# Software Project Final Report

# ClimaTune Prototype

Matthew Brown - 2831661 | Rafi Maciejewski - 2911235 | Rishabh Parmar – 2861287

## Table of Contents

# List of Figures

Figure 1: Development Gantt Chart

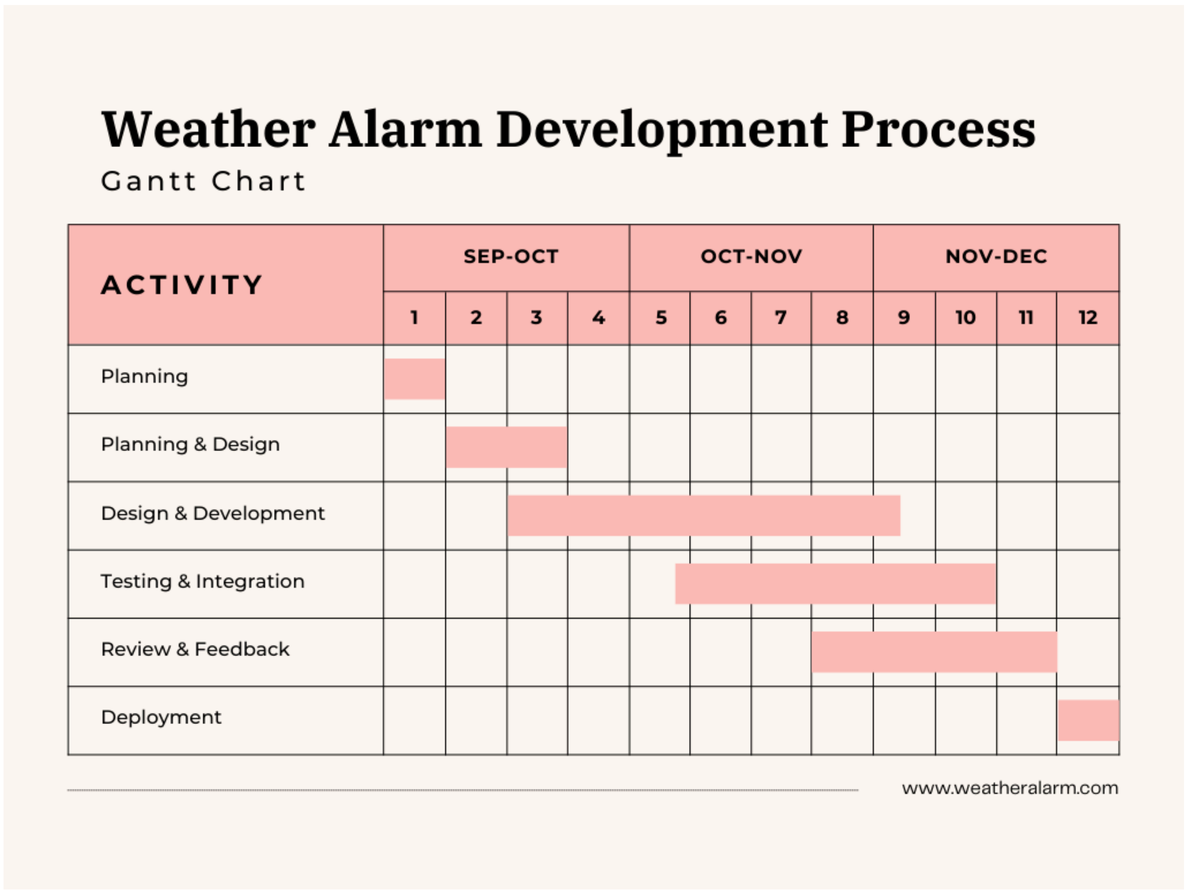
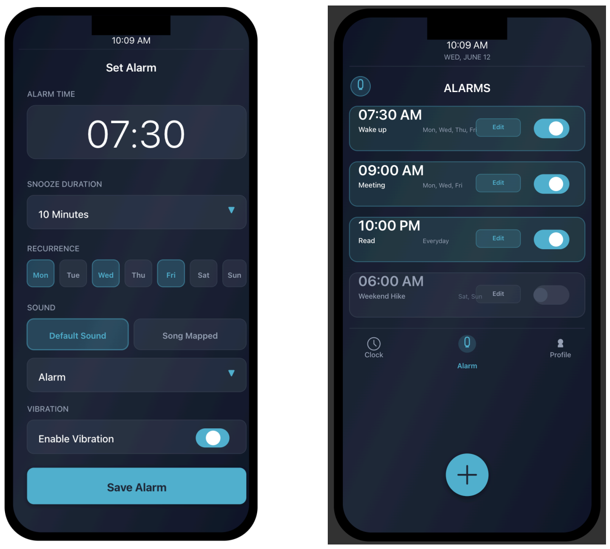


