Module Guide for Software Engineering

Team 21, Alkalytics
Sumanya Gulati
Kate Min
Jennifer Ye
Jason Tran

January 18, 2025

1 Revision History

Date	Version	Notes
11 Janury 2025	1.0	Added initial content for Rev 0.
17 Janury 2025	1.1	Finalized document for Rev 0 submission.

Note: Please note that our team has adapted and extended this Module Guide document to include the contents of an MIS or other such document. For this reason, only one design document has been submitted.

2 Reference Material

This section records information for easy reference.

2.1 Abbreviations and Acronyms

symbol	description
AC	Anticipated Change
API	Application Programming Interface
CSV	Comma Separated Values
DAG	Directed Acyclic Graph
JSON	JavaScript Object Notation
KPI	Key Performance Indicator
M	Module
MG	Module Guide
MIS	Module Interface Specification
OS	Operating System
R	Requirement
SC	Scientific Computing
SRS	Software Requirements Specification
Software Engineering	Explanation of program name
UC	Unlikely Change
UI	User Interface
XML	Extensible Markup Language

Contents

List of Tables

List of Figures

3 Introduction

3.1 Summary

Alkalytics is a project designed to provide a scalable data management and analysis solution for ocean alkalinity research, in particular by streamlining the data organization, querying, and visualization processes. Through its various modules, the primary goal of the system is to offer a comprehensive solution for data ingestion, processing and reporting while maintaining adaptability to future changes.

Each functional component is developed as an independent module to encapsulate specific responsibilities, minimize dependencies, and promote information hiding. This modular approach, advocated for widely in the software sector, not only simplifies development and testing but also allows the system to accommodate evolving user requirements and technology upgrades.

A work context model and a use case diagram for the system have been included in the Software Requirements Specification (SRS) in sections 6.2 and 8.1, respectively.

3.2 Purpose

This Module Guide (MG) has been written to serve as a roadmap for the Alkalytics system, detailing its structure, functionality, and the relationships between its components. It provides clarity on how the system meets the requirements outlined in the SRS and supports the following stakeholders:

- **New Developers**: To understand the modular architecture and ensure consistent implementation.
- **Maintainers**: To efficiently identify, update, or rewrite modules as needed.
- **Designers**: To validate the system's feasibility, flexibility, and alignment with project goals.

4 Anticipated and Unlikely Changes

This section identifies potential changes to the system and classifies them into two categories: anticipated changes (AC) as listed in section 4.1 and unlikely changes (UC) as listed in section 4.2. AC represent decisions that have been encapsulated within specific modules to minimize the impact of modifications while UC are those that, while possible, are fixed at the system architecture stage to reduce complexity.

4.1 Anticipated Changes

Anticipated changes are the source of the information that is to be hidden inside the modules. Ideally, changing one of the anticipated changes will only require changing the one module

that hides the associated decision. These changes are encapsulated within specific modules to ensure the system's adaptability.

AC1: Hardware Configuration - The software may need to run on a different hardware platform like a server or on a cloud solution. Changes in hardware specifications will primarily affect the Hardware-Hiding Module, isolating their imoact.

AC2: Data Processing Algorithms - Changes in analytical techniques along with advances in the machine learning space would introduce the need for new statistical models. These changes would be encapsulated in the Data Processing Module.

AC3: User Interface (UI) Design - Changes in analytical techniques along with advances in the machine learning space would introduce the need for new statistical models. These changes would be encapsulated in the Data Processing Module.

AC4: Input Data Formats - Currently, the system is expected to process data from Comma Separated Values (CSV) files only. In the future, however, modifications may have to be made to accommodate different file formats (such as JavaScript Object Notation (JSON), Extensible Markup Language (XML) etc) which will be handled by the Data Ingestion Module without impacting other parts of the system.

AC5: Data Source Integration - New data sources such as third-party application programming interfaces (APIs), Internet of Things (IoT) devices may have to be added in the future. The Data Ingestion Module will have to be redesigned to handle the integration of these new sources.

AC6: Scaling Data Volume - As the number of experiments increases, the system may need to handle increasing data volumes as usage grows. This is addressed by the Data Storage Module which has been designed to support database scalability strategies.

AC7: User Roles and Permissions - Future requirements may demand the addition of new user roles or changes to existing permissions. The Administration Module is designed to encapsulate these changes.

AC8: Input Schema - With an increase in the number of diverse experiments, the schema for the data inputs may have to changed to support the addition or removal of new parameters. This is handled by the Data Ingestion Module.

AC9: Notification Rules - The conditions of triggering alerts or notifications may evolve, including but not limited to additional thresholds or new types of anomalies. These are handled by the Notifications Module without affecting other parts of the system.

AC10: Analytical Metrics - New metrics or Key Performance Indicators (KPIs) might be requested by stakeholders. This would involve adapting requirements by introducing new calculations or processing pipelines by modifying the Data Processing Module.

4.2 Unlikely Changes

Unlikely changes are those that are fixed early in the design to simplify the system and reduce complexity. These changes, if necessary, would have a significant impact on multiple modules.

UC1: Input/Output Devices - The system is designed to support file-based inputs. Changes can include additional input and/or output methods such as direct hardware interaction, would require substantial redesign across multiple modules.

UC2: Core System Architecture - The underlying architectural decisions, such as the use of modular decomposition and separation of concerns, are not expected to change. Altering these decisions would necessitate a complete overhaul of the system.

UC3: Communication Protocols - The communication methods between modules such as function calls, API interactions etc are fixed. Switching to a different communication protocol would impact the interfaces of all interacting modules.

UC4: Programming Language - The choice of programming languages is asssumed to be fixed for the project. A change would require rewriting most of the system.

UC5: Database Type - The choice of storage solution (relational versus NoSQL databases, for example) is assumed to remain fixed. Switching to a different type of database would require reworking the Data Storage Module and parts of the Data Processing Module.

5 Module Hierarchy

This section provides an overview of the module design for the Alkalytics system. The modules are summarized in a hierarchy that follows the principle of information hiding. Each module encapsulates specific secrets, ensuring changes are localized and do not affect unrelated parts of the system. These modules are summarized in a hierarchy decomposed by secrets in table 1. This hierarchy represented as a directed acyclic graph, shown in **??**, shows relationships betweem higher-level and lower-level modules, with the leaf modules representing those that will actually be implemented.

