# Group 17

# DriftMark AI

# Software Requirements Specification

# For

# Comparative Evaluation of Data Drift Detection Algorithms: A Benchmarking Study

**Version 1.0**

Mentor : Dr. Uthayasanker Thayasivam

Team members: 210386A - Mendis P.H.M.N

210401T - Nadawa R.M.N

210404F - Nanayakkara A.H.M

# Revision History

| Date | Version | Description | Author |
| --- | --- | --- | --- |
| | | | |
| | | | |
| | | | |
| | | | |

# Table of Contents

# Software Requirements Specification

## 1. Introduction

### 1.1 Purpose

The purpose of this document is to build a comprehensive benchmarking system for evaluating concept drift detection algorithms tailored for regression datasets. This system aims to provide a structured and automated approach to assess the performance of various concept drift detection algorithms in dynamic data environments, ensuring the robustness and reliability of regression models. By facilitating the identification and comparison of key algorithms, this project will help organizations enhance the accuracy and trustworthiness of their predictive models, ultimately leading to better decision-making and increased confidence in AI systems.

### 1.2 Scope

The purpose of the benchmarking system for evaluating concept drift detection algorithms is to enhance the robustness and reliability of regression models in dynamic data environments. This system is designed to provide a comprehensive and automated approach to evaluate the performance of various concept drift detection algorithms, tailored specifically for regression datasets. Above all, we hope to build a dashboard where we can publicly showcase the results, providing valuable insights and fostering transparency in the evaluation process.

### 1.3 Definitions, Acronyms, and Abbreviations

**Concept Drift**: Changes in the statistical properties of a target variable over time.

**Drift Detection Algorithm**: Algorithms designed to identify changes in data distribution or statistical properties over time.

**Benchmarking**: The process of comparing the performance of various algorithms or models to determine their effectiveness and efficiency.

**SRS**: Software Requirements Specification

**AI**: Artificial Intelligence

**DDM**: Drift Detection Method

**EDDM**: Early Drift Detection Method

**ADWIN**: Adaptive Windowing

**ML**: Machine Learning

**API**: Application Programming Interface

**UI**: User Interface

**PEP**: Python Enterprise Proposal

**MTBF**: Mean Time Between Failures

**MTTR**: Mean Time To Repair

### 1.4 References

1. Thomas Lambart. "Concept Drift Detection: An Overview." Medium. Available at: https://medium.com/@thomaslambart/concept-drift-detection-an-overview-d087feea9676. Accessed on 26 June 2024.
2. DataCamp. "Understanding Data Drift & Model Drift." DataCamp Tutorial. Available at: https://www.datacamp.com/tutorial/understanding-data-drift-model-drift. Accessed on 27 June 2024.
3. Jie Lu, Anjin Liu, Fan Dong, Feng Gu, Joao Gama, and Guangquan Zhang. "Learning under Concept Drift: A Review." IEEE Transactions on Knowledge and Data Engineering. Accessed on 29 June 2024.
4. Lucas Baier, Marcel Hofmann, Niklas Kühl, Marisa Mohr, and Gerhard Satzger. "Handling Concept Drifts in Regression Problems – the Error Intersection Approach." Accessed on 1 July 2024.
5. Marília Lima, Manoel Neto, Telmo Silva Filho, and Roberta A. de A. Fagundes. "Learning Under Concept Drift for Regression—A Systematic Literature Review." IEEE Transactions on Neural Networks and Learning Systems. Accessed on 6 July 2024.
6. Frouros library. GitHub repository. Available at: https://github.com/IFCA-Advanced-Computing/frouros. Accessed on 8 July 2024.
7. DeepChecks package. GitHub repository. Available at: https://github.com/deepchecks/deepchecks. Accessed on 8 July 2024.

### 1.5 Overview

This document contains the detailed requirements for the benchmarking system for concept drift detection in regression datasets. It includes sections on the purpose and scope of the project, definitions and acronyms, and references. The subsequent sections describe the specific requirements, design constraints, and the functionality of the system.

## 2. Overall Description

**2.1 Product Perspective:**

The benchmarking system exists within the domain of machine learning and data science, specifically focusing on evaluating concept drift detection algorithms for regression models. It interfaces with data sources, algorithm implementations, and evaluation metrics to assess algorithmic effectiveness over time.

**2.2 Product Functions:**

The primary function of the system is to automate the evaluation of concept drift detection algorithms. This includes data ingestion, algorithm implementation, benchmarking processes, and visualizing results through a user-friendly dashboard.