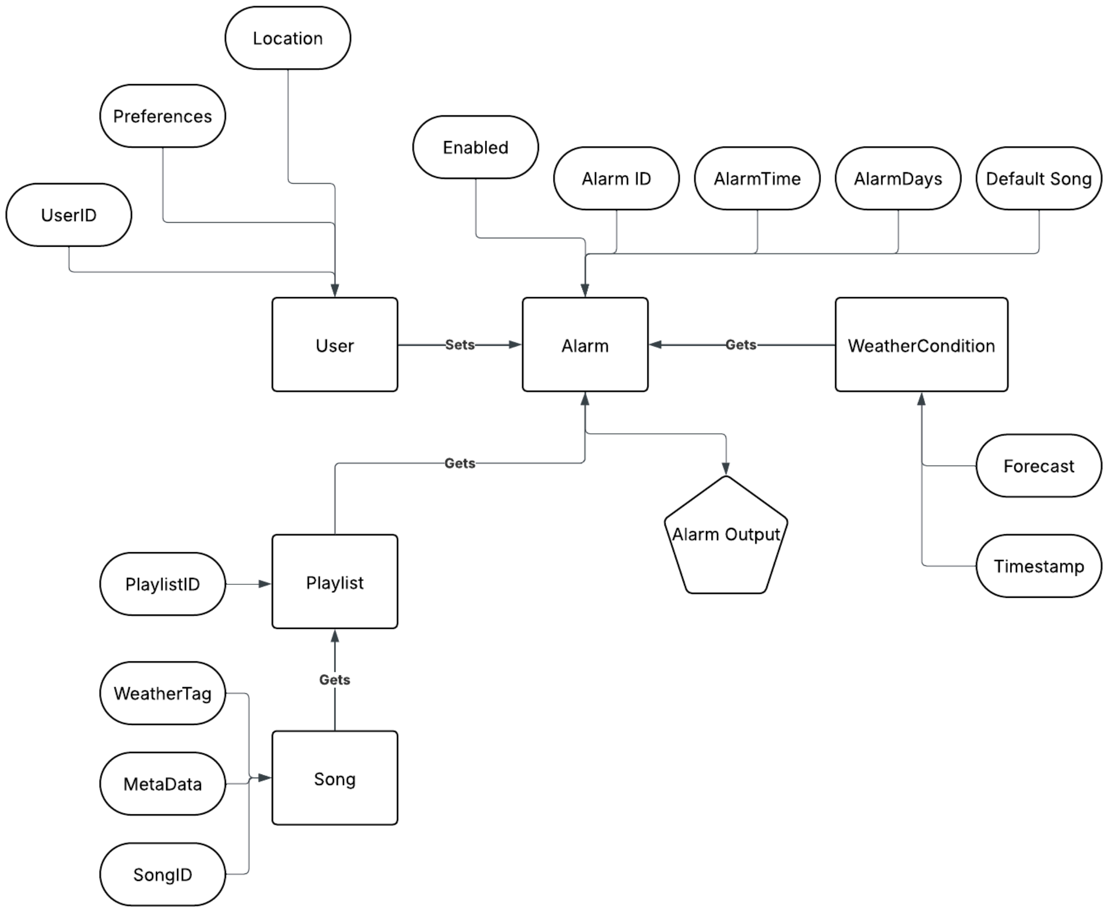
Figure 2: User Interface



ALARM TIME
07:30
SNOOZE DURATION
10 Minutes
RECURRENCE
Mon Tue Wed Thu Fri Sat Sun
SOUND
Default Sound
Song Mapped
Alarm
VIBRATION
Enable Vibration