M1: Hardware-Hiding Module

M2: Behaviour-Hiding Module

M3: Interface Module

M4: Administration Module

M5: Data Acquisition Module

M6: Data Storage Module

M7: Data Retrieval Module

M8: Input Module

M9: Processing Module

M10: Output Module

M11: UI Design Module

M12: Visualization Module

M13: User Management Module

M14: Configuration Management Module

M15: Data Ingestion Module

M16: Data Validation Module

M17: Data Transformation Module

M18: Machine Learning Module

M19: Reporting Module

M20: Notification Module

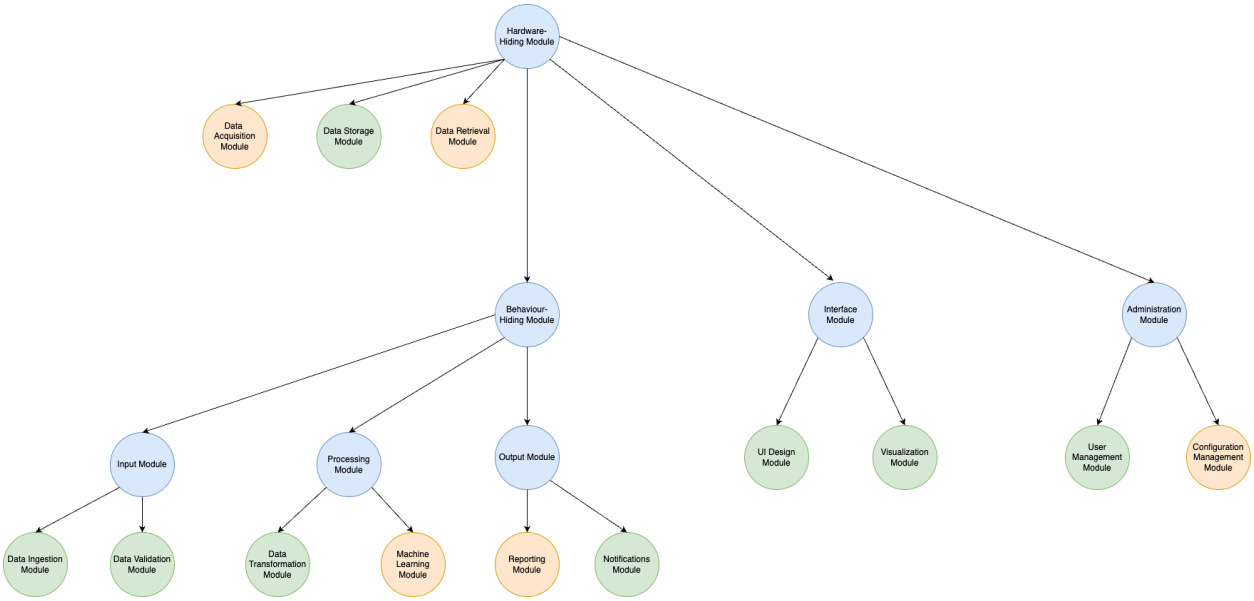

Figure 1: A DAG representing the implemented module hierarchy of Alkalytics.

It must be noted that the blue nodes shown in figure 1 represent 'container' modules that are non-leaf modules and encompass other leaf nodes. The orange nodes represent leaf modules that will not be implemented for Revision 0 of Alkalytics and the green nodes represent leaf modules that will be implemented.

Level 1	Level 2	Level 3
Hardware-Hiding Module		
	Data Acquisition Module	
	Data Storage Module	
	Data Retrieval Module	
Behaviour-Hiding Module	Input Module	Data Ingestion Module
		Data Validation Module
	Processing Module	Data Transformation Module
		Machine Learning Module
	Output Module	Reporting Module
		Notifications Module
Interface Module	UI Design Module	
	Visualization Module	
Administration Module	User Management Module	
	Configuration Management Module	

Table 1: Module Hierarchy

6 Module Decomposition

Modules are decomposed according to the principle of "information hiding" proposed by Parnas et al. (1984). In this section, the *Secrets* field acts as a brief statement of the design decision hidden by the module. The *Services* field specifies *what* the module will do without documenting *how* to do it.

For each module, a suggestion for the implementing software is given under the *Implemented By* title. If the entry is *OS*, this means that the module is provided by the operating system or by standard programming language libraries. *Software Engineering* means the module will contain custom code and thus, will be implemented by the software engineering team.

Descriptions for non-leaf modules or the modules that will not be implemented for Revision 0 of Alkalytics have been provided below.

6.1 Data Storage Module (M6)

Secrets: The data structure and algorithm used to store the received CSV files into the NoSQL database.

Services: This module handles the storage of system data by interacting with a NoSQL

database. It receives the transformed data from the Data Ingestion Module in a suitable format for storage in the database. This module also provides interfaces to query, retrieve, and update stored data.

Implemented By: Software Engineering

Rationale: This module abstracts the complexities of the NoSQL database structure, ensuring flexibility in storage design. It allows the system to efficiently store the data in a schema-less NoSQL database. This ensures scalability and adaptability to handle large datasets with varying structures.

6.1.1 Uses

This module is used by the Behavior-Hiding and Interface Module to interact with the stored data. It provides mechanisms transform the CSV data, and perform CRUD (Create, Read, Update, Delete) operations on the NoSQL database.

6.1.2 Syntax

Exported Constants and Access Programs:

`DB_CONNECTION_STRING`: The connection string for accessing the NoSQL database.

`MAX_BATCH_SIZE`: The maximum number of rows to process and upload in a single batch.

Name	In	Out	Exceptions
`initializeDB`	-	Boolean	Throws `DatabaseInitializationError` if the database connection fails.
`uploadData`	filePath(string)	Boolean	Throws `FileNotFoundError` if the file does not exist, or `DataTransformationError` if the Transformation fails.
`queryData`	query(string)	Data (JSON)	Throws `QueryExecutionError` if the query is invalid.
`updateData`	query(string), updates	Boolean	Throws `UpdateError` if the update fails.

Table 2: Exported Access Programs for the Data Storage Module

6.1.3 Semantics

State Variables:

`database`: Represents the connection to the NoSQL database.

`transformationRules`: Defines the set of rules for transforming CSV data to JSON format.

Environment Variables:

- Interacts with the local file system to access CSV files.
- Connects to an external NoSQL database, MongoDB to be specific.

Assumptions:

- Assumes the NoSQL database is accessible and the credentials are correct.
- Assumes the received CSV files are well-formed and follow the expected schema.
- Assumes sufficient storage space is available in the database.

Access Routine Semantics:

`initializeDB()`:

- **Transition**: Establishes a connection to the NoSQL database and validates the connection.
- **Output**: Returns `True` if the database is successfully initialized. Throws a `DatabaseInitializationError` otherwise.

`uploadData(filePath:string)`:

- **Transition**: Reads the specified CSV file, transforms the contents into JSON format using `transformationRules`, and uploads the transformed data to the database in batches.
- **Output**: Returns `True` if successful. Throws `FileNotFoundError` if the file does not exist or `DataTransformationError` if the transformation fails.

`queryData(query:string)`:

- **Transition**: Executes the specified query on the database and retrieves the corresponding data.
- **Output**: Returns the retrieved data in JSON format. Throws `QueryExecutionError` if the query is invalid or fails.