**2.3 User Characteristics:**

The system targets users with backgrounds in data science, machine learning, and algorithm evaluation. Users include data scientists, machine learning engineers, researchers, and organizational stakeholders involved in maintaining model accuracy and reliability in dynamic data environments.

**2.4 Constraints:**

Design constraints include compatibility requirements with existing data formats, scalability to handle large

datasets and adherence to regulatory standards governing data privacy and algorithmic evaluation.

**2.5 Assumptions and Dependencies:**

The system assumes the availability of reliable data sources for both real and synthetic datasets, access to computational resources for algorithm implementation and evaluation, and dependencies on external libraries and frameworks for algorithmic implementation and data visualization.

**2.6 Requirements Subsets:**

Detailed requirements subsets include functional aspects such as data ingestion methods, algorithm integration protocols, and dashboard visualization requirements. Nonfunctional subsets cover performance metrics, usability standards, reliability criteria, scalability expectations, and security measures necessary for safeguarding data integrity and user access.

# 3. Specific Requirements

This section outlines the detailed software requirements for the concept drift detection system, enabling designers to create a system that meets these requirements and allowing testers to verify that the system functions correctly. (See Appendix A for the use case diagram.)

## 3.1 Functionality

This section describes the functional requirements of the system in detail.

### 3.1.1 Data Ingestion

**Description**: The system must be capable of ingesting real-world datasets. It should support various data formats such as CSV, JSON, and Excel.
**Inputs**: File upload or direct connection to data sources.
**Processing**: Validate data format and structure, handle missing values, and standardize data.
**Outputs**: A cleaned and standardized dataset ready for drift detection analysis.

### 3.1.2 Concept Drift Detection

**Description**: Implement multiple drift detection algorithms such as DDM, EDDM, ADWIN, Page Hinkley, and others as identified in the benchmarking study.
**Inputs**: Standardized dataset from the data ingestion module.
**Processing**: Apply selected drift detection algorithms to the dataset.
**Outputs**: Drift detection results indicating if and where drift has occurred in the dataset.

### 3.1.3 Dashboard Interface

**Description**: Develop a user-friendly dashboard to visualize the results of the drift detection algorithms.
**Inputs**: Drift detection results.
**Processing**: Generate visualizations like line charts, bar charts, and heatmaps to represent the drift detection findings.
**Outputs**: Interactive visualizations displayed on the dashboard.

### 3.1.4 User Management

**Description**: Provide a user management system to handle multiple users with different roles and

permissions.
**Inputs**: User data (username, password, role).
**Processing**: Authenticate users, assign roles, and manage access permissions.
**Outputs**: Secure access to the system based on user roles.

## 3.2 Usability

This section outlines the requirements that will ensure the system is user-friendly and easy to use, helping users to become productive quickly and efficiently.

### 3.2.1 Training Time

Specify the required training time for normal users and power users to become productive at particular operations.

- **Normal Users**: Users should be able to perform basic operations such as viewing data visualizations and receiving alerts within 2 hours of training.
- **Power Users**: Users who need to interact with more advanced features, such as configuring algorithms and generating custom reports, should be proficient within 4 hours of training.

### 3.2.2 User Interface Design

The user interface should be designed to be intuitive and easy to navigate.

- **Intuitive Layout**: Use a clean and organized layout with clearly labeled sections and intuitive navigation.
- **Interactive Elements**: Include interactive elements such as tooltips, hover effects, and progress indicators to enhance the user experience.
- **Feedback Mechanisms**: Provide immediate feedback for user actions, such as confirmation messages for data uploads and visual cues for processing status.

### 3.2.3 Usability Standards Compliance

The system should conform to common usability standards to ensure a consistent and intuitive user experience.

- **Compliance Standards**: Follow IBM's CUA (Common User Access) standards and Microsoft's GUI (Graphical User Interface) standards.
- **Consistency**: Ensure a consistent look and feel across all components of the system, including color schemes, button placements, and font styles.
- **Accessibility**: Implement accessibility features to support users with disabilities, such as keyboard navigation, screen reader compatibility, and high-contrast modes.

### 3.2.4 Customizability

Allow users to customize their experience based on their preferences and needs.