07:30 AM
Wake up Mon, Wed, Thu, Fri
09:00 AM
Meeting Mon, Wed, Fri
10:00 PM
Read Everyday
06:00 AM
Weekend Hike Sat, Sun

Figure 3: Complete Data Model



# List of Tables

Table1: Risk Table

| ID | Risk | Probability | Impact | RM3 Pointers |
|---|---|---|---|---|
| R1 | API integration failure or limitations | Medium | High | Prototype APIs early; document fallback paths; maintain mock APIs for testing |
| R2 | Time constraints due to semester schedule | High | High | Prioritize core alarm and fallback functionality; de-scope non-essential features |
| R3 | Internet dependency (Spotify/Weather APIs unavailable) | Medium | High | Implement offline fallback alarm sound; cache last known weather where possible |
| R4 | OAuth authentication/security complexity | Medium | Medium | Research OAuth early; follow official documentation; consult references and sample apps |
| R5 | Fallback alarm failure | Low | High | Rigorously test offline alarm path; make fallback default sound independent of APIs |
| R6 | Performance lag in triggering alarms | Medium | High | Optimize alarm scheduling; test on multiple devices; minimize background load |
| R7 | Lack of expertise in APIs/tools | Medium | Medium | Assign research tasks early; share findings in team meetings; allocate buffer time |
| R8 | Licensing/usage limitations | Low | Medium | Monitor API call limits; plan for demo within free tiers; document compliance clearly |
| R9 | Poor usability of weather-to-song mapping | Medium | Medium | Conduct early usability testing; simplify UI; allow default mappings if users don't customize |

Table 2: Data Dictionary

| **Entity:** | **Attribute:** | **Type:** | **Description:** |
|---|---|---|---|
| User: | UserID | UUID / String | Unique identifier for each user. |
| | Preferences | JSON / String | User-defined rules for mapping weather conditions to songs. |
| | Location | String | Geographic location used for weather retrieval. |
| Alarm: | AlarmID | UUID / String | Unique identifier for each alarm. |
| | AlarmTime | Time | Scheduled time of the alarm. |
| | AlarmDays | Array[String] | Days of the week on which the alarm repeats. |
| | Enabled | Boolean | Indicates whether the alarm is active. |
| | DefaultSong | String / URI | A fallback sound or song played if APIs are unavailable. |
| WeatherCondition: | Forecast | Enum/String | Weather forecast description (e.g., “Sunny,” “Rainy,” “Snowy”). |
| | Timestamp | DateTime | Time when the weather data was retrieved. |
| Playlist: | PlaylistID | String | Spotify playlist linked to the user. |
| Song: | SongID | UUID / String | Unique identifier for each song. |
| | MetaData | JSON / String | Song metadata including title, artist, and Spotify URI. |
| | WeatherTag | Array[String] | One or more weather conditions associated with the song. |
| Alarm Output: | N/A | Process Event | Represents the system’s output when an alarm is triggered, combining time, weather, and selected song. |

## Table 3: Objects

| **Object Name** | **Description** |
|---|---|
| Alarm Object | The Alarm object represents a single alarm configured by the user and contains all necessary information for scheduling, triggering, and playing the appropriate audio. |
| Weather Data Object | Weather Data Object The Weather Data object stores current or forecasted weather information retrieved from the OpenWeatherMap API. |
| Weather-to-Song Mapping Object | This structure maps weather conditions to Spotify playlists, allowing the system to select appropriate songs based on current weather. |
| Spotify Playlist Object | Represents a Spotify playlist available to the application after authentication. |
| User Authentication Object | Stores authentication tokens and user session information for Spotify API access. |
| User Settings Object | Contains application-wide user preferences and configuration. |
| Cache Object | Stores temporary data to support offline functionality and reduce API calls. |

