Group 17

DriftMark AI

Software Architecture Document

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
17/07/2024	1.o	Comparative Evaluation of Data Drift Detection Algorithms: A Benchmarking Study	Group 17

Table of Contents

Software Architecture Document

1. Introduction

1.1 Purpose

This document provides a comprehensive architectural overview of the system, using a number of different architectural views to depict different aspects of the system. It is intended to capture and convey the significant architectural decisions which have been made on the system.

1.2 Scope

The Software Architecture Document applies to the DriftMark AI platform, a system designed to benchmark and evaluate concept drift detection algorithms for regression datasets. The document encompasses all aspects of the system's architecture, including user management, data ingestion, algorithm selection and configuration, benchmarking, evaluation, performance monitoring, and notification systems. It influences the design, development, testing, and maintenance phases of the project, ensuring consistency and alignment with the architectural goals and constraints.

1.3 Definitions, Acronyms, and Abbreviation

SRS: Software Requirements Specification

SAD: Software Architecture Document

AI: Artificial Intelligence

ML: Machine Learning

CRUD: Create, Read, Update, Delete

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.
8. LucidChart.Official website.Available at:

https://www.googleadservices.com/pagead/aclk?sa=L&ai=DChcSEwjWwqCm562HAxXOpGYCHQY0BDsYABAAGgJzbQ&ase=2&gclid=CjwKCAjw1920BhA3EiwAJT3lSWVdofoLa5TeKnM_dnEmjgvbGMcom4D-1QmiGa9o8auoOgMJ7E3ioxoCcUEQAvD_BwE&ei=95SXZtnsA5u6seMP0ZS2gAo&ohost=www.google.com&cid=CAESVeD2PFU6hpaJn7mJW92DcS32gNp3dW7gl2qEes7fo1q3ley9wvYM2Gk7MrqFqDw_C2F1DW3FGHeriyAIUltUPQzzjMWeu3rNs_HgVl7UnXnMdGqi8F0&sig=AOD64_2RIv7qvDNFqyXAKT8KFEG2NOr_uQ&q&sqi=2&nis=4&adurl&ved=2ahUKEwiZxJum562HAxUbXWwGHVGKDaAQ0Qx6BAgGEAE

1.5 Overview

The Software Architecture Document for our concept drift detection benchmarking system is crucial for understanding its design and structure comprehensively. It begins with a clear Introduction, defining its purpose, scope, and providing essential definitions and references relevant to the project. Utilizing various architectural views such as Use-Case, Logical, Process, Deployment, and Implementation, the document outlines how different components interact and function within the system. This approach not only highlights key architectural decisions but also addresses critical constraints like data security and system performance, ensuring that the system meets its objectives effectively. By detailing subsystems, class responsibilities, data flows, and deployment specifics, the document guides development efforts to ensure the system's reliability, scalability, and adherence to architectural best practices.

2. Architectural Representation

2.1 Use-Case View

The Use-Case View will depict key functionalities related to data ingestion, algorithm implementation, and benchmarking and evaluation processes. It will illustrate how different user roles interact with the system to manage and evaluate concept drift detection algorithms.

2.2 Logical View

The Logical View will focus on the decomposition of the system into subsystems and packages, highlighting architecturally significant components such as data handling modules, algorithmic components, and reporting functionalities. This view will clarify how these components interact to support the system's objectives.

2.3 Process View

The Process View will outline the workflows and processes involved in data acquisition, algorithm testing, and performance assessment. It will illustrate the sequence of activities required to ensure the reliability and effectiveness of concept drift detection algorithms.

2.4 Deployment View

The Deployment View will describe how the system components are distributed across hardware resources, including servers for data storage, processing units for algorithm execution, and user interfaces for interaction. This view will ensure that deployment considerations, such as scalability and resource allocation, are adequately addressed.

2.5 Implementation View

The Implementation View will detail the technologies, frameworks, and tools chosen for developing the system. It will specify how these components are integrated and deployed within the architecture to meet performance and reliability requirements.