- **Customizable Dashboards**: Enable users to customize their dashboard layout, including the arrangement of widgets and selection of data visualizations.
- **Personal Settings**: Allow users to save personal settings, such as notification preferences and default data views.
- **Themes and Modes**: Offer multiple themes and display modes, including light and dark modes, to cater to different user preferences.

## 3.3 Reliability

This section specifies requirements for the reliability of the system.

### 3.3.1 Availability

The system should be available 99.95% of the time, which translates to approximately 4.38 hours of downtime per year. The system should be operational 24/7, with scheduled maintenance windows occurring monthly between 2 AM and 4 AM to minimize impact on users.
**Degraded Mode Operations**: In case of partial failures, the system should maintain essential functionalities such as data ingestion and basic drift detection while displaying a notification of limited functionality to the user.
**Measurement**: System availability should be monitored continuously, and uptime reports should be generated monthly.

### 3.3.2 Mean Time Between Failures (MTBF)

The system should have an MTBF of at least 5,000 hours, indicating high reliability.
**Measurement**: System logs and failure reports should be analyzed regularly to calculate the MTBF.

### 3.3.3 Mean Time To Repair (MTTR)

The system should have a maximum MTTR of 2 hours, ensuring quick recovery from failures.
**Measurement**: Track the duration of all system outages from detection to resolution to ensure MTTR is within the specified limit.

### 3.3.4 Accuracy

The system should provide drift detection results with a precision of 95% and accuracy by some known standard.
**Measurement**: Validate the accuracy of drift detection algorithms using benchmark datasets and compare results against known standards.

### 3.3.5 Maximum Bugs or Defect Rate

The system should have no more than 0.5 bugs per 1,000 lines of code (0.5 bugs/KLOC).
**Measurement**: Conduct regular code reviews and use automated testing tools to track the defect rate. Bug tracking systems should be used to log and categorize defects.

### 3.3.6 Bugs or Defect Rate Categorization

**Minor Bugs**: Issues that do not affect system functionality or user experience significantly. The system should have no more than 10 minor bugs per release.
**Significant Bugs**: Issues that affect some functionalities but have workarounds. The system should have no more than 2 significant bugs per release.
**Critical Bugs**: Issues that result in complete loss of data or a complete inability to use certain parts of the system. The system should have zero critical bugs in any release.
**Measurement**: Implement a bug-tracking system to log, categorize, and track all detected bugs. Regularly review bug reports to ensure compliance with the defined thresholds.

## 3.4 Performance and Security

This section outlines the system's performance characteristics and security requirements.

### 3.4.1 Response Time

The system should have an average response time of 2 seconds for all transactions. The system should have a maximum response time of 5 seconds for all transactions.

### 3.4.2 Throughput

The system should be capable of handling up to 100 transactions per second (TPS).
**Measurement**: Load testing should be conducted to measure the system's throughput and ensure it meets the specified requirement.

### 3.4.3 Capacity

The system should support up to 1,000 concurrent users.
**Measurement**: Stress testing should be conducted to measure the system's capacity and ensure it can accommodate the specified number of users.

### 3.4.4 Degradation Modes

In case of system degradation, essential functionalities such as data ingestion and basic drift detection should remain operational, while non-essential features may be temporarily disabled. The system should notify users of limited functionality.
**Measurement**: Conduct scenario-based testing to ensure the system can maintain essential operations during degradation.

### 3.4.5 Resource Utilization

**Memory Utilization**: The system should use no more than 75% of available memory under normal operating conditions.
**Disk Utilization**: The system should use no more than 70% of available disk space under normal operating conditions.
**Communications Utilization**: The system should use no more than 60% of available network bandwidth under normal operating conditions.
**Measurement**: Regular monitoring and logging of resource usage should be implemented to ensure compliance.

### 3.4.6 Data Encryption

All data in transit and at rest should be encrypted using industry-standard encryption protocols (e.g., TLS 1.2+ for data in transit and AES-256 for data at rest). Regular security audits should be conducted to ensure encryption protocols are properly implemented and maintained.

### 3.4.7 Access Control

Implement role-based access control (RBAC) to restrict access to sensitive data and functions. Users should be assigned roles with specific permissions based on their job requirements.

### 3.4.8 Data Integrity

Implement mechanisms to ensure data integrity, such as checksums and hashing. The system should detect and report any data corruption. Regular integrity checks should be conducted to ensure data remains

accurate and unaltered.

#### 3.4.9 Authentication and Authorization

The system should use multi-factor authentication (MFA) for user login and secure APIs with OAuth 2.0 or similar standards.

