
Trivial	Minimal impact, often related to documentation or visual consistency

Table 6.1: Defect severity classification

6.2.0.5 Quality Gates

The project implemented quality gates at key development milestones, with specific entry and exit criteria. Table 6.2 summarises these quality gates.

Quality Gate	Release Criteria
Sprint Completion	• All unit tests pass (100%) • Code coverage $> 95\%$ • No critical or major defects • Static analysis shows no critical issues
System Test Entry	• All integration tests pass • Documentation complete and reviewed • Feature completeness verified

Table 6.2: Quality gates and release criteria

These quality gates ensured that no deliverable proceeded to the next phase without meeting minimum quality standards, preventing the accumulation of technical debt and quality issues.

6.2.0.6 QA Tools and Infrastructure

Our quality assurance process was supported by an integrated toolchain that facilitated test management, execution, and reporting:

- **Defect Tracking**: Dedicated Discord channel
- **Automation Framework**: Playwright
- **Unit Testing**: JUnit/Pytest and Vitest for backend and frontend testing respectively
- **Continuous Integration**: GitHub Actions
- **Code Analysis**: SonarQube for static code analysis and code quality metrics

This infrastructure enabled continuous quality monitoring and rapid feedback on quality issues.

6.2.0.7 Continuous Improvement Mechanisms

The QA process itself was subject to ongoing improvement through:

- Bi-weekly retrospectives focused on quality processes
- Root cause analysis of significant defects
- Regular review and refinement of test cases

- Automation of repetitive testing activities
- Knowledge sharing sessions for quality best practices

6.2.0.8 Challenges and Mitigations

Despite our robust QA approach, several challenges were encountered during the project lifecycle:

Challenge 1: Test Environment Stability Functional tests with Playwright were exceptionally tricky to stabilise. This was mostly due to the integration with Amazon Cognito, which led to inconsistent test results. This caused confusion and frustration within the team.

Mitigation: We reduced the scope of functional tests to what was stable, primarily focusing on our application rather than integration with 3rd-party services.

Challenge 2: Test Data Management Complex test scenarios required realistic test data that was difficult to generate and maintain.

Mitigation: A dedicated test data management solution was developed, enabling on-demand generation of synthetic test data with appropriate relationships and constraints.

Challenge 3: Regression Testing Scope As the application grew, regression testing cycles lengthened, threatening to impact sprint velocity.

Mitigation: We implemented risk-based regression test selection and increased automation coverage, reducing regression test cycles while maintaining detection effectiveness.

6.3 Evaluation of Testing Methodology

The development team implemented a comprehensive test-centered approach throughout the project lifecycle. While not adhering strictly to traditional Test-Driven Development (TDD) methodologies, testing remained a core priority within our development framework. Our systematic testing strategy established an ambitious target of 100% code coverage for each sprint, thereby ensuring that subsequent feature implementations would not compromise existing functionality.

6.3.1 Strategic Testing Approach

Our testing philosophy emphasised continuous validation through iterative development cycles. When implementation changes rendered existing tests obsolete, significant effort was dedicated to revising test suites to accurately reflect the current codebase architecture. This disciplined approach maintained the integrity of our testing environment, ensuring all tests remained functional and coverage metrics consistently met our established thresholds.

6.3.2 Testing Hierarchy Implementation

The team strategically prioritised unit and integration testing methodologies for several critical reasons:

- **Granular diagnostic capability**: These testing methodologies provided fine-grained analysis, facilitating precise identification of failure points.
- **Execution efficiency**: Lower-level tests exhibited substantially faster execution cycles, enabling more frequent validation iterations.
- **Direct coverage correlation**: These methodologies demonstrated the most immediate and quantifiable impact on code coverage metrics.

This granular testing architecture significantly enhanced our ability to isolate failure points, often narrowing the diagnostic scope to specific functional components. Analysis revealed that most test failures did not result from direct implementation breakages but rather from architectural refactoring designed to accommodate new feature integration.