3. Architectural Goals and Constraints

3.1 Goals

3.1.1 Enhanced Algorithm Effectiveness:

Ensure that concept drift detection algorithms effectively maintain the accuracy of regression models over time.

3.1.2 Scalability:

Design the architecture to handle large volumes of data and accommodate future growth in dataset sizes and system users.

3.1.3 Robustness:

Develop a system that can withstand data quality issues and integrate seamlessly with existing data pipelines.

3.1.4 Performance:

Optimize algorithm performance to minimize computational overhead and maximize real-time responsiveness.

3.2 Constraints

3.2.1 Data Privacy and Compliance:

Adhere to data privacy regulations and ethical standards by using publicly available and synthetic datasets for evaluation.

3.2.2 Integration Challenges:

Address compatibility issues with existing systems to ensure seamless integration of concept drift detection tools.

3.2.3 Team Expertise and Collaboration:

Leverage diverse team expertise in data science, machine learning, and software engineering to implement and maintain the architecture effectively.

3.2.4 Timeline and Resource Management:

Adhere to a structured timeline and resource allocation plan to ensure timely delivery and efficient project management.

4. Use-Case View

4.1 Use-Case Realizations

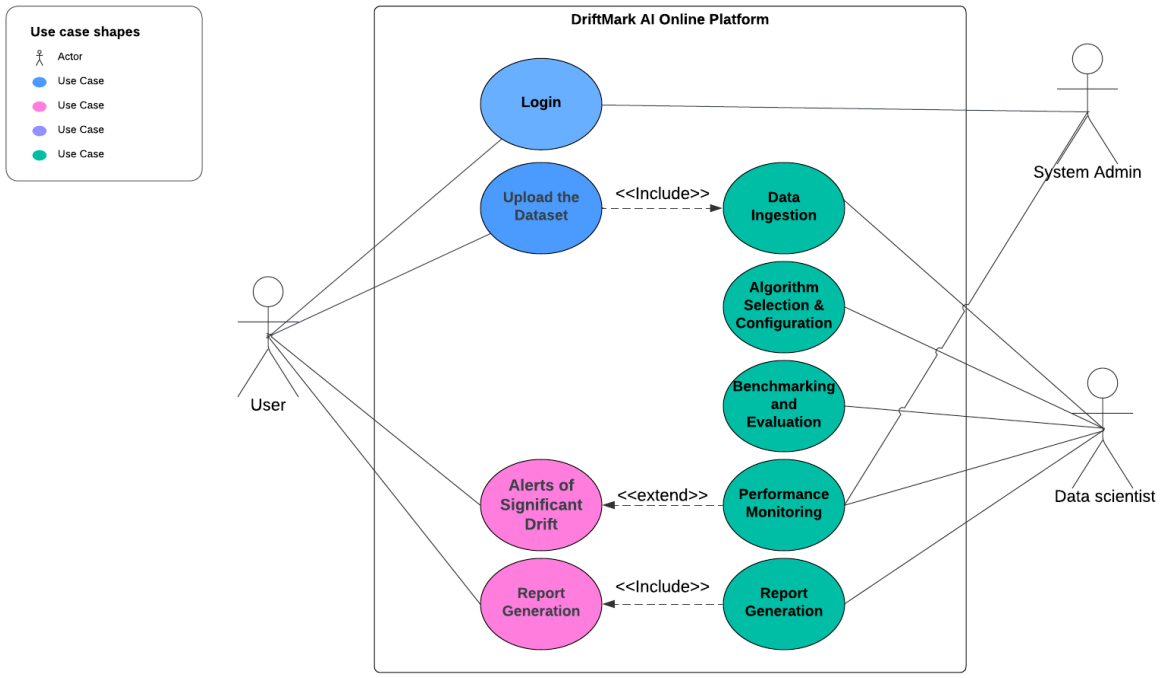


Figure 1. Use case diagram for DriftMark AI Platform which allow users to upload their data sets and check whether a concept drift is present

4.1.1 Use Case 1 : User Login to the DriftMark AI Online Platform

Use case name	User Login to the DriftMark AI Online Platform
Actor	User, System Administrator
Description	This use case involves the user logging into the DriftMark AI platform to access its functionalities.
preconditions	1. User has valid login credentials.
Main flow	1. User navigates to the login page. 2. User enters username and password. 3. System authenticates the credentials.