#### 3.4.10 Audit Logging

Implement comprehensive audit logging to track all user activities, system access, and data changes. Logs should be tamper-evident and stored securely.
**Measurement**: Regularly review audit logs to detect and investigate any suspicious activities.

### 3.5 Supportability

This section indicates requirements that will enhance the supportability or maintainability of the system.

#### 3.5.1 Coding Standards

Adhere to established coding standards to ensure code quality and consistency. For Python, follow PEP 8 guidelines, and for JavaScript, follow the AirBnB JavaScript Style Guide.

#### 3.5.2 Naming Conventions

Use clear and consistent naming conventions for variables, functions, classes, and other code elements. Naming conventions should be descriptive and follow camelCase for variables and functions, and PascalCase for classes.

#### 3.5.3 Class Libraries

Utilize well-documented and widely-supported class libraries and frameworks to ensure code reusability and maintainability. For data processing and machine learning, use libraries such as NumPy, Pandas, and Scikit-learn. For the benchmarking purpose use libraries such as Frouros, Scikit-Multiflow, and River.

#### 3.5.4 Documentation

Provide comprehensive and up-to-date documentation for the system, including API documentation, user manuals, and developer guides. Documentation should cover installation, configuration, usage, and troubleshooting.

#### 3.5.5 Maintenance Access

Design the system to allow easy access for maintenance tasks, such as updating configurations, monitoring system health, and performing backups. Provide administrative interfaces and tools to facilitate maintenance activities.

#### 3.5.6 Maintenance Utilities

Implement utilities and tools to support system maintenance, such as automated backup solutions, monitoring tools, and logging systems. Ensure these utilities are integrated seamlessly into the system.

#### 3.5.7 Issue Tracking

Utilize an issue tracking system (e.g.: GitHub Issues) to manage and prioritize bugs, feature requests, and maintenance tasks. Ensure that all issues are logged, categorized, and assigned appropriately.

### 3.6 Design Constraints

This section outlines the design constraints for the concept drift detection system, representing mandatory design decisions.

#### 3.6.1 Standards Compliance

Ensure the system complies with relevant standards to maintain quality, interoperability, and security.
**Coding Standards**: Follow PEP 8 for Python and Airbnb JavaScript Style Guide for JavaScript.
**Security Standards**: Implement security best practices and standards, such as OWASP Top Ten, to protect against common vulnerabilities.
**Data Standards**: Adhere to data formatting and exchange standards like JSON, CSV, and XML to ensure compatibility and ease of data integration.

#### 3.6.2 Hardware Limitations

Account for the hardware limitations that may impact the system's performance and scalability.
**Processing Power**: Optimize the system to run efficiently on servers with limited CPU and memory resources.
**Storage Capacity**: Ensure the system can handle large datasets while managing storage constraints effectively.
**Network Bandwidth**: Design the system to operate efficiently over networks with limited bandwidth, minimizing data transfer and optimizing communication protocols.

#### 3.6.3 Software Languages

Mandate the use of specific programming languages for different components of the system.
**Backend**: Use Python for data processing, and machine learning.
**Frontend**: Use JavaScript, specifically React, for building the user interface.
**Scripting and Automation**: Use shell scripting and Python for automation tasks.

#### 3.6.4 Software Process Requirements

Follow specific software development processes to ensure consistency and quality.
**Agile Methodology**: Use Agile practices, including Scrum or Kanban, for iterative development and continuous feedback.
**Version Control**: Use Git for version control, following a branching strategy such as Gitflow.
**Continuous Integration/Continuous Deployment (CI/CD)**: Implement CI/CD pipelines using tools like Jenkins, Travis CI, or GitHub Actions.

#### 3.6.5 Prescribed Use of Developmental Tools

Mandate the use of specific tools for development, testing, and deployment.
**Development Tools**: Use Google Colab and Jupyter Notebook for Python development and Visual Studio Code for frontend development.
**Testing Tools**: Use pytest for backend testing and Jest for frontend testing.

### 3.6.6 Architectural and Design Constraints

Define the architectural and design constraints that shape the system's structure and behavior.
**Modular Architecture**: Design the system with a modular architecture to facilitate independent development, testing, and deployment of components.
**Microservices**: Use a microservices architecture for the backend to ensure scalability and flexibility.
**APIs**: Design RESTful APIs for communication between the frontend and backend, ensuring clear and consistent interfaces.