## Table 4: Storage Keys

| **Key** | **Data Type** | **Description** |
|---|---|---|
| climatune_alarms | Array<Alarm> | All user-created alarms |
| climatune_settings | UserSettings | Application settings and preferences |
| climatune_weather_mappings | Array<WeatherMapping> | Weather-to-playlist mappings |
| climatune_auth | AuthToken | Encrypted Spotify authentication tokens |
| climatune_cache | CachedData | Cached weather and playlist data |

Table 5: Component Description

| Component Name | Interface Description | Static Model | Dynamic Model |
|---|---|---|---|
| 3.2.1 Alarm Management Component | Handles creation, editing, enabling/disabling, and deletion of alarms. Manages alarm scheduling, triggering, and user notifications. When activated, retrieves current weather data, determines the appropriate song, and initiates playback. | Defines an Alarm object with attributes such as AlarmID, AlarmTime, RepeatDays, Enabled, and DefaultSong. | User sets an alarm → parameters are saved → system waits for scheduled time → weather data is fetched → mapped song or fallback sound is played. |
| 3.2.2 Weather Integration Component | Connects to the OpenWeatherMap API to retrieve current or forecasted weather data. Parses JSON responses and extracts temperature, condition codes, and descriptions for use in song selection. | Represents a WeatherCondition object with fields such as Forecast, Timestamp, and Location. | At alarm trigger, the component calls the weather API → receives data → extracts condition → passes it to the song selection logic. |
| 3.2.3 Spotify Integration Component | Manages OAuth 2.0 authentication and communication with the Spotify Web API. Retrieves playlists, tracks, and associated metadata for song selection and playback. | Defines a Song object with attributes such as SongID, Metadata, and WeatherTags. | Upon user login, component retrieves access token → fetches playlists → matches song to weather tag → streams audio through Spotify. |
| 3.2.4 Sound Playback Component | Controls all aspects of alarm sound playback, including fallback sounds and volume control. Ensures accurate timing and reliable audio output even during connectivity issues. | Uses built-in or external audio playback objects to manage streams and sound buffers. | Receives song URI → loads and plays sound → loops or stops as configured → handles playback errors gracefully. |
| 3.2.5 User Interface Component | Provides graphical user interface for alarm management, playlist linking, and weather-song mapping. Designed for clarity, accessibility, and ease of use. | Implements HTML/CSS/JavaScript components for form input, buttons, and responsive layout. | User interacts via GUI → event triggers update logic → model updates → results displayed on screen. |

| | | | |
|---|---|---|---|
| 3.2.6 System and Data Storage Component | Manages local storage of alarms, user preferences, and cached data for offline functionality. | Stores serialized UserSettings, Alarm, and Cache objects in local or indexed storage. | On save or update, component writes data locally → retrieves data on load → synchronizes with APIs when connectivity is restored. |

Table 6: Requirements Traceability Matrix

| **SRS Section** | **Design Component** | **Associated Data/Structures** |
|---|---|---|
| 4.1.1 Alarm Management | Alarm Controller, Alarm Scheduler, Alarm UI Screen | Alarm object {AlarmID, AlarmTime, AlarmDays, Enabled, DefaultSong} |
| 4.1.2 Weather Functions | Weather API | WeatherCondition object {Forecast, Timestamp, Location} |
| 4.1.3 Spotify/Apple Music Functions | Spotify Integration and Authentication | Playlist and Song objects {PlaylistID, SongID, MetaData, WeatherTag} |
| 4.1.4 Sound Functions | Alarm Audio Playback, Notification Handler | Sound Playback Settings |
| 4.1.5 User Interface Functions | Web UI, Navigation Controller | User Preferences {Preferences, ThemeMode} |
| 4.1.6 System Functions | Local Storage Management | Database Tables: Users, Alarms, Mappings |
| 4.2.1 External Machine Interfaces | System Clock, Notification Service, Audio Device Interface | OS level I/O |
| 4.2.2 External System Interfaces | Weather API Client/Spotify API Client | API Key, JSON Response Handling |
| 4.2.3 Human Interface | Frontend Interface (HTML/ CSS/JavaScript) | Screen Components |