6.3.3 End-to-End Testing Considerations

While end-to-end (E2E) testing remained a secondary objective with limited implementation scope, this decision reflected deliberate architectural considerations rather than oversight. Two significant challenges constrained our E2E testing implementation:

1. **Authentication complexity**: Integration with Amazon Cognito for authentication services introduced testing inconsistencies, as test outcomes frequently depended on Cognito's session management protocols rather than our implementation logic.
2. **Resource constraints**: Triggering prototype generation within E2E testing environments would have imposed substantial performance overhead, significantly de-

grading test execution efficiency to the point where the diagnostic value was outweighed by operational costs.

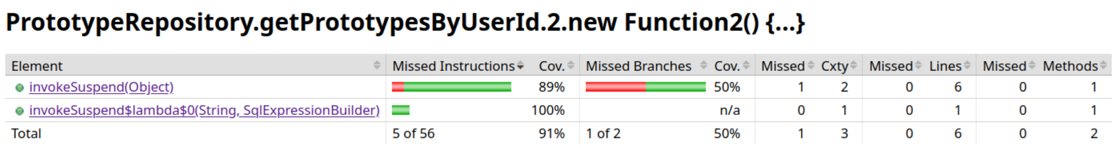
6.3.4 Balanced Testing Ecosystem

Despite these technical constraints, the development team successfully implemented targeted E2E testing focused on critical user interface validation, ensuring proper page rendering and verifying that core user interaction pathways functioned as specified. This balanced approach to testing, focusing on unit and integration tests while strategically implementing E2E validation, provided comprehensive quality assurance throughout the development lifecycle.

6.3.5 Tooling Limitations

In backend testing, our process was impacted by a known limitation inherent to JVM-based coverage analysis tools. As our backend is developed using Kotlin, the code undergoes compilation into JVM bytecode, which is subsequently analysed by these coverage tools. However, Kotlin's compilation mechanics differ significantly from other JVM languages, introducing features such as inlined lambdas, coroutine invocations, and companion objects. Although these elements are not explicitly written as part of our business logic, they appear embedded within the compiled bytecode. Consequently, coverage tools frequently misidentify these internal compiler-generated structures as untested code.

Importantly, these segments of code fall outside the testing scope, as they represent library-generated constructs and auto-generated stubs required solely for supporting compiler-specific operations rather than reflecting the application's core functionality. For instance, the figures below, obtained using **JaCoCo**, illustrate the invocation of a coroutine event through the `invokeSuspend()` function, which coverage tools incorrectly flag as partially untested. This occurs primarily because internal code paths within coroutine setup include safeguards for exceptional conditions, such as thread unavailability. Conversely, the anonymous lambda passed into the coroutine, representing our actual business logic, has been fully tested.



PrototypeRepository.getPrototypesByUserId.2.new Function2() {...}

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
invokeSuspend(Object)		89%		50%	1	2	0	6	0	1
invokeSuspend$lambda$0(String, SqlExpressionBuilder)		100%		n/a	0	1	0	1	0	1
Total	5 of 56	91%	1 of 2	50%	1	3	0	6	0	2

Figure 6.1: Coverage Example to illustrate coverage tool limitation

Another important consideration when evaluating test coverage metrics for Kotlin codebases relates to the fundamental mismatch between modern coverage tools and Kotlin's language features. Most prevalent coverage analysis tools, including JaCoCo, were originally designed for Java and not specifically engineered to accommodate Kotlin's distinctive language constructs. Although these tools can analyse Kotlin code by examining the compiled JVM bytecode, they frequently struggle to properly interpret certain Kotlin-specific patterns, particularly those involving null safety mechanisms. A quintessential example of this limitation manifests in the analysis of Kotlin's safe call operator (`?.`). When safe call chains are employed, coverage tools often incorrectly flag logically unreachable code branches as uncovered. Consider the following illustrative function:

```
fun processCharacter(string: String?, index: Int): String? {
    return string?.elementAt(index)?.uppercase()
}
```