### 3.6.7 Class Libraries

Specify the use of certain class libraries and frameworks to streamline development.
**Backend Libraries**: Use libraries such as NumPy, Pandas, Scikit-learn, Matplotlib, and TensorFlow for data preprocessing. Use Frouros for synthetic data generation, Scikit-Multiflow for importing concept drift detecting algorithms, and River for handling data streams. Utilize Flask or Django to build the backend web services
**Frontend Libraries**: Use React for building the user interface and D3.js for data visualization.

## 3.7 On-line User Documentation and Help System Requirements

This section describes the requirements for providing user documentation and help systems to assist users in understanding and using the concept drift detection system effectively.

### 3.7.1 Documentation Portal

Provides a comprehensive online documentation portal accessible through the dashboard.

- **User Guides**: Include step-by-step guides for setting up, configuring, and using the system.
- **API Documentation**: Detailed documentation of all API endpoints, including request and response formats, authentication methods, and example use cases.
- **FAQ Section**: A Frequently Asked Questions section to address common queries and issues.
- **Search Functionality**: Implement a search feature to help users quickly find relevant information.

### 3.7.2 Help System

Implement an integrated help system within the dashboard to assist users in real time.

- **Interactive Tutorials**: Develop interactive tutorials and walkthroughs to guide new users through key functionalities.
- **Live Chat Support**: Offer a live chat support option for users to get immediate assistance from support personnel.

## 3.8 Purchased Components

This section describes any purchased components to be used with the system, including any applicable licensing or usage restrictions, and any associated compatibility and interoperability or interface standards.

### 3.8.1 Cloud Services

Leverage cloud services for scalable data storage and processing.
**Service Example**: Amazon Web Services (AWS) or Google Cloud Platform (GCP)
**Licensing**: Comply with the cloud provider's licensing and usage policies, including costs associated with storage, computation, and data transfer.
**Interface Standards**: Ensure the system can interact with cloud services using standard APIs and protocols (e.g., REST, gRPC).

## 3.9 Interfaces

This section defines the interfaces that must be supported by the concept drift detection system. It includes user interfaces, hardware interfaces, software interfaces, and communication interfaces, ensuring the software can be developed and verified against the interface requirements.

### 3.9.1 *User Interfaces*

The user interfaces will provide a platform for users to interact with the system, perform data entry, view analysis results, and manage settings. See Appendix B to view a draft diagram (block diagram) showing some main interfaces required by the user.

**Login Page**:

**Functionalities**: User authentication via username and password, password reset option.
**Menu Items/Components**: Text boxes for username and password, 'Login' button, 'Forgot Password' link.

**Dashboard**:

**Functionalities**: Display an overview of key metrics and system status, access different modules.
**Menu Items/Components**: Panels for visualizations, navigation menu, notifications area, settings icon.

**Data Entry Page**:

**Functionalities**: Upload datasets, and input data manually.
**Menu Items/Components**: File upload button, text boxes for manual entry, dropdown lists for data categories, 'Submit' button.

**Analysis Page**:

**Functionalities**: Display analysis results, visualizations, and detected concept drifts.
**Menu Items/Components**: Graphs and charts, filters for date range and data type, 'Export' button, detailed results panel.

**Settings Page**:

**Functionalities**: Configure system settings, notification preferences, and user management.
**Menu Items/Components**: Toggle switches, dropdown lists, text boxes, 'Save' button.

### 3.9.2 *Hardware Interfaces*

- The software interacts with standard hardware components such as computers, tablets, and printers.
- For client-side, it needs a minimum of 500 MB free disk space, at least 4 GB of RAM, and a dual-core processor (e.g., Intel i5 or AMD equivalent) to manage data-intensive operations efficiently, ensuring compatibility with various memory configurations and avoiding performance degradation.
- Server-side requirements include at least 1 TB of storage, 16 GB of RAM, and a multi-core processor with at least 8 cores (e.g., Intel Xeon, AMD EPYC) to handle large datasets and concurrent users, leveraging RAID configurations and advanced memory management for high concurrency and low latency. Both client and server setups should support smooth data transfer, efficient task distribution, and high throughput, maintaining system responsiveness even under peak loads.

  **Expected Behavior**: The system should perform efficiently without significant lag or delays on machines meeting the above specifications.

