

Module Guide for Software Engineering

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

January 17, 2025

1 Revision History

Date	Version	Notes
11 January 2025	1.0	Added initial content for Rev 0.

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

2.2 Notation

Contents

1 Revision History	i
2 Reference Material	ii
2.1 Abbreviations and Acronyms	ii
2.2 Notation	ii
3 Introduction	1
3.1 Summary	1
3.2 Purpose	1
4 Anticipated and Unlikely Changes	1
4.1 Anticipated Changes	1
4.2 Unlikely Changes	2
5 Module Hierarchy	3
6 Module Decomposition	5
6.1 Data Storage Module (M6)	5
6.1.1 Uses	6
6.1.2 Syntax	6
6.1.3 Semantics	6
6.2 Data Ingestion Module (M15)	8
6.2.1 Uses	8
6.2.2 Syntax	8
6.2.3 Semantics	9
6.3 Data Validation Module (M16)	11
6.3.1 Uses	11
6.3.2 Syntax	11
6.3.3 Semantics	12
6.4 Data Transformation Module (M17)	13
6.4.1 Uses	13
6.4.2 Syntax	14
6.4.3 Semantics	14
6.5 Notifications Module (M20)	15
6.5.1 Uses	15
6.5.2 Syntax	15
6.5.3 Semantics	15
6.6 UI Design Module (M11)	16
6.6.1 Uses	16
6.6.2 Syntax	16
6.6.3 Semantics	16
6.7 Visualization Module (M12)	17
6.7.1 Uses	17
6.7.2 Syntax	17

6.7.3	Semantics	17
6.8	User Management Module (M13)	17
6.8.1	Uses	18
6.8.2	Syntax	18
6.8.3	Semantics	18
7	Traceability Matrix	19
8	Use Hierarchy Between Modules	20
9	User Interfaces	21
10	Timeline	21

List of Tables

1	Module Hierarchy	4
2	Exported Access Programs for the Data Storage Module	6
3	Exported Access Programs for the Data Ingestion Module	9
4	Exported Access Programs for the UI Design Module	16
5	Exported Access Programs for the User Management Module	18
6	Trace Between Requirements and Modules	20
7	Trace Between Anticipated Changes and Modules	20

List of Figures

1	A DAG representing the implemented module hierarchy of Alkalytics.	5
2	Use hierarchy among modules	21
3	The dashboard, or main home page of the application upon successful login.	21
4	The upload page for users to upload new experiment data files.	22
5	The main view page for querying and modifying data.	22
6	The main view page for generating and viewing visualizations of queried data.	23

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.

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 [Software Requirements Specification \(SRS\)](#) 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 impact.
- 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 be 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 assumed 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 between 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

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

Table 1: Module Hierarchy

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.