`updateData(query:string, updates)`:

- **Transition**: Applies the specified updates to the records matching the query in the database.
- **Output**: Returns `True` if successful. Throws `UpdateError` if the operation fails.

Local Function:

`transformCSVToJSON(filePath:string)`:

- **Description**: Reads a CSV file and converts its rows into JSON objects based on `transformationRules`.
- **Output**: Returns a list of JSON objects or throws `DataTransformationError` if the transformation fails.

`validateConnection()`:

- **Description**: Checks the connection to the NoSQL database and returns `True` if valid.

6.2 Data Ingestion Module (M15)

Secrets: The data structure and algorithm used to ingest and preprocess data from the external source (CSV files) into the system.

Services: This module is responsible for ingesting external data into the system. It interacts with various external data sources and converts incoming data into a standardized format for further processing. The module ensures and facilitates seamless data flow into the system for subsequent operations.

Implemented By: Software Engineering

Rationale: The Data Ingestion Module abstracts the complexities of interacting with an external data source enabling the system to ingest data in CSV. By ensuring data Validation and transformation, the module guarantees data integrity and consistency before it is processed further. It ensures scalability, adaptability and flexibility in handling data from a different source while maintaining the quality of ingested data.

6.2.1 Uses

The Data Ingestion Module is used by the Data Processing Module to feed raw data into the system for analysis and storage. It also supports integration with external services to retrieve and preprocess data as needed. This module, in addition to the Data Validation Module ensures that all incoming data is correctly formatted and validated before entering the system.

6.2.2 Syntax

Exported Constants and Access Programs:

Name	In	Out	Exceptions
`initializeIngestion`	-	Boolean	Throws `IngestionError` if the process fails.
`ingestData`	sourcePath(string)	Boolean	Throws `FileNotFoundError` if the file does not exist, or `DataTransformationError` if the Ingestion fails.
`validateData`	data(Object)	Boolean	Throws `DataValidationError` if the data fails validation.
`retrieveAPIData`	apiEndPoint(string)	Data (JSON)	Throws `APIRequestError` if the API request fails.
`processData`	rawData(Object)	Object	Throws `DataProcessingError` if the processing fails.

Table 3: Exported Access Programs for the Data Ingestion Module

6.2.3 Semantics

State Variables:

`ingestionSource`: Represents the path or endpoint from which data is ingested.

`dataFormat`: Defines the expected format of incoming data (CSV, in this case).

Environment Variables:

- Interacts with the local file system to access CSV files.

Assumptions:

- Assumes the external data sources are available and accessible.
- Assumes the incoming data follows the expected format and schema.
- Assumes adequate network connectivity for external API requests.
- Assumes sufficient memory and storage to handle large volumes of incoming data.

Access Routine Semantics:

`initializeIngestion()`:

- **Transition**: Initialized the data ingestion process by configuring connections to external data sources.
- **Output**: Returns `True` if initialization is successful. Throws an `IngestionInitializationError` if initialization fails.

`ingestData(sourcePath:string)`:

- **Transition**: Reads the specified data source, retrieves the raw data and prepares it for processing.
- **Output**: Returns `True` if successful. Throws `FileNotFoundError` if the source path is invalid or does not exist, or `DataTransformationError` if the transformation process fails.

`retrieveAPIData(apiEndPoint:string)`:

- **Transition**: Sends a request to the specified API endpoints and retrieves data in JSON format.
- **Output**: Returns the retrieved data in JSON format. Throws `APIRequestError` if the request fails.

`processData(rawData:Object)`:

- **Transition**: Processes the raw data by applying transformation rules such as data normalization, mapping, filtering and more.
- **Output**: Returns the processed data in the appropriate format. Throws `DataProcessingError` if the processing fails.

Local Function:

`transformRawDataToStandardFormat(sourcePath:string)`:

- **Description**: Reads the raw data from the specified source and applies the necessary Transformation rules and converts the data into a standardized format.
- **Output**: Returns the transformed data as a JSON object or throws `DataTransformationError` if the transformation fails.

`validateSourceConnection()`:

- **Description**: Checks the connection to the external data source and returns `True` if valid.

- **Output**: Returns `True` if the connection to the source is valid. Throws `SourceConnectionError` if the connection fails.

6.3 Data Validation Module (M16)

Secrets: The data structure is a finite-state machine designed to model transitions based on real-time input data. The algorithm uses discrete math principles to validate parameters by comparing them against predefined thresholds and conditions derived from system requirements.

Services: This module ensures valid data is provided for processing by validating input streams, flagging anomalies, and initiating corrective actions when necessary.

Implemented By: OS and Software Engineering

Rationale: Ensuring the integrity and realiability of input data is critical for accurate system operation. The FSM representation enables systematic handling of transitions, facilitating robust validation and error detection mechanisms.

6.3.1 Uses

It is used to validate the sensor readings against operational thresholds while monitoring real-time data streams for deviations or inconsistencies. This module serves as a pre-processing step to ensure clean input data for dependent modules.

6.3.2 Syntax

Exported Constants:

`MAX_VOLTAGE`: Maximum allowable voltage (e.g., 10 V).

`MIN_VOLTAGE`: Minimum allowable voltage (e.g., 0 V).

`MAX_CURRENT`: Maximum allowable current (e.g., 1 A).

`MIN_CURRENT`: Minimum allowable current (e.g., 0 A).

`FLOW_RATE_THRESHOLD`: Acceptable range for flow rates (e.g., 0.1 to 60 L/min).

`PH_RANGE`: Acceptable pH range (e.g., 6.0 to 8.5).

Exported Access Programs:

`validate_voltage(v:Real)` $\rightarrow$ `Boolean`

`validate_current(i:Real)` $\rightarrow$ `Boolean`

`validate_flow_rate(f:Real)` $\rightarrow$ `Boolean`

validate_ph(v:Real) → Boolean

get_error_flags() → Set

6.3.3 Semantics

State Variables:

- voltage: Current voltage reading.
- current: Current current reading.
- flow_rate: Current flow rate reading.
- ph: Current pH value.
- error_flags: Set of flags indicating validation failures.

Environment Variables:

- Sensor input streams.
- System clock for time-based validation.

Assumptions:

- Sensors provide data at consistent intervals.
- Thresholds are pre-configured and static unless updated manually.
- Input data is numeric and within hardware limits.

Access Routine Semantics:

validate_voltage(v:Real):

- If MIN_VOLTAGE $\leq v \leq$ MAX_VOLTAGE, return True.
- Else, add Voltage_Error to error_flags and return False.

validate_current(i:Real):

- If MIN_CURRENT $\leq i \leq$ MAX_CURRENT, return True.
- Else, add Current_Error to error_flags and return False.

validate_flow_rate(f:Real):

- If FLOW_RATE_THRESHOLD[0] $\leq f \leq$ FLOW_RATE_THRESHOLD[1], return True.
- Else, add Flow_Rate_Error to error_flags and return False.

validate_ph(p:Real):

- If `PH_RANGE[0]` $\leq p \leq$ `PH_RANGE[1]`, return True.
- Else, add `PH_Error` to `error_flags` and return False.