Table 7: Test Strategy

| Test Type | Objective | Focus Areas |
|---|---|---|
| Functional Testing | Validate all features against the SRS. | Alarm scheduling, recurrence, editing. Authentication and token management (OAuth 2.0). Weather-to-song selection logic. |
| Integration Testing | Verify correct communication between modules and external systems. | Alarm Controller to Alarm Scheduler. Weather API Client to OpenWeatherMap API. Spotify Integration to Spotify Web API. |
| Constraint Testing | Verify system behavior under limitations. | Offline functionality (playing fallback sound during connection failure). Handling of API latency and rate-limit constraints (OpenWeatherMap). |
| Usability Testing | Ensure the Human Interface is intuitive and easy to use. | Clarity, accessibility, and ease-of-use of the frontend (e.g., alarm setup, playlist management). |

Table 8: Test Schedule

| Testing Phase | Duration | Milestones/Deliverables |
|---|---|---|
| Unit Testing | Continuous | Module-level tests for Alarm Controller and Alarm Scheduler. |
| Integration Testing | Immediately following component development | Successful integration of OpenWeatherMap data fetching and Spotify OAuth 2.0 flow. |
| System/Functional Testing | Once project will be complected and prior to presentation | Full execution of all test cases, leading to the Final Test Report. |

Table 9: Test Cases

| ID | Test Input | Expected Output | Description |
|---|---|---|---|
| TC-01 | Create a new alarm for 7:00 AM | Alarm is saved and appears in list | Tests alarm creation function |
| TC-02 | Edit existing alarm to 7:30 AM | Alarm time updates correctly | Ensures editing works |
| TC-03 | Disable an active alarm | Alarm no longer triggers | Tests enable/disable toggle |
| TC-04 | Snooze active alarm | Alarm reactivates after defined snooze time | Validates snooze feature |
| TC-05 | No internet connection during alarm trigger | Default alarm sound plays | Tests fallback behavior |
| TC-06 | Retrieve current weather from API | Weather condition (e.g., "Rain") displays correctly | Confirms API connectivity |
| TC-07 | Weather API unavailable | Error message displays and fallback sound used | Tests error handling |
| TC-08 | User logs into Spotify | Spotify authentication successful | Verifies OAuth flow |
| TC-09 | Retrieve playlists from Spotify | User playlists are displayed | Tests playlist retrieval |
| TC-10 | Assign playlist to "Sunny" condition | Playlist successfully saved | Tests mapping logic |
| TC-11 | Alarm triggers on a rainy day | Weather-mapped playlist plays | Validates weather-music integration |
| TC-12 | Adjust alarm volume | Sound plays at updated volume | Tests sound control |
| TC-13 | Access settings and change theme | UI switches to dark or light mode | Tests UI customization |
| TC-14 | Delete alarm | Alarm is removed from list | Confirms deletion functionality |
| TC-15 | Log out of Spotify | Session is cleared | Tests logout and token revocation |
| TC-16 | Invalid input (blank alarm time) | Warning displayed, alarm not saved | Validates input validation |
| TC-17 | Alarm triggers with valid playlist and weather | Correct song plays | Full system test case |

# 1.0 Introduction

### 1.1 Purpose and Scope

This final report documents the development and evaluation of the ClimaTune prototype, a web-based alarm system that selects alarm audio based on real-time or forecasted weather conditions. The purpose of the report is to summarize the project's objectives, requirements, design decisions, architectural structure, testing results, and overall outcomes. The scope of the project included creating a functional prototype that demonstrates the integration of alarm scheduling, weather data retrieval, Spotify playlist access, and the mapping of weather categories to user-selected music. The system emphasizes reliable alarm triggering, personalization, and fallback behavior under network or API failures.