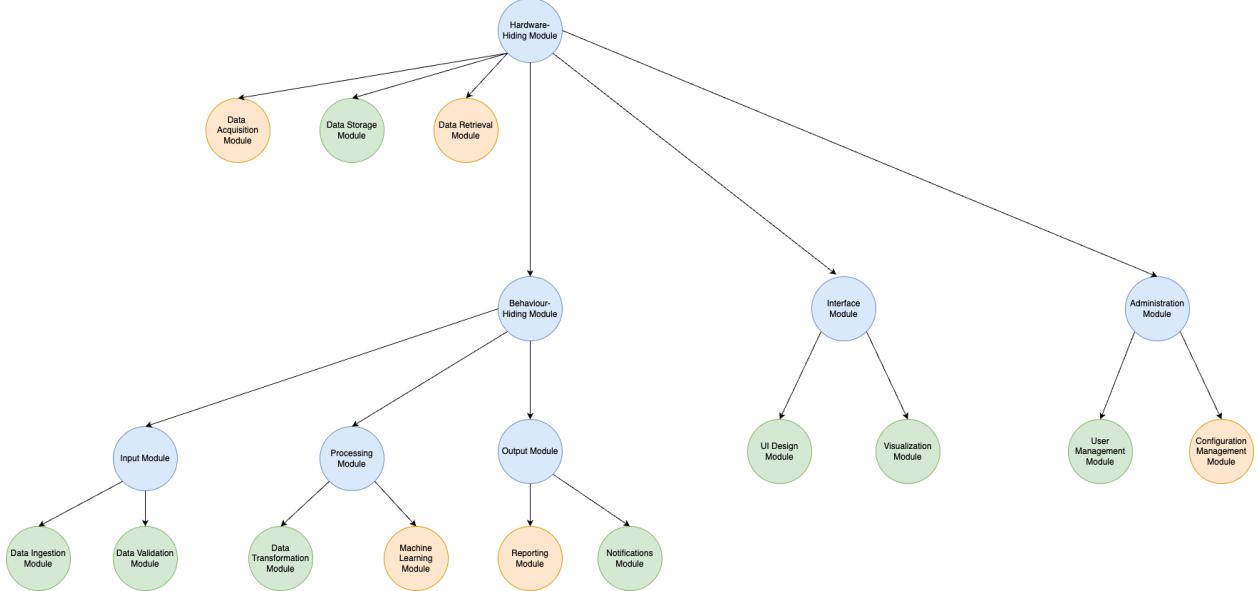


Figure 1: A DAG representing the implemented module hierarchy of Alkalytics.

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 to 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
<code>initializeDB</code>	-	Boolean	Throws <code>DatabaseInitializationError</code> if the database connection fails.
<code>uploadData</code>	<code>filePath(string)</code>	Boolean	Throws <code>FileNotFoundException</code> if the file does not exist, or <code>DataTransformationError</code> if the Transformation fails.
<code>queryData</code>	<code>query(string)</code>	Data (JSON)	Throws <code>QueryExecutionError</code> if the query is invalid.
<code>updateData</code>	<code>query(string), updates</code>	Boolean	Throws <code>UpdateError</code> 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 `FileNotFoundException` 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 <code>IngestionError</code> if the process fails.
ingestData	sourcePath(string)	Boolean	Throws <code>FileNotFoundException</code> if the file does not exist, or <code>DataTransformationError</code> if the Ingestion fails.
validateData	data(Object)	Boolean	Throws <code>DataValidationException</code> if the data fails validation.
retrieveAPIData	apiEndPoint(string)	Data (JSON)	Throws <code>APIRequestError</code> if the API request fails.
processData	rawData(Object)	Object	Throws <code>DataProcessingError</code> 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 `FileNotFoundException` 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 reliability 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) → Boolean`

`validate_current(i:Real) → Boolean`

`validate_flow_rate(f:Real) → 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 $\text{MIN_VOLTAGE} \leq v \leq \text{MAX_VOLTAGE}$, return True.
- Else, add **Voltage_Error** to **error_flags** and return False.

```
validate_current(i:Real):
```

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

```
validate_flow_rate(f:Real):
```

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

```
validate_ph(p:Real):
```

- If $\text{PH_RANGE}[0] \leq p \leq \text{PH_RANGE}[1]$, 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]) → 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 module 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 involve 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: The data structure and algorithm used to implement the virtual hardware.

Services: Serves as a virtual hardware used by the rest of the system. This module provides the interface between the hardware and the software. So, the system can use it to display outputs or to accept inputs.

Implemented By: OS

Rationale:

6.5.1 Uses

6.5.2 Syntax

Exported Constants:

Exported Access Programs:

6.5.3 Semantics

State Variables:

Environment Variables:

Assumptions:

Access Routine Semantics:

Local Function:

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 <code>RenderError</code> if UI rendering fails.
updateUI	component	-	Throws <code>UpdateError</code> 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: The data structure and algorithm used to implement the virtual hardware.

Services: Serves as a virtual hardware used by the rest of the system. This module provides the interface between the hardware and the software. So, the system can use it to display outputs or to accept inputs.