Listing 6.1: Sample Kotlin function with safe call chain

In this scenario, a conventional coverage tool like JaCoCo would identify four potential execution branches:

1. Non-null string with in-bounds index: Returns uppercase character
2. Non-null string with out-of-bounds index: Returns null
3. Null string with in-bounds index: Returns null
4. Null string with out-of-bounds index: Returns null

However, this analysis fundamentally misinterprets Kotlin's null safety semantics. In reality, branches 3 and 4 represent the same execution path—when `string` is null, the entire chain short-circuits at the first safe call, making the index value entirely irrelevant. Consequently, there are only three distinct logical branches to cover, not four. Despite this logical reality, coverage tools will report incomplete branch coverage, erroneously indicating that one branch remains untested even when all viable execution paths have been thoroughly exercised. This discrepancy between logical code paths and reported coverage metrics can significantly distort quality assessments in Kotlin projects, potentially leading to unnecessary testing efforts or misguided quality assurance priorities.

Implementation of Code Refactoring for Coverage Accuracy In this project, the mismatch between Kotlin's null safety features and JaCoCo's branch coverage analysis was not identified during the initial development phases. As a result, the team invested significant effort to refactor code patterns similar to the example function above. One particularly effective refactoring strategy involved wrapping safe-call chains in Kotlin's built-in `let` function, as demonstrated below:

```
fun processCharacter(string: String?, index: Int): String? =
    string?.let { nonNullString ->
        nonNullString.elementAt(index)?.let {character ->
            character.uppercase()
        }
    }
```

Listing 6.2: Refactored function using let for improved testability

While this approach did introduce additional nesting in the code structure, it represents the Kotlin-idiomatic approach to ensuring both testability and reliable results. The `let` function creates a clear scope for non-null values, allowing code coverage tools to more accurately identify and track distinct execution branches.

After thorough consideration, the development team determined that reverting these refactorings would not be beneficial, as the improved coverage accuracy outweighed the aesthetic drawbacks of increased nesting. It is important to note that code sections exhibiting what might appear as excessive nesting due to these `let`-based refactorings should be understood as having effectively one fewer level of logical nesting from a readability and maintainability perspective.

Appendix: Comprehensive Testing Coverage Analysis and Report Generation Procedures

Coverage Report Generation Procedures

The testing infrastructure has been configured to generate detailed coverage reports on demand. These reports provide granular insights into specific code segments that have been validated through our testing protocols. Stakeholders interested in examining the current coverage status can generate these reports using the following procedures:

Backend Coverage Report Generation

To generate comprehensive backend coverage documentation, navigate to the `one-day-poc-server` directory in your terminal environment and execute the appropriate command for your operating system:

- `npm install` and then depending on your system:
- **For Linux/macOS environments:**
  `./gradlew clean test jacocoMergedReport`
- **For Windows environments:**
  `gradlew clean test jacocoMergedReport`

Upon successful execution, the system will produce a detailed HTML report accessible at:
`one-day-poc-server/build/reports/jacoco/jacocoMergedReport/html/index.html`

This report provides visualisation of code coverage through color-coded highlighting, allowing for efficient identification of both well-tested and undertested code segments.

Frontend Coverage Report Generation

To generate the frontend coverage analysis, navigate to the `one-day-poc-client` directory and execute the following sequence of commands:

1. Ensure all dependencies are properly installed (required only for first-time execution):
   `npm install`
2. Execute the testing suite with coverage analysis enabled:
   `npm run test`

Following successful test execution, a comprehensive coverage report will be generated and stored at:
`one-day-poc-client/coverage/index.html`

This interactive report provides detailed metrics on component-level coverage, enabling targeted improvements to testing protocols as needed during future development iterations.