### 1.2 Product Overview

ClimaTune offers users a way to personalize their morning routines by connecting environmental conditions with music. Users create alarms through a browser-based interface, link their Spotify accounts through OAuth 2.0, and assign playlists or individual songs to categories such as sunny, rainy, or snowy. When an alarm activates, the system queries the OpenWeatherMap API to determine current conditions and select an appropriate audio track. If weather retrieval or Spotify access fails, the system plays a built-in fallback alarm sound to preserve essential alarm functionality. The prototype operates entirely within a browser environment, storing user preferences locally and relying on web APIs for dynamic data. Its design prioritizes simplicity, reliability, and ease of use.

### 1.3 Structure of the Document

This document proceeds by presenting the project's management plan, a summary of system requirements, the architectural and design models used during development, the testing approach and results, and a concluding reflection on achievements, lessons learned, and possible future expansion.

### 1.4 Terms, Acronyms, and Abbreviations

Throughout this report, several technical terms recur. "API" refers to external web-based service interfaces used to obtain Spotify and weather data. "OAuth 2.0" designates the authorization protocol enabling secure user login to Spotify. “URI” refers to identifiers used to retrieve songs. Acronyms such as SRS, SDS, and Test Plan refer to the project’s formal requirement, design, and testing documents.

## 2.0 Project Management Plan

### 2.1 Project Organization

The development team consisted of three members who collaborated on gathering requirements, designing the system, integrating external APIs, implementing the prototype, and testing it. Responsibilities were shared in iterative cycles, allowing the team to coordinate efforts across alarm functionality, user interface design, API integration, and documentation. Regular communication ensured that each stage of development aligned with prior planning and the evolving needs of the system.

### 2.2 Lifecycle Model Used

The project followed an Agile Incremental Development Model as outlined in the Project Plan. This approach divided the work into several increments, each adding a key subsystem: initial alarm scheduling, weather API integration, Spotify connectivity and playback, and user interface refinement. Each increment built on the previous one, allowing early testing of essential functions such as alarm triggering and fallback behavior. The incremental model also allowed the team to adapt to the complexities of OAuth authentication and external API dependencies as they emerged.

### 2.3 Risk Analysis

Several risks were identified early in development. Integration with external APIs such as Spotify and OpenWeatherMap presented uncertainties in terms of availability, authentication requirements, and rate limits. The system’s dependency on internet connectivity introduced potential points of failure, particularly since alarm reliability was central to the system’s purpose. OAuth authentication added additional security and implementation challenges. There were also risks tied to performance, as alarms needed to activate precisely at the scheduled time, and delays in API responses could lead to unacceptable latency. Time constraints from the academic semester placed limits on how many advanced features could be implemented. Many of these risks were mitigated by developing a robust fallback alarm mechanism, prototyping API interactions early, caching weather and playlist data, and prioritizing essential functionality over optional enhancements.

### 2.4 Hardware and Software Resource Requirements

The system was designed to run entirely in modern web browsers on standard consumer devices. No specialized hardware was required beyond a computer or mobile device capable of playing audio. Development required standard web technologies including HTML, CSS, and JavaScript, and relied on external APIs provided by Spotify and OpenWeatherMap. Internet connectivity was essential for retrieving weather data and accessing Spotify accounts, though fallback behavior allowed alarms to function offline.

### 2.5 Deliverables and Schedule

Project deliverables included the Project Plan, Software Requirement Specification, Software Design Specification, Software Test Plan, prototype implementation, and this final report. The schedule followed the timeline in the Project Plan, beginning with planning and scope definition, followed by design and initial alarm implementation, then incremental addition of weather and Spotify services, and concluding with UI integration, testing, and documentation.

## 3.0 Requirement Specifications

### 3.1 Stakeholders

The primary stakeholders for the system were end users seeking a personalized alarm clock that integrates music and weather information. These users required a reliable, simple interface with minimal configuration burden. Secondary stakeholders included developers and testers responsible for validating alarm precision, API communication, and fallback behavior during simulated failures.

### 3.2 Use Cases

ClimaTune's functionality is defined by several core use cases. Users can create, edit, and delete alarms, specifying times, recurrence patterns, and preferred sound options. They can link their Spotify accounts through OAuth, which grants the application access to playlists. A separate interface allows users to map particular weather categories to specific songs or playlists. When an alarm triggers, the system retrieves weather conditions and uses user-defined mappings to select a matching song. Users may snooze or dismiss alarms when they sound. Special use cases include handling internet loss, in which the system plays a default alarm sound.