Implemented By: OS

Rationale:

6.7.1 Uses

6.7.2 Syntax

Exported Constants:

Exported Access Programs:

6.7.3 Semantics

State Variables:

Environment Variables:

Assumptions:

Access Routine Semantics:

Local Function:

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
<code>createUser</code>	<code>email(string), pass- word(string), role(string)</code>	<code>dict</code>	Throws <code>UserAlreadyExistsError</code> if the user already exists.
<code>validateUser</code>	<code>email(string), pass- word(string)</code>	<code>dict</code>	Throws <code>UserNotFoundError</code> if the user does not exist or <code>IncorrectPasswordError</code> if the password is incorrect.
<code>getUserAndRole</code>	<code>email(string)</code>	<code>dict</code>	Throws <code>UserNotFoundError</code> 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.	Modules
R1	M1, M??, M??, M??
R2	M??, M??
R3	M??
R4	M??, M??
R5	M??, M??, M??, M??, M??, M??
R6	M??, M??, M??, M??, M??, M??
R7	M??, M??, M??, M??, M??
R8	M??, M??, M??, M??, M??
R9	M??
R10	M??, M??, M??
R11	M??, M??, M??, M??

Table 6: Trace Between Requirements and Modules

AC	Modules
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

In this section, the uses hierarchy between modules is provided. 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.

[The uses relation is not a data flow diagram. In the code there will often be an import statement in module A when it directly uses module B. Module B provides the services that module A needs. The code for module A needs to be able to see these services (hence the import statement). Since the uses relation is transitive, there is a use relation without an import, but the arrows in the diagram typically correspond to the presence of import statement. —SS]

[If module A uses module B, the arrow is directed from A to B. —SS]

Figure 2: Use hierarchy among modules

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.

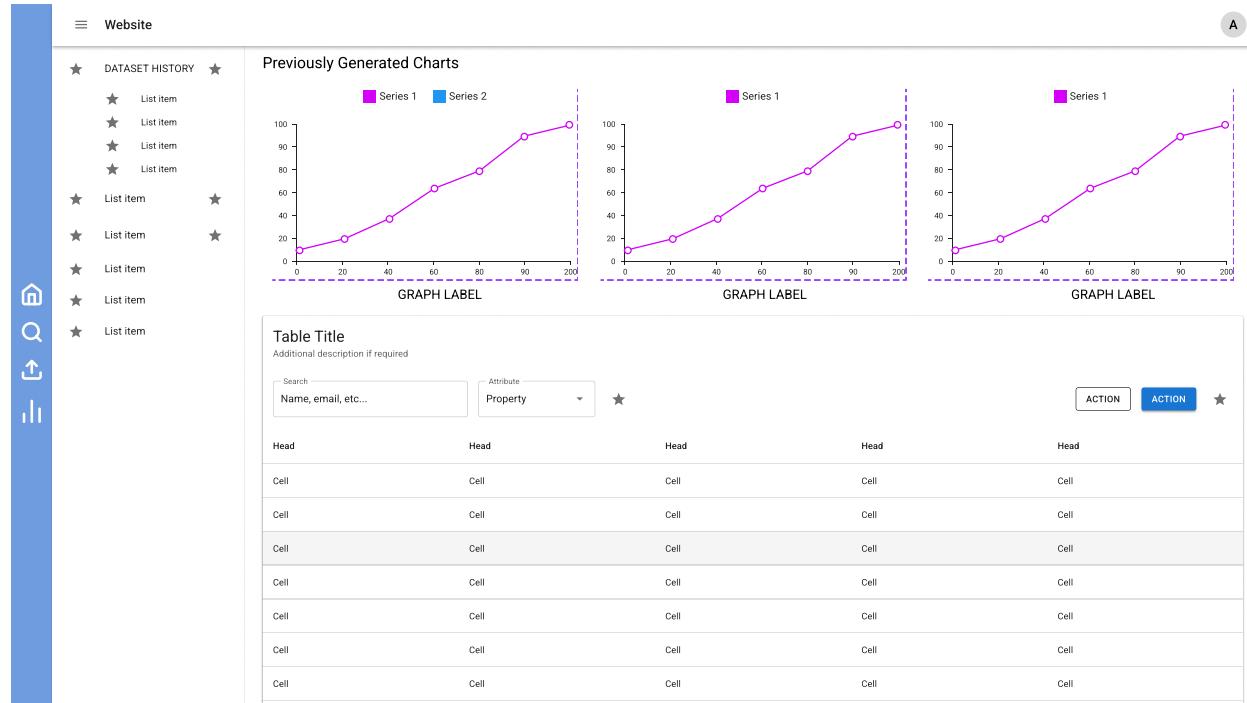


Figure 3: The dashboard, or main home page of the application upon successful login.