`get_error_flags()`:

- **Output:** Returns the current set of error flags.

Local Function:

`is_within_range(value: Real, range: Tuple[Real, Real])` $\rightarrow$ `Boolean`:
Checks if a given value lies within a specified range.

6.4 Data Transformation Module (M17)

Secrets: The module uses a structured format to store input data and processed results, ensuring efficient access and manipulation during analysis and transformation. In addition to that, the module implements algorithms to process raw data, apply transformations, and perform statistical or mathematical operations, ultimately generating graph-ready data for the visualization module.

Services: The modules handles data transformation and analysis tasks by interfacing directly with both the input data and the visualization module which requires processed data for display. This module performs calculations such as data normalization, aggregation, or statistical analysis, and ensures that data is presented in a format that can be used by the visualization module for graph rendering.

Implemented By: OS

Rationale: The module is designed to centralize the data transformation and analysis logic, providing the core functionality needed for the system to process raw inputs and generate graphical outputs. By abstracting these tasks from other system components, this module enables a cleaner and more modular architecture, improving maintainability and reusability.

6.4.1 Uses

Data Transformation: The module processes raw input data, applying necessary transformations to make the data ready for analysis. This could include scaling, filtering, or feature extraction.

Graph Data Preparation: Once transformed, the data is formatted for graphing. This could imvolve calculating aggregates, trends, or applying statistical functions to produce actionable insights.

Interface with Visualization Module: The module feeds processed data into the visualization module, which then generates graphical outputs based on the processed data.

6.4.2 Syntax

Exported Operations:

`transformData(inputData)`: Applies predefined transformations to the input data.

`generatedGraphData(transformedData)`: Prepares the data for use by the visualization module.

Exported Constants:

`TRANSFORMATION_TYPE`: Defines the types of transformations available including normalization and filtering.

`GRAPH_TYPES`: Constants indicating types of graphs to be generated.

Exported Access Programs:

`startTransformation()`: Initializes the transformation process.

`retrieveGraphData()`: Accesses processed data for the visualization module.

6.4.3 Semantics

State Variables:

`isTransformed`: Boolean indicating whether the data has been transformed.

`transformedData`: Holds the data after transformation, ready for graphing.

Environment Variables:

`DATA_SOURCE`: The location or source from which raw input data is retrieved.

`GRAPH_OUTPUT_PATH`: Path where the generated graphs are stored or streamed to.

Assumptions:

- The input data is assumed to be in a compatible format.
- The system is capable of handling large datasets for transformation without significant performance degradation.

Access Routine Semantics:

`startTransformation()`: Initiates the data transformation process which must be completed before any graph generation can occur.

`retrieveGraphData()`: Returns the transformed data that is now ready to be passed to the visualization module.

Local Function:

`applyTransformation()`: Applies a specific transformation to the data.

`prepareGraphData()`: Formats the transformed data into a structure that the visualization module can use.

6.5 Notifications Module (M20)

Secrets: Each notification is associated with metadata such as the type of notification (info, warning, error), the recipient and the message content. This means the module uses event-driven algorithms to trigger notifications based on system actions or states.

Services: This module provides notifications to users based on predefined events or actions within the system. Notifications can include real-time alerts, status updates or reminders and can be filtered or categorized allowing users to specify which types of notifications they wish to receive.

Implemented By: Software Engineering. The module may also rely on external libraries or APIs for delivering emails or push notifications. A few examples of such libraries include SendGrid for email or Firebase for push notifications.

Rationale: The purpose of this module is to ensure that users are kept up-to-date on critical events, issues and actions within the system. Timely notifications ensure that users can act on important tasks or system status changes without needing to constantly monitor the system.

6.5.1 Uses

- **Real-Time Alerts**: To notify users in real time when significant events occur such as when a task is completed or an error is detected.
- **System Status Updates**: To inform users about the current status of ongoing processes. It also ensures that the application complies with Norman's design principles (feedback, to be specific).
- **Error and Issues Notifications**: To alert users about system errors or issues that require immediate attention such as data transformation errors and more.

6.5.2 Syntax

Exported Constants:

`NOTIFICATION_TYPES`: Defines the types of notifications (e.g., "info", "warning", "error", etc.)

`DELIVERY_METHODS`: Defines the available notification delivery methods.

`DEFAULT_PREFERENCES`: Default notification preferences for users including the frequency of notifications and the types of events they wish to be notified about.

Exported Access Programs:

`sendNotification()`: Initiates the process of sending notifications to users.

setNotificationPreferences(): Access routine to set or modify user preferences regarding notifications.

`getPendingNotifications()`: Provides access to the list of pending notifications for a user.

`triggerNotification()`: Access routine to trigger notifications based on system events.

6.5.3 Semantics

State Variables:

`currentNotifications`: Stores a list of active or pending notifications that need to be sent.

`userPreferences`: Stores user-specific notification preferences such as preferred delivery method and event types they want to be notified about.

`notificationQueue`: A queue that holds notifications waiting to be delivered.

Environment Variables:

`SMTP_SERVER`: The email server used for delivering email notifications.

`PUSH_NOTIFICATION_SERVER`: Endpoint or server for delivering push notifications to users.

Assumptions:

- The system assumes that external services like email APIs are available and functional.
- It assumes that users are registered in the system and have specified preferences for notifications.
- The system will handle retries or failures in sending notifications, ensuring that critical messages are delivered.

Access Routine Semantics:

`sendNotification(type, recipient, message)`: This routine sends a notification to a specified recipient with a message. It ensures that the notification is delivered through the correct channel based on the recipient's preferences.

`setNotificationPreferences(user, preferences)`: This routine updates the user's notification settings, including how they want to receive notifications and which types they wish to receive.

`getPendingNotifications(user)`: This routine retrieves the list of notifications that are pending delivery for a specific user.

`triggerNotification(event)`: This routine triggers the sending of a notification based on a system event.

Local Function:

`formatMessage(event)`: Formats the message to be sent in the notification based on the type of event.

`validatePreferences(preferences)`: Validates the user preferences to ensure that they are correctly set.

6.6 UI Design Module (M11)

Secrets: The design specifications and UI components that make up the overall appearance and user experience of the system.

Services: This module is responsible for the design and layout of the user interface.

Implemented By: Software Engineering

Rationale: The UI Design Module abstracts the presentation of data and interactions with the user.

6.6.1 Uses

This module is used by the Interface Module to manage the display of data and user-related actions, ensuring that the design is consistent and offers a positive user experience.