### 3.3 Rationale for Use Case Model

These use cases represent the normal interactions required to support a functional, personalized alarm clock. They also reflect the system's dependence on external data sources and the need to preserve consistent user experience under varying network conditions. The scenarios capture both regular user tasks and exceptional cases where API calls fail or connectivity is lost.

### 3.4 Non-functional Requirements

Non-functional requirements emphasize reliability, security, performance, and usability. Alarms must activate at the exact scheduled time regardless of network conditions. Spotify authentication must follow secure OAuth 2.0 standards, and user data must be handled appropriately. The prototype must remain responsive, load efficiently, and require minimal cognitive effort from users, particularly during early-morning interaction. The interface should remain understandable even for users who choose not to customize weather mappings.

## 4.0 Architecture

### 4.1 Architectural Style

ClimaTune employs a three-tier architecture composed of a presentation layer, an application logic layer, and a data layer. This separation provides clarity between user-facing features, business logic, and persistent data management. The architecture supports incremental development and modular integration of external services such as Spotify and OpenWeatherMap.

### 4.2 Architectural Model

The architecture integrates several interacting components. The presentation layer displays alarm dashboards, setup screens, and mapping interfaces. The application layer contains subsystems responsible for scheduling alarms, retrieving and interpreting weather data, handling Spotify authentication and playback, and determining appropriate songs for specific conditions. The data layer stores alarms, user preferences, weather mappings, authentication tokens, and cached data using browser storage. When an alarm activates, the scheduler triggers a process that queries the weather subsystem, selects music through the Spotify subsystem, and delegates playback to the sound component. If API requests fail, the scheduler instructs the sound component to use a fallback file.

### 4.3 Technology and Resources Used

The implementation uses web technologies for the user interface and logic, including JavaScript for asynchronous API calls and data management. Spotify's Web API provides playlist and playback access once authorization tokens are obtained through OAuth 2.0. Weather data is retrieved from OpenWeatherMap using its REST API. Local Storage or IndexedDB preserve user data between sessions. The architecture also incorporates encryption for stored tokens using browser-supported cryptographic functions.

### 4.4 Rationale for Architectural Choices

The chosen structure supports the prototype's goals by providing flexibility, maintainability, and clear separation of concerns. A browser-based design reduces deployment complexity and eliminates the need for dedicated servers. The modularity of the architecture makes it easier to adjust or expand individual subsystems without disrupting the entire application. The model aligns well with the constraints of API-dependent functionality and allows caching and fallback behavior to be incorporated naturally into the system.

## 5.0 Design

### 5.1 User Interface Design

The user interface was designed to be simple, readable, and easy to navigate, especially in early-morning conditions. The alarm dashboard displays all stored alarms, indicating their times and recurrence patterns. An alarm setup screen allows adjustments to time, days, snooze duration, and sound options. Weather-to-song mapping screens present common weather categories and allow users to assign playlists retrieved from Spotify. A playlist management screen shows accessible playlists and refresh options. The settings menu provides access to Spotify login, theme preferences, and volume adjustments. The UI follows modern design conventions, offering both light and dark modes, consistent navigation, and accessible contrast levels.

### 5.2 Component Design

Major components include the alarm management subsystem, which handles alarm creation, modification, scheduling, and triggering. The weather subsystem communicates with the external weather API, parses JSON responses, and extracts relevant fields. The Spotify subsystem manages OAuth authorization, playlist retrieval, and track selection. The sound subsystem handles playback of either Spotify tracks or fallback audio files. The data subsystem manages serialization, storage, caching, and retrieval of settings, tokens, alarms, and mappings. Static and dynamic models for these components are presented in the SDS and demonstrate how data flows through the system during regular and exceptional events.

### 5.3 Database Design

Although the prototype does not use a traditional server-hosted database, it organizes client-side storage into structured categories. Alarms, settings, weather mappings, encrypted Spotify tokens, and cached data are stored under dedicated keys. Data is serialized as JSON and automatically loaded at application startup. The caching strategy reduces API calls by storing recently retrieved weather data and playlist metadata, which supports limited offline functionality.