10 Timeline

[Schedule of tasks and who is responsible —SS]

[You can point to GitHub if this information is included there —SS]

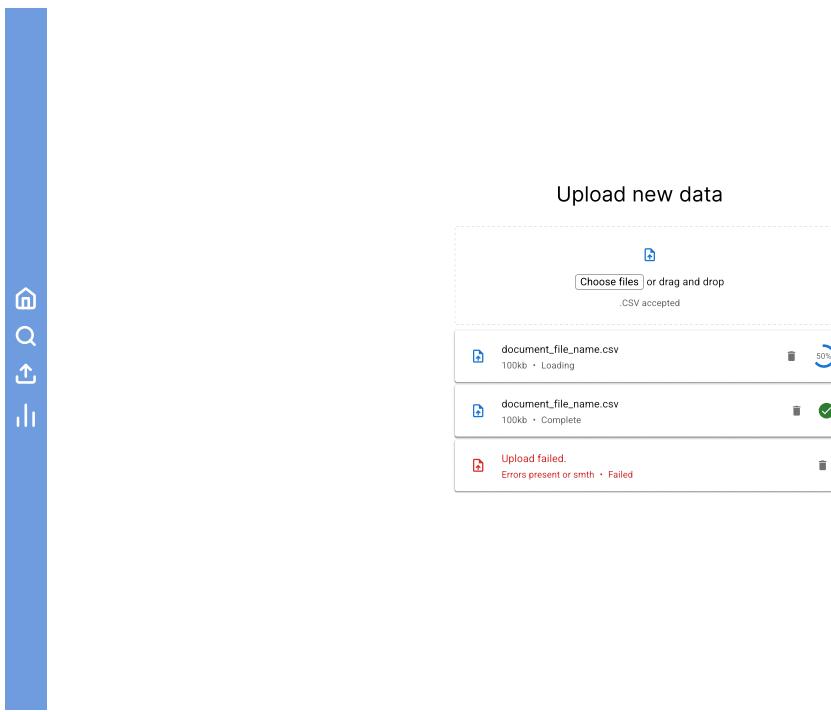


Figure 4: The upload page for users to upload new experiment data files.