6.6.2 Syntax

Exported Constants and Access Programs:

Name	In	Out	Exceptions
`renderUI`	[components]	-	Throws `RenderError` if UI rendering fails.
`updateUI`	component	-	Throws `UpdateError` if UI components cannot be updated.

Table 4: Exported Access Programs for the UI Design Module

6.6.3 Semantics

State Variables: N/A

Environment Variables:

- Interacts with the frontend framework (i.e., React) for rendering UI components.

Assumptions:

- Assumes the frontend framework is correctly set up for rendering UI components.

Access Routine Semantics:

`renderUI()`:

- **Transition**: Renders the components to display on the user interface.
- **Output**: The rendered interface. Throws `RenderError` if the UI rendering fails.

`updateUI()`:

- **Transition**: Updates specific components of the UI based on route changes or user interactions.
- **Output**: The updated UI. Throws `UpdateError` if updates fail.

Local Function: N/A

6.7 Visualization Module (M12)

Secrets: This module implements data mapping algorithms to convert raw data into visual elements like charts, graphs, and plots. These include scaling, data aggregation, and color-coding based on user preferences or data thresholds.

Services: This module provides static and interactive visual representations of data such as charts, graphs, and dashboards enabling users to explore trends, patterns and insights through intuitive visual tools. It also integrates seamlessly with other system modules allowing processed data from the Data Transformation Module to be displayed effectively.

Implemented By: Software Engineering. The module also uses inbuilt libraries or frameworks.

Rationale: The Visualization Module is essential for conveying complex data insights in an accessible and actionable format. By transforming raw or processed data into visual forms, users can quickly grasp key trends and make informed decisions.

6.7.1 Uses

- **Trend Analysis**: To display temporal or spatial trends in datasets.
- **Comparative Analysis**: To compare different datasets or categories using bar charts, pie charts etc.
- **Anomaly Detection**: To highlight outliers or anomalies in datasets through scatter plots or heatmaps.
- **Interactive Dashboards**: To allow users to explore data dynamically using filters, zooming, and drilldowns.

6.7.2 Syntax

Exported Constants:

`CHART_TYPES`: Defines supported chart types.

`DEFAULT_STYLES`: Specifies default styles for visualizations including colours, fonts and sizes.

`INTERACTION_MODES`: Defines supported user interaction modes (e.g., "hover", "zoom", "filter", etc.).

Exported Access Programs:

`createChart(data, chartType, options)`: Generates a chart of the specified type using the provided data and configuration options.

`updateChart(chartID, newData)`: Updates an existing chart with new data.

`addInteraction(chartID, interactionType)`: Adds an interaction mode to an existing chart.

`renderDashboard(dashboardConfig)`: Renders a complete dashboard with multiple visualizations based on a configuration file.

`exportVisualization(chartID, format)`: Exports a visualization in the specified format.

6.7.3 Semantics

State Variables:

`currentVisualizations`: A dictionary that stores all active visualizations along with their metadata.

`userPreferences`: Stores user-specific preferences for visualization styles, colour schemes, and interaction settings.

`dashboardLayouts`: Maintains the configuration and layout of user-created dashboards.

Environment Variables:

`BROWSER_SUPPORT`: Ensures compatibility with web browsers for rendering visualizations.

`GRAPHICS_LIBRARY`: Specifies the underlying graphics library used for rendering.

`DATA_API`: Interface to retrieve data from the Data Transformation Module or other backend services.

Assumptions:

- Input data is preprocessed and cleaned by the Data Transformation Module before being passed to the Visualization Module.
- Users will have access to compatible devices and software capable of rendering visualizations.
- The system's graphics library is fully operational and supports the required visualization types.

Access Routine Semantics:

`createChart(data, chartType, options)`:

- **Inputs**:
  1. `data`: The dataset to be visualized.
  2. `chartType`: Type of chart to generate.
  3. `options`: Optional configuration settings such as axis labels, colours etc.
- **Effects**: Generates a visualization based on the provided inputs and adds it to the `currentVisualizations`.
- **Outputs**: Returns a unique `chartID` for the created visualization.

`updateChart(chartID, newData)`:

- **Inputs**:
  1. `chartID`: Identifier for the chart to update.
  2. `newData`: New dataset to render in the chart.
- **Effects**: Replaces the chart's data with `newData` and refreshes the visualization.

`addInteraction(chartID, interactionType)`:

- **Inputs**:
  1. `chartID`: Identifier for the chart to modify.
  2. `interactionType`: Type of interaction to enable.
- **Effects**: Enhances the chart with the specified interaction mode.

`renderDashboard(dashboardConfig)`:

- **Inputs**:
  1. `dashboardConfig`: Configuration file defining the layout and contents of the dashboard.
- **Effects**: Creates a multi-chart dashboard and stores it in `dashboardLayouts`

`exportVisualization(chartID, format)`:

- **Inputs**:
  1. `chartID`: Identifier of the chart to export.
  2. `format`: Export format.
- **Effects**: Exports the chart as a file in the specified format.

Local Function:

`scaleData(data, chartType)`: Adjusts the scale and format of input data based on the chart type, e.g., logarithmic scaling for larger ranges.

`applyStyles(chartID, styles)`: Applies style customizations.

`validateConfig(dashboardConfig)`: Ensures that the dashboard configuration file is correctly formatted and contains valid references to data and charts.

6.8 User Management Module (M13)

Secrets: The data structure and algorithm used to store and manage user credentials.

Services: This module manages the creation, retrieval, and validation of user accounts. It interacts with the database to store user information and also provides interfaces for user authentication and Role-Based Access Control (RBAC).

Implemented By: Software Engineering

Rationale: This module abstracts the management of user data, ensuring that sensitive information, such as passwords, is hashed and not stored in plain-text. It also enables the management of user access levels within the application.

6.8.1 Uses

This module interacts with the Hardware-Hiding Module for user credential storage and verification.

6.8.2 Syntax

Exported Constants and Access Programs:

`SESSION_COOKIE`: The id of the session cookie used to maintain user sessions.

Name	In	Out	Exceptions
`createUser`	email(string), password(string), role(string)	dict	Throws `UserAlreadyExistsError` if the user already exists.
`validateUser`	email(string), password(string)	dict	Throws `UserNotFoundError` if the user does not exist or `IncorrectPasswordError` if the password is incorrect.
`getUserAndRole`	email(string)	dict	Throws `UserNotFoundError` if the user does not exist.

Table 5: Exported Access Programs for the User Management Module

6.8.3 Semantics

State Variables:

`usersCollection`: Represents the MongoDB collection storing user data.

Environment Variables:

N/A

Assumptions:

- Assumes passwords are securely hashed before being stored.
- Assumes emails are unique across the system.

Access Routine Semantics:

`createUser(email:string, password:string, role:string)`:

- **Transition**: Checks if a user with the given email already exists. If not, hashes the password, creates a new user, and stores the new user in the database.
- **Output**: Returns a dictionary with user email and role if successful. Throws `UserAlreadyExistsError` if the user already exists.