### 3.9.3 *Software Interfaces*

- The system's software interfaces are designed to ensure seamless interaction with various components.
- The system will connect the frontend to backend services using HTTP/HTTPS protocols over standard ports (80/443), ensuring secure and efficient communication.
- It will interact with third-party web services through RESTful APIs with OAuth2 authentication for additional functionalities such as notifications.
- The system will support plugin integration via a standard plugin architecture with defined hooks and callbacks, enabling easy extension and customization.
- Additionally, it will ensure compatibility with other software components by adhering to common interface standards, facilitating interaction with external libraries, purchased components, and subsystems developed outside the SRS scope.

### 3.9.4 *Communications Interfaces*

- The concept drift detection system will utilize robust communication interfaces to ensure effective interaction with other systems and devices.
- It will employ asynchronous HTTP/HTTPS protocols for data exchange between the frontend and backend, supporting standard ports (80/443) for secure and efficient communication.
- For real-time updates and notifications, WebSockets will be used, maintaining persistent connections for instant data push from the server to clients.
- The system will also support FTP/SFTP protocols for secure file transfers, enabling efficient handling of large datasets.
- Furthermore, integration with local area networks (LAN) will facilitate smooth internal communications, while remote serial devices will be supported through standardized serial communication protocols, ensuring compatibility and reliability across various network configurations.

### 3.10 Database Requirements

The DriftMark AI system requires robust database support to store and manage large volumes of raw and processed data from datasets like the NYC taxi dataset. For efficient storage and retrieval of time-series data, NoSQL databases such as MongoDB or time-series databases like InfluxDB are recommended.

Relational databases like PostgreSQL or MySQL should be used for storing preprocessed data, algorithm outputs, drift results, user profiles, and preferences, ensuring data security and privacy.

### 3.11 Licensing, Legal, Copyright, and Other Notices

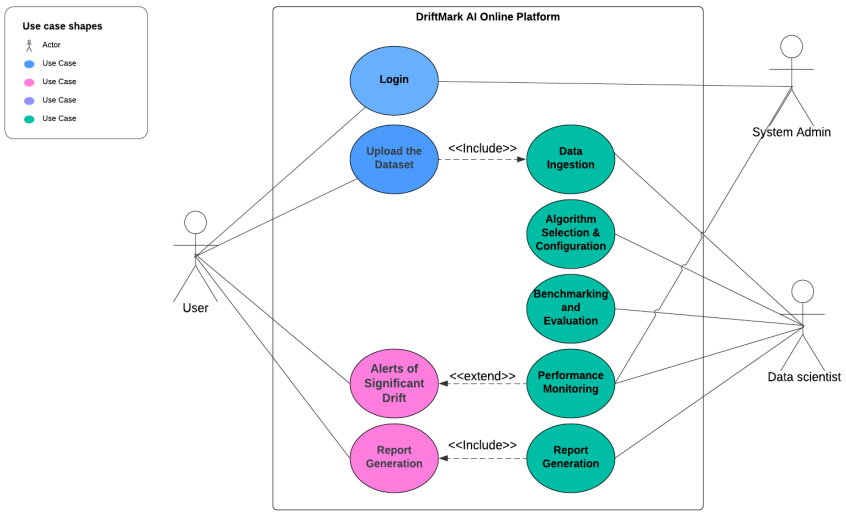
The system must comply with all relevant licensing and legal requirements, ensuring proper usage of third-party components and libraries. This includes adhering to open-source licenses (e.g., MIT, GPL) and commercial software licenses. The software should display legal disclaimers, warranties, and copyright notices as required. Any use of trademarks, wordmarks, or logos must comply with their respective guidelines and agreements. Additionally, the system should include patent notices where applicable and ensure that it does not infringe on any existing intellectual property rights.

### 3.12 Applicable Standards

The concept drift detection system will adhere to several applicable standards to ensure quality, usability, and compliance. This includes industry standards for software development, such as ISO/IEC 25010 for software quality and ISO/IEC 12207 for software lifecycle processes. Usability standards like ISO 9241 will be followed to enhance user experience, while interoperability standards like IEEE 829 will ensure smooth integration with other systems. The system will also comply with internationalization standards (e.g., ISO/IEC 15897) to support multiple languages and cultural conventions. Additionally, adherence to regulatory standards, such as GDPR for data protection and HIPAA for healthcare data, will be ensured to maintain legal compliance and user trust.

## 4. Supporting Information

### Appendix A: Use Case Diagram



**Appendix B: A draft diagram (block diagram) showing some main interfaces required by the user**