	4. User gains access to the platform.
Successful end/post condition	User successfully logs in and gains access to the platform.
Fail end/post condition	Login fails due to invalid credentials.
Extensions	If the user forgets the password, they can use the password recovery feature.

4.1.2 Use Case 2 : Upload the Dataset

Use case name	Upload the Dataset
Actor	User
Description	This use case involves the user uploading datasets to the platform for evaluation.
preconditions	1. User is logged into the platform. 2. Datasets are available in a supported format.
Main flow	1. User navigates to the upload section. 2. User selects the dataset file. 3. Dataset is uploaded and stored on the server.
Successful end/post condition	Dataset is successfully uploaded and ready for ingestion.
Fail end/post condition	Upload fails due to file format issues or connectivity problems.
Extensions	If upload fails, the system provides error messages and possible solutions.

4.1.3 Use Case 3 : Data Ingestion

Use case name	Data Ingestion
Actor	Data Scientist/ Machine Learning Engineer
Description	This use case involves acquiring and preprocessing data from multiple sources to prepare it for concept drift detection algorithm evaluation.
preconditions	1. Access to data sources (synthetic datasets, real-world datasets). 2. Availability of data ingestion pipelines and tools.

Main flow	1. Data Scientist selects datasets for evaluation. 2. Data ingestion pipeline retrieves data from selected sources. 3. Data preprocessing steps (cleaning, transformation) are applied as needed. 4. Preprocessed data is stored in a structured format suitable for algorithm evaluation.
Successful end/post condition	Data is successfully ingested and prepared for further evaluation.
Fail end/post condition	Data ingestion fails due to connectivity issues or data quality problems.
Extensions	If data quality issues are detected during preprocessing, notify data scientist for manual intervention or automatic correction if possible.

4.1.4 Use Case 4 : Algorithm Selection and Configuration

Use case name	Algorithm Selection and Configuration
Actor	Data Scientist/ Machine Learning Engineer
Description	This use case involves selecting appropriate concept drift detection algorithms and configuring them based on the characteristics of the dataset and the expected drift patterns.
preconditions	1. Availability of a variety of concept drift detection algorithms. 2. Understanding of dataset characteristics and expected drift scenarios.
Main flow	1. Machine learning engineer reviews available algorithms. 2. Algorithm parameters are adjusted based on dataset characteristics (e.g., feature space, data distribution). 3. Selected algorithms are configured to monitor specific drift patterns (e.g., gradual drift, sudden drift).
Successful end/post condition	Algorithms are configured and ready for benchmarking and evaluation.
Fail end/post condition	Selection or configuration of algorithms fails due to lack of suitable algorithms or inappropriate parameter settings.
Extensions	If selected algorithms prove ineffective during evaluation, consider alternative algorithms or adjust parameters.

4.1.5 Use Case 5 : Benchmarking and Evaluation

Use case name	Benchmarking and Evaluation

Actor	Data Scientist/ Machine Learning Engineer
Description	This use case involves executing benchmarking experiments to evaluate the effectiveness of selected concept drift detection algorithms.
preconditions	1. Availability of preprocessed data. 2. Configured algorithms for concept drift detection.
Main flow	1. Benchmarking experiments are designed and executed using preprocessed data. 2. Algorithms monitor and detect concept drifts over time. 3. Performance metrics (e.g., detection accuracy, false positives) are computed and analyzed. 4. Results are visualized through reports and dashboards.
Successful end/post condition	Evaluation results indicate effective algorithms for maintaining model accuracy under concept drift.
Fail end/post condition	Benchmarking experiments fail to provide clear insights due to algorithm inefficiencies or inadequate data quality.
Extensions	If initial evaluation results are inconclusive, refine experiment parameters or reevaluate algorithms with additional datasets.