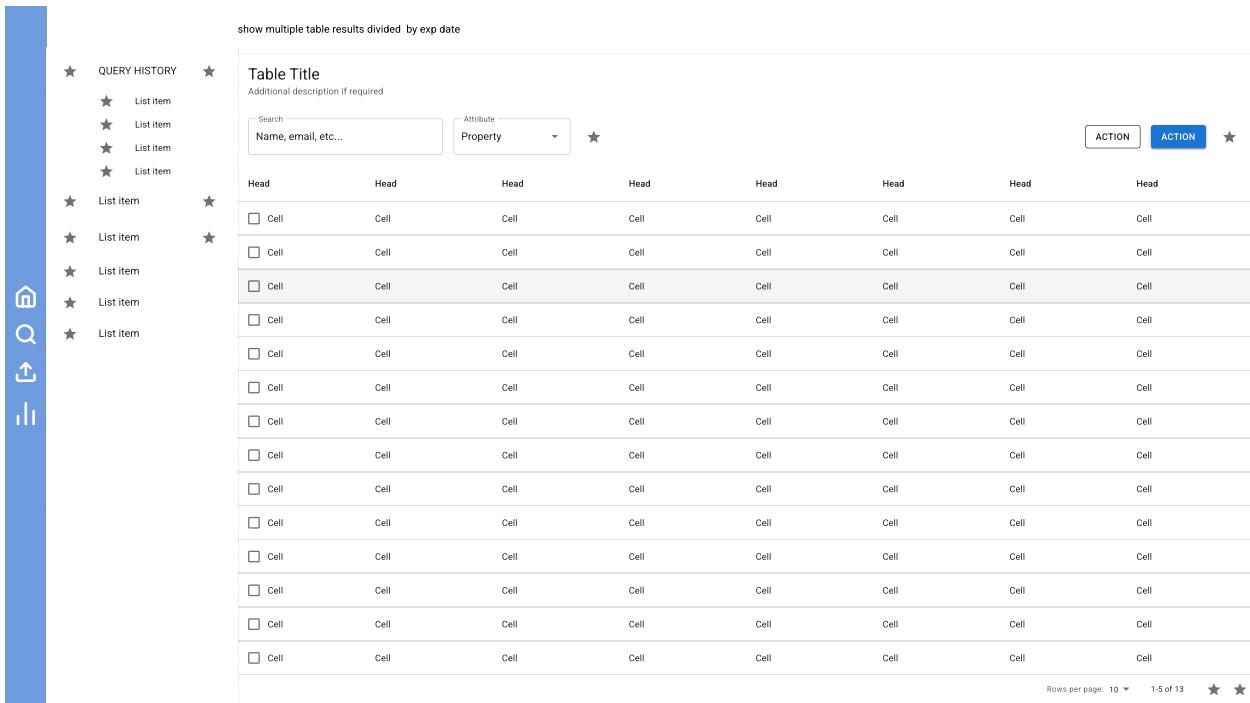


Figure 5: The main view page for querying and modifying data.

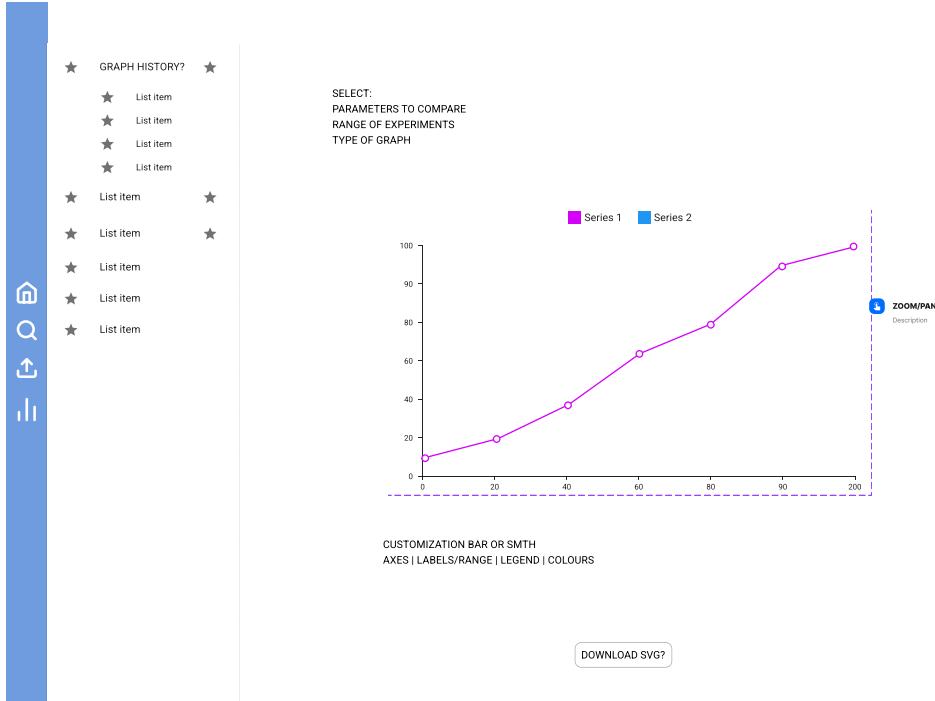


Figure 6: The main view page for generating and viewing visualizations of queried data.

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.