`validateUser(email:string, password:string)`:

- **Transition**: Attempts to find user by email and compares hashed input password with the stored hash.
- **Output**: Returns a dictionary with user email and role if the password is valid. Throws `UserNotFoundError` if the user doesn't exist or `IncorrectPasswordError` if the password is incorrect.

`getUserAndRole(email:string)`:

- **Transition**: Retrieves user with the specified email and corresponding role.
- **Output**: Returns a dictionary with user email and role if the user exists. Throws `UserNotFoundError` if the user does not exist.

Local Function:

`hashPassword(password:string)`:

- **Description**: Hashes the given password using `bcrypt` and a randomly-generated salt for secure storage.
- **Output**: Returns the hashed password string.

7 Traceability Matrix

This section shows two traceability matrices: between the modules and the requirements and between the modules and the anticipated changes.

Req.	Module(s)
FR-1	M15
FR-2	M6
FR-3	M15
FR-4	M6
FR-5	M17
FR-6	M17
FR-7	M11, M12
FR-8	M12
FR-9	M12
FR-10	M17
FR-11	M17
FR-12	M16, M20
FR-13	M6, M20
FR-14	M13
FR-15	M17

Table 6: Trace Between Requirements and Modules

AC	Module(s)
AC1	M1
AC2	M9
AC4	M9
AC3	M3
AC4	M3
AC5	M3
AC6	M6
AC7	M4
AC8	M3
AC9	M20
AC10	M9

Table 7: Trace Between Anticipated Changes and Modules

8 Use Hierarchy Between Modules

This section provides a uses hierarchy between modules that have been described in detail above. As Parnas (1978) said, of two programs A and B, that A *uses* B if correct execution of B may be necessary for A to complete the task described in its specification. That is, A *uses* B if there exist situations in which the correct functioning of A depends upon the availability of a correct implementation of B.

Figure 2 illustrates the use relation between the modules. It can be seen that the graph is a directed acyclic graph (DAG). Each level of the hierarchy offers a testable and usable subset of the system, and modules in the higher level of the hierarchy are essentially simpler because they use modules from the lower levels.

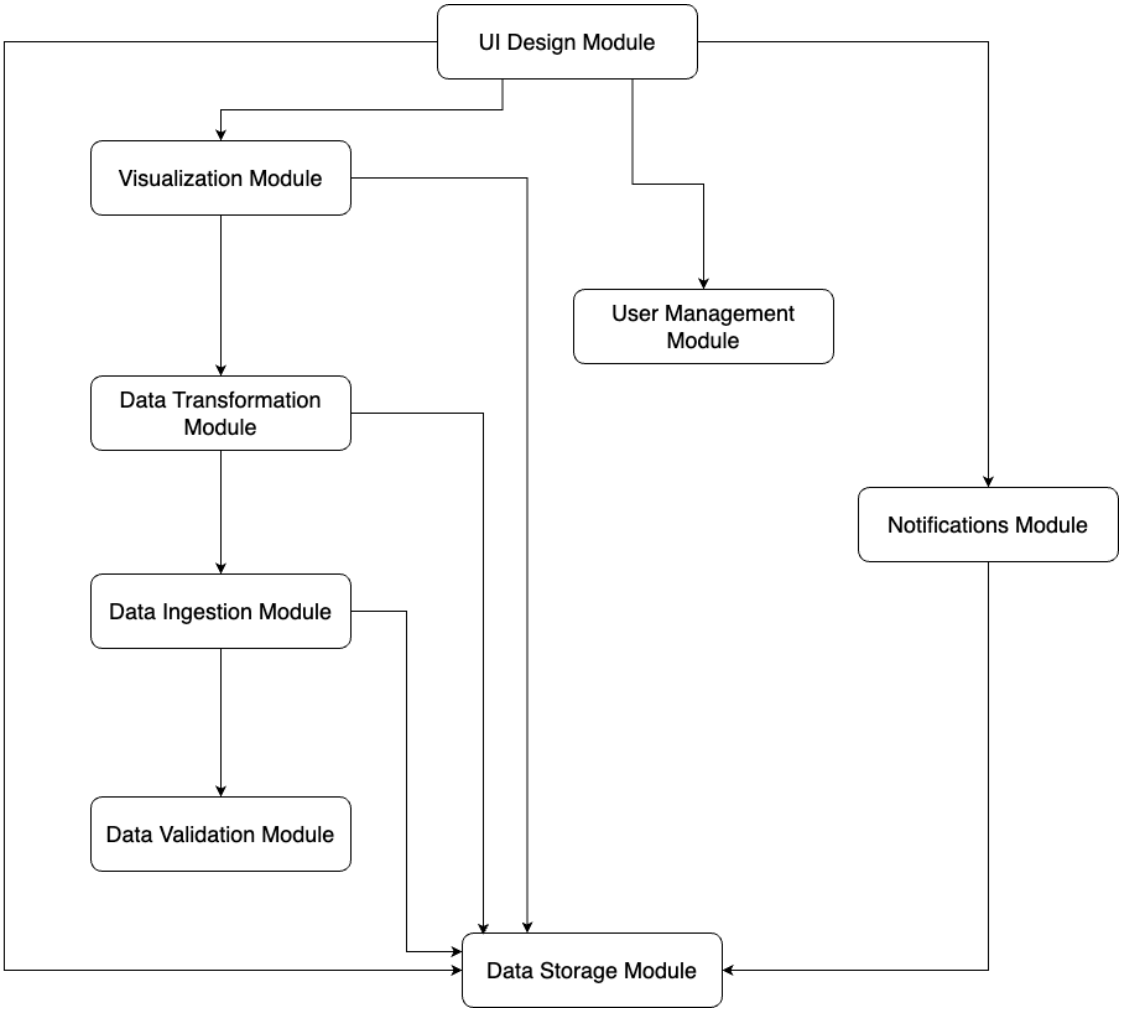

Figure 2: Use hierarchy among modules

The diagram shows the following:

1. The UI Design Module (M11) is at the highest level and uses multiple modules, namely,

the Data Storage Module (M6), the Visualization Module (M12), the User Management Module (M13) and the Notifications Modules (M20).

2. The Visualization Module (M12) uses the Data Transformation Module (M17) and the Data Storage Module (M6).

3. The Data Ingestion Module (M15) uses the Data Validation Module (M16) and the Data Storage Module (M6).

4. The Data Transformation Module (M17) uses the Data Ingestion Module (M15) and the Data Storage Module (M6).

5. The Notifications Module (M20) uses the Data Storage Module (M6).

6. The Data Storage Module (M6) appears at the lowest level as it is used by many other modules but does not depend on any other modules itself.

9 User Interfaces

The following figures encompass key design mock-ups of the application interface, created using Figma. These mock-ups are merely wireframes and do not represent the final interface design of the application.