4.1.6 Use Case 6 : Performance Monitoring

Use case name	Performance Monitoring
Actor	Data Scientist, System Administrator
Description	This use case involves continuously monitoring the performance of the concept drift detection system to ensure it operates efficiently and accurately.
preconditions	1. System is deployed and operational. 2. Monitoring tools and frameworks are in place.
Main flow	1. Performance metrics are defined (e.g., system latency, detection speed). 2. Monitoring tools collect and display real-time performance data. 3. Alerts are configured to notify relevant personnel of performance anomalies. 4. Regular performance reviews are conducted to ensure optimal operation.
Successful end/post condition	The system maintains high performance and efficiency, with timely alerts for any anomalies.
Fail end/post condition	Performance monitoring fails, leading to undetected issues affecting system operation.

Extensions	Implement additional monitoring tools or refine existing tools to improve performance tracking.

4.1.7 Use Case 7 : Report Generation and Visualization

Use case name	Report Generation and Visualization
Actor	Data Scientist, User
Description	This use case involves generating detailed reports and visualizations of the benchmarking results to communicate findings effectively.
preconditions	1. Benchmarking and evaluation processes are completed. 2. Reporting tools and frameworks are in place.
Main flow	1. Data scientist selects metrics and results for reporting. 2. Reporting tools compile and format the selected data. 3. Visualizations (e.g., charts, graphs) are created to illustrate key findings. 4. Reports are generated and shared with stakeholders.
Successful end/post condition	Clear and comprehensive reports and visualizations are produced, facilitating informed decision-making.
Fail end/post condition	Report generation fails due to data processing issues or tool limitations.
Extensions	Integrate additional data visualization tools or customize existing ones to enhance report quality.

5. Logical View

5.1 Overview

The logical view of the DriftMark AI platform describes the architecturally significant parts of the design model, focusing on its decomposition into subsystems and packages. This view provides a comprehensive understanding of the system's structure in terms of package hierarchy and layers. The system is decomposed into several key packages, each responsible for specific functionalities, ensuring modularity, maintainability, and scalability.

5.2 Architecturally Significant Design Packages

5.2.1 User Management Package

1. Description: This package handles all user-related functionalities, including authentication, authorization, and user profile management.
2. Significant Classes:
    a. User: Manages user information such as username, password, and role.

i. Responsibilities: Authentication, storing user details, role assignment.
ii. Key Operations: login(), logout().

b. AuthenticationService: Handles the login and logout process.
i. Responsibilities: User authentication, session management.
ii. Key Operations: authenticate(), terminateSession().

5.2.2 Data Management Package

1. Description: This package is responsible for data ingestion, preprocessing, and storage.
2. Significant Classes:
   a. Dataset: Represents the dataset to be processed.
      i. Responsibilities: Data storage, metadata management.
      ii. Key Operations: loadData(), preprocessData().
   b. DataIngestionService: Manages the data ingestion process.
      i. Responsibilities: Ingesting data from various sources, initial validation.
      ii. Key Operations: ingestData(), validateData().