### 5.4 Rationale for Design Models

The design supports the non-functional requirements by ensuring that essential alarm functionality remains reliable even when API requests fail. Storing all necessary data locally eliminates dependence on external servers and allows the application to persist user preferences without login sessions. The modular component structure simplifies implementation and aligns with the architectural goals of separation and maintainability.

### 5.5 Traceability

Each functional requirement in the SRS corresponds directly to components and data structures described in the SDS. Alarms map to the alarm controller and scheduler, weather functions map to the weather component, Spotify functions map to the authentication and playlist components, and UI functions map to the screens and interface elements described in the design. This traceability ensures consistency between requirements and implementation.

## 6.0 Test Management

### 6.1 System Test Cases

Testing covered all functional areas of the system, including alarm creation, editing, disabling, snoozing, and deletion; Spotify login, playlist retrieval, and logout; weather retrieval and error handling; theme selection; and validation of alarm triggering with mapped songs. Offline scenarios were tested to ensure that the fallback alarm activated properly. The complete set of test cases appears in the Test Plan and was executed prior to final evaluation of the prototype.

### 6.2 Traceability of Tests to Use Cases

Test cases map directly to use cases. For example, tests involving alarm creation, modification, and deletion correspond to the "Set Alarm" and "Edit/Delete Alarm" use cases. Tests involving Spotify authentication and playlist retrieval correspond to "Link Spotify Account." Weather-related test cases correspond to "Trigger Alarm" under varying conditions. Offline scenarios directly verify the fallback logic required by the reliability use case.

### 6.3 Techniques Used for Test Case Generation

A risk-based testing approach guided the focus of the testing process. High-risk areas such as authentication, API communication, and timing behavior were prioritized. Functional, integration, and constraint testing were used to verify correct operation across modules, confirm that external APIs interacted properly with the system, and assess how the system behaved under network loss or API latency.

### 6.4 Test Results and Assessment

Test results indicated that the system's core functionality behaved correctly and consistently. Alarms triggered at the correct times and played either the weather-appropriate song or the fallback alarm sound when necessary. The weather subsystem correctly retrieved and interpreted conditions, and error messages appeared when API access failed. Spotify authentication and playlist retrieval functioned within the expected constraints of OAuth 2.0. The weather-to-song mapping logic performed as intended, and UI interactions such as theme changes and input validation behaved predictably. Overall, the test suite provided strong coverage of essential features and validated the reliability of the prototype.

### 6.5 Defect Reports

During testing, several defects were identified, including delays in weather retrieval caused by network latency, token expiration errors in the Spotify subsystem, and occasional inconsistencies in how Local Storage synchronized alarm updates across UI components. These defects were addressed by refining asynchronous logic, improving error handling, implementing token refresh procedures, and restructuring certain UI update functions.

## 7.0 Conclusions

### 7.1 Outcomes of the Project

The project successfully met its primary objectives. The prototype provides reliable alarm scheduling, integrates with both the weather and Spotify APIs, allows user-defined weather-to-song mappings, and maintains functional behavior even when offline. The system demonstrates the intended concept of context-based music selection and offers a usable interface suitable for demonstration purposes.

### 7.2 Lessons Learned

Several lessons emerged during development. Integrating OAuth 2.0 requires a careful approach to token handling, refreshing, and security practices. External APIs introduce uncertainty in terms of response times, availability, and rate limits, making caching and fallback mechanisms essential. Browser-based environments place constraints on background execution and storage security, which required careful architectural planning. Incremental development proved effective in managing complexity and uncovering issues early.

## 7.3 Future Development

The prototype could be extended in several ways. A dedicated mobile application would improve alarm accuracy and overcome browser limitations. Expanding the system to support multiple music services or advanced scheduling features would increase flexibility. Multi-device synchronization and enhanced offline capabilities could also improve the user experience. With additional development, ClimaTune could evolve into a fully featured context-aware alarm platform.