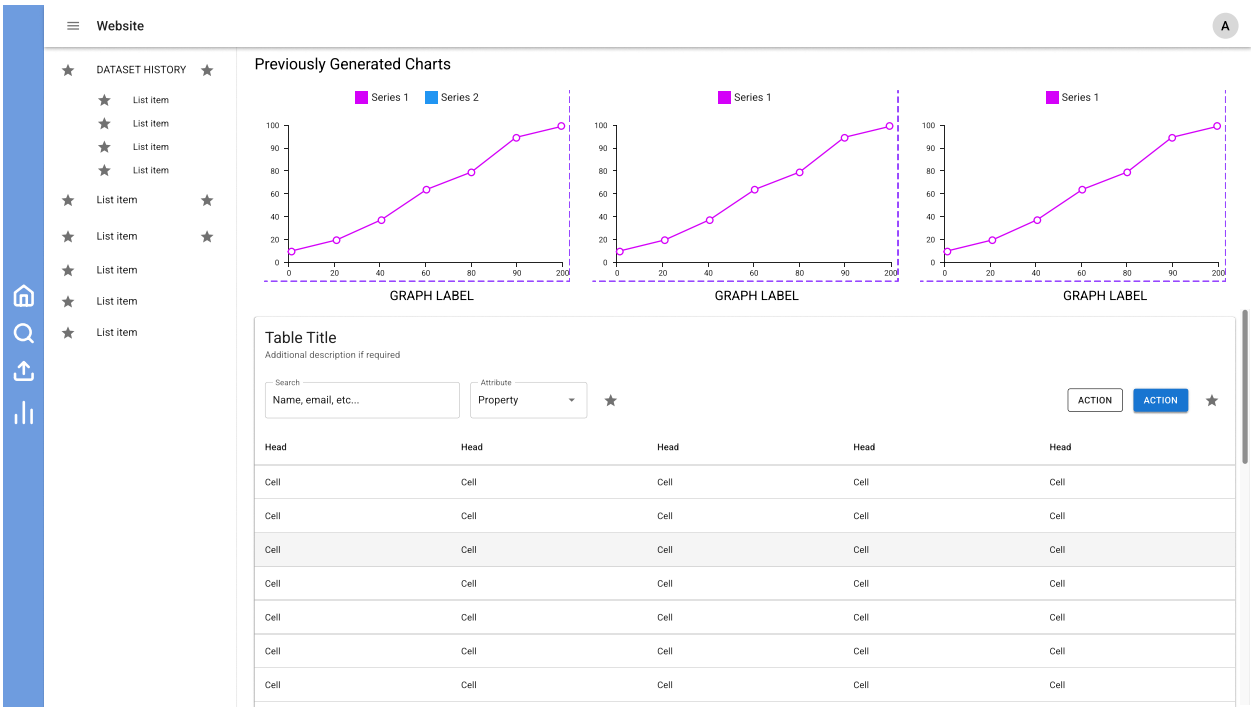

Figure 3: The dashboard, or main home page of the application upon successful login.

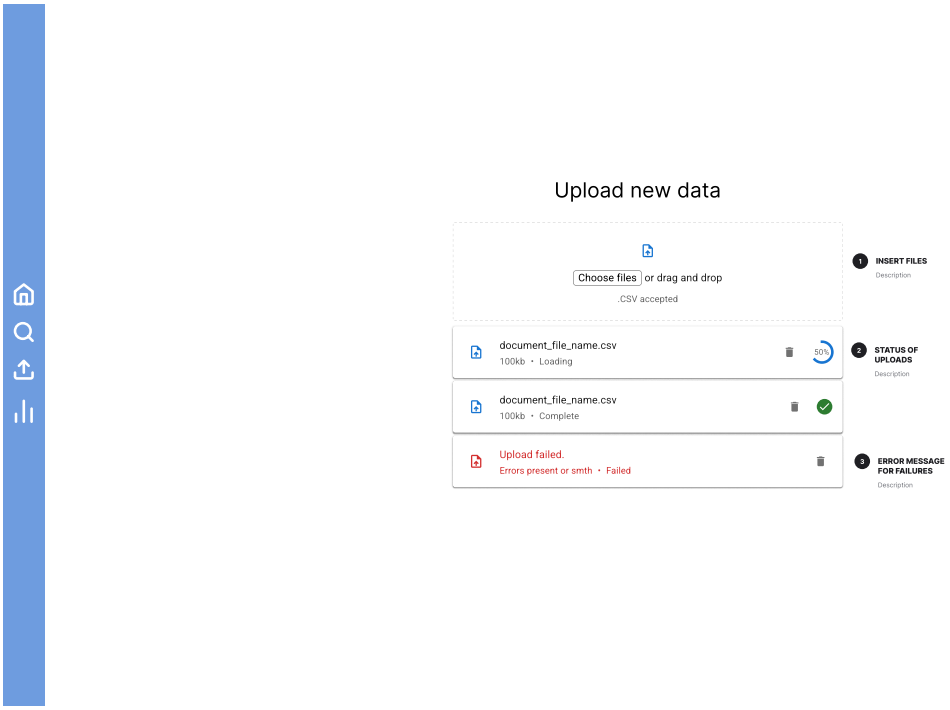

Figure 4: The upload page for users to upload new experiment data files.