5.2.3 Algorithm Management Package

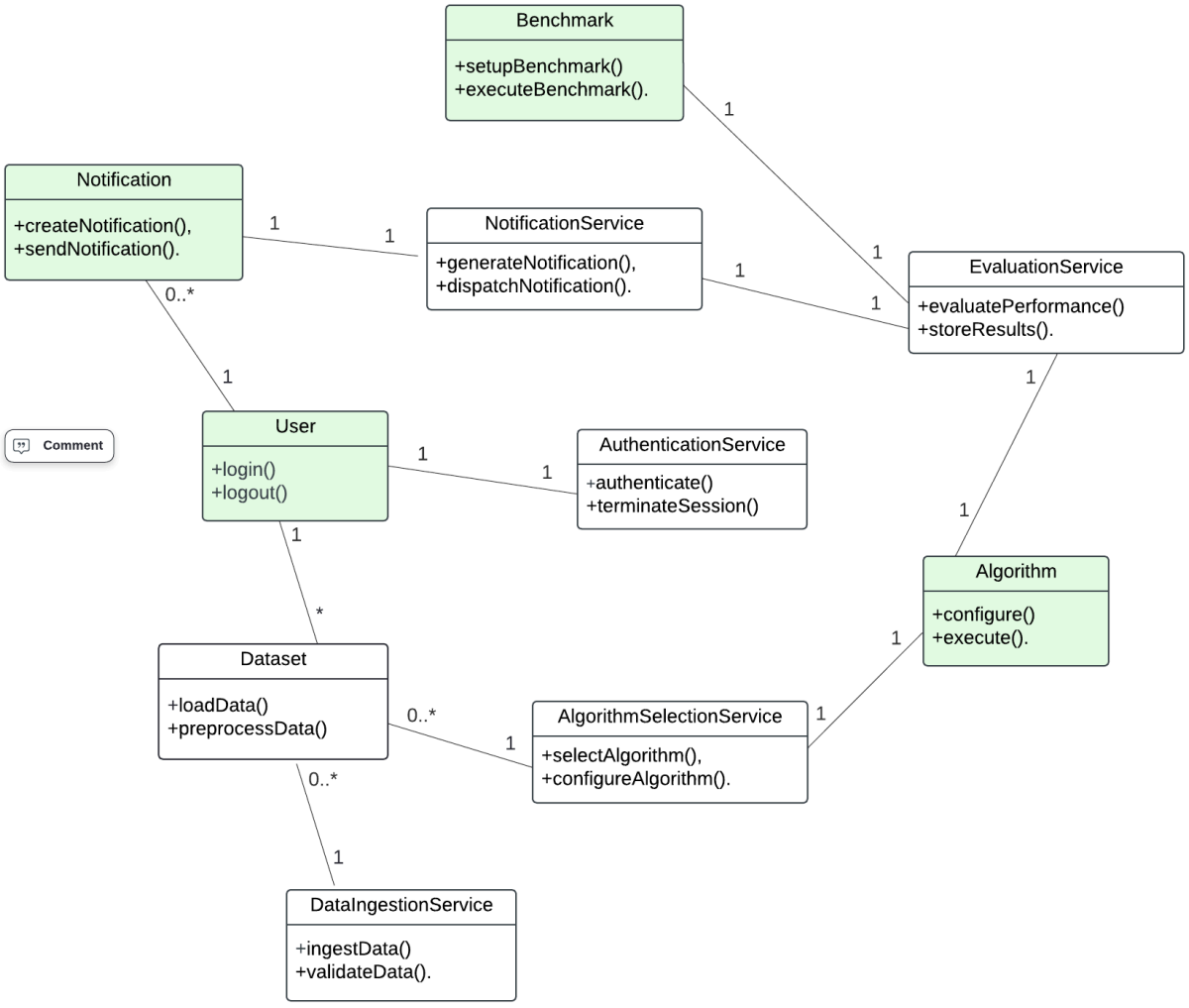
1. Description: This package handles the selection, configuration, and execution of machine learning algorithms.
2. Significant Classes:
   a. Algorithm: Represents a machine learning algorithm.
      i. Responsibilities: Algorithm configuration, execution.
      ii. Key Operations: configure(), execute().
   b. AlgorithmSelectionService: Manages the selection and configuration of algorithms.
      i. Responsibilities: Providing algorithm options, configuring selected algorithm.
      ii. Key Operations: selectAlgorithm(), configureAlgorithm().