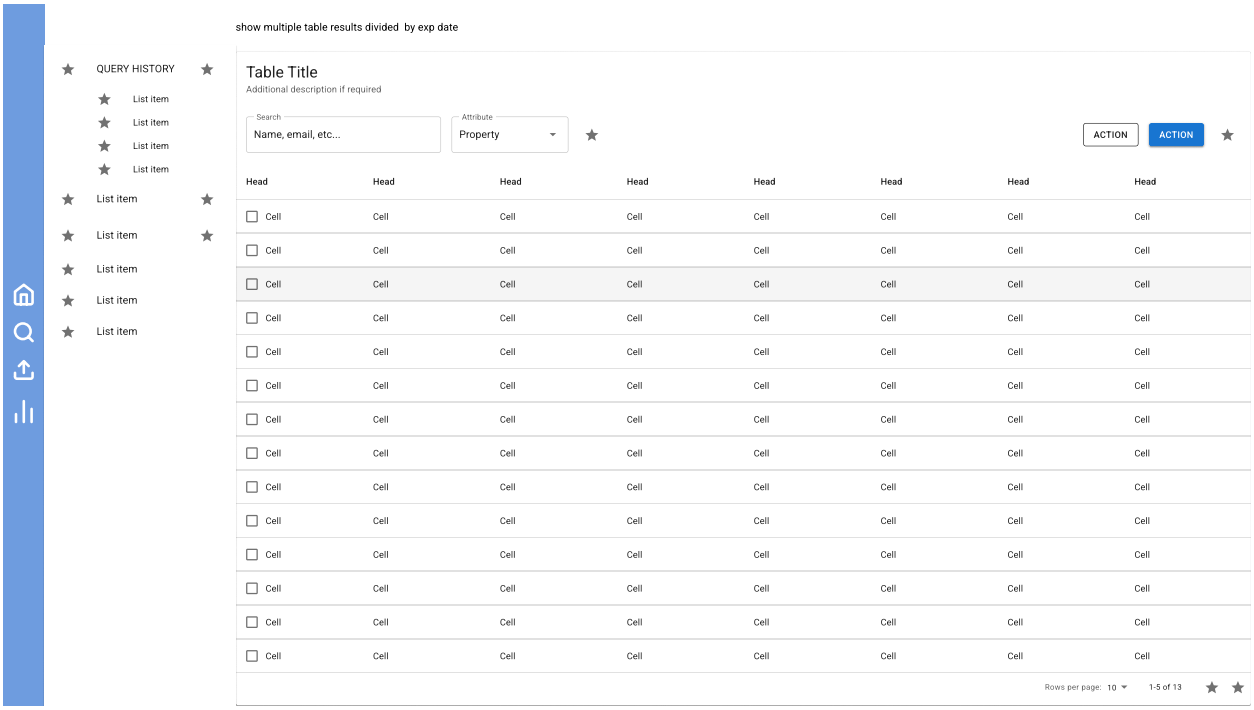

Figure 5: The main view page for querying and modifying data.

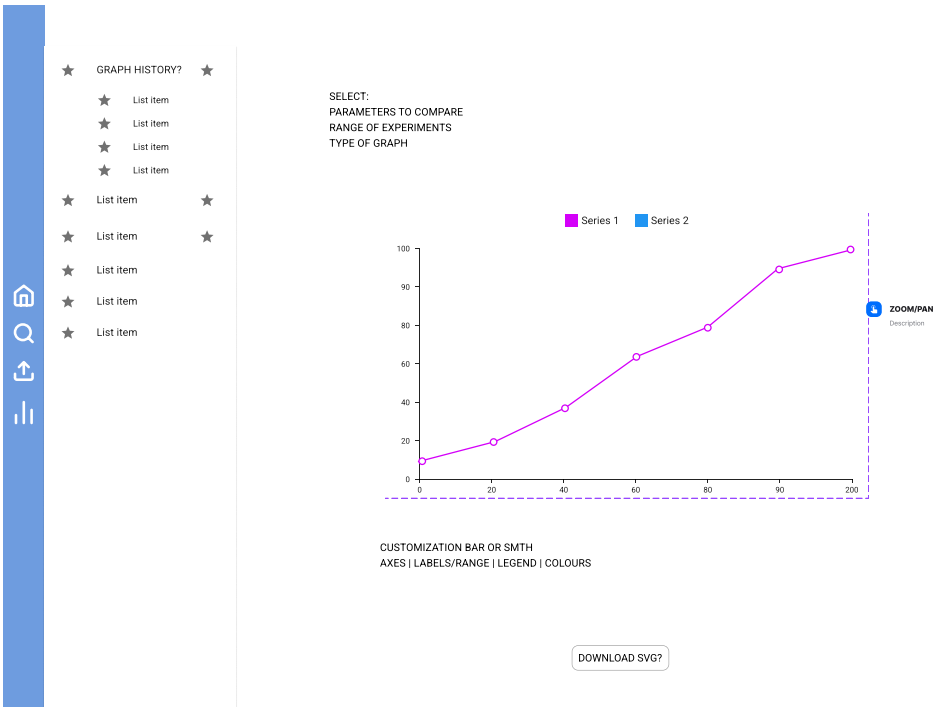

Figure 6: The main view page for generating and viewing visualizations of queried data.

10 Timeline

10.1 Phase 1: Initial Setup and Core Architecture

Task	**Module(s)**	**Status**	**Responsible**
Setting up a skeletal database.	M6	Done	Kate
Creating algorithm to convert CSV files to JSON.	M15	Done	Jason
Migrating all data to the database.	M15, M6	Done	Jason
Testing the database and algorithm.	-	Done	Jennifer, Sumanya
Setting up a CI/CD Infrastructure.	-	Done	Kate, Sumanya

Table 8: Phase 1 Timeline

10.2 Phase 2: Front End and User Roles

Task	Module(s)	Status	Responsible
Drawing wireframes and finalizing design elements.	M11	Done	All
Making a login page.	M11, M13	Weeks 14-16	Kate
Creating the upload screen.	M11, M15	Weeks 14-16	Jason
Building key frontend components including navigation and basic layouts.	M11	Weeks 14-16	Jennifer
Develop user login, registration and session management.	M13	Weeks 15-17	Kate

Table 9: Phase 2 Timeline

10.3 Phase 3: Data Ingestion Pipeline and Core Analytics Engine

Task	Module(s)	Status	Responsible
Implementing the pipeline to handle data import, transformation and storage.	M15, M17, M6	Weeks 16-17	Jason, Sumanya
Migrate basic analytical functionalities.	M17	Weeks 15-17	Sumanya
Basic testing to ensure data integrity.	M16	Week 15	Jennifer
Implementing data validation rules.	M16, M17	Weeks 15-17	Kate

Table 10: Phase 3 Timeline

10.4 Phase 4: Setting-up Dashboards and Connecting Components

Task	Module(s)	Status	Responsible
Use processed and transformed data to generate graphs.	M17, M12	Week 18	Jennifer, Kate
Connect frontend, backend and data modules via APIs.	-	Week 18	Jason
Integration testing to ensure smooth system operation.	-	Week 18	Sumanya

Table 11: Phase 4 Timeline

References

David L. Parnas. Designing software for ease of extension and contraction. In *ICSE '78: Proceedings of the 3rd international conference on Software engineering*, pages 264–277, Piscataway, NJ, USA, 1978. IEEE Press. ISBN none.

D.L. Parnas, P.C. Clement, and D. M. Weiss. The modular structure of complex systems. In *International Conference on Software Engineering*, pages 408–419, 1984.

Appendix — Reflection

The information in this section will be used to evaluate the team members on the graduate attribute of Problem Analysis and Design.

1. What went well while writing this deliverable?
2. What pain points did you experience during this deliverable, and how did you resolve them?
3. Which of your design decisions stemmed from speaking to your client(s) or a proxy (e.g. your peers, stakeholders, potential users)? For those that were not, why, and where did they come from?
4. While creating the design doc, what parts of your other documents (e.g. requirements, hazard analysis, etc), it any, needed to be changed, and why?
5. What are the limitations of your solution? Put another way, given unlimited resources, what could you do to make the project better? (LO_ProbSolutions)
6. Give a brief overview of other design solutions you considered. What are the benefits and tradeoffs of those other designs compared with the chosen design? From all the potential options, why did you select the documented design? (LO_Explores)