5.2.4 Benchmarking Package

1. Description: This package manages the benchmarking and evaluation of algorithms.
2. Significant Classes:
   a. Benchmark: Represents a benchmarking process.
      i. Responsibilities: Setting up benchmark tests, executing tests.
      ii. Key Operations: setupBenchmark(), executeBenchmark().
   b. EvaluationService: Handles the evaluation of algorithm performance.
      i. Responsibilities: Performance metrics calculation, results storage.
      ii. Key Operations: evaluatePerformance(), storeResults().

5.2.5 Notification Package

1. Description: This package handles the notification system for alerting users about significant drifts.
2. Significant Classes:
   a. Notification: Represents a notification.
      i. Responsibilities: Storing notification details, sending notifications.
      ii. Key Operations: createNotification(), sendNotification().
   b. NotificationService: Manages the notification process.
      i. Responsibilities: Generating and dispatching notifications.
      ii. Key Operations: generateNotification(), dispatchNotification().



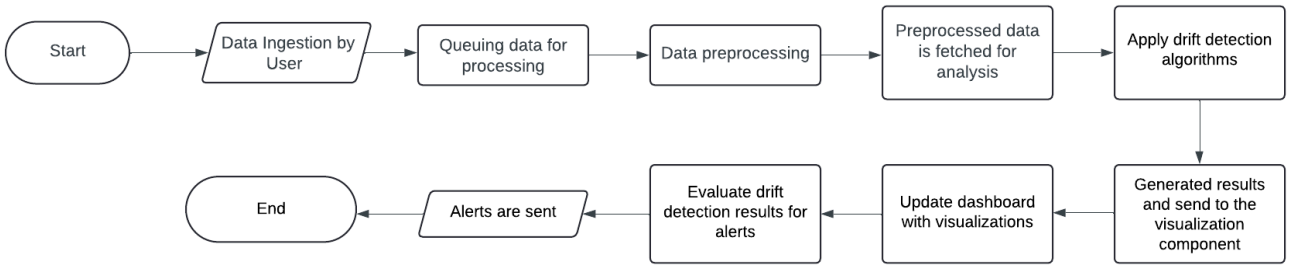
Class Diagram

6. Process View

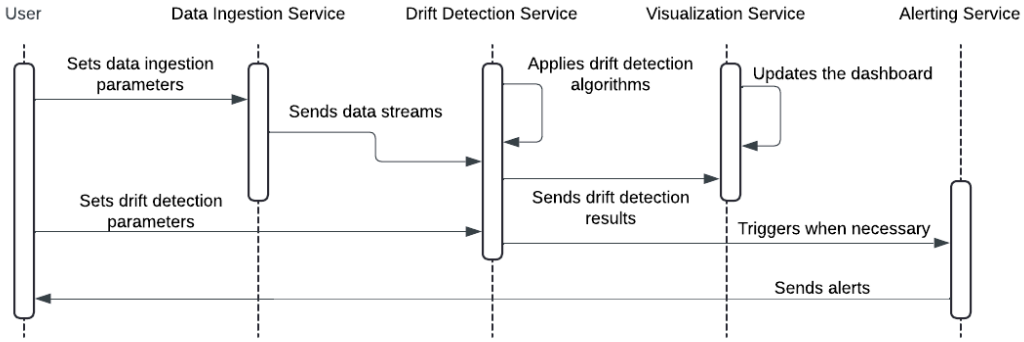
The Drift Detection Dashboard System is composed of several lightweight and heavyweight processes that manage data ingestion, drift detection, visualization, and alerting. This section details the interaction and communication between these processes.

Following heavyweight processes include their own lightweight processes as mentioned.

- Data Ingestion Process
    - Description: Continuously ingests data from various sources based on user-defined parameters.
    - Lightweight processes:
        - Data is continuously ingested from multiple sources.
        - Ingested data is queued for processing.
    - Communication: Utilizes message passing to queue new data batches.
- Data Preprocessing process
    - Description: Data is preprocessed as dataset is ready for detect drifts
    - Lightweight processes:
        - Data cleaning
        - Data Transformation
    - Communication: Listens for new data messages and preprocesses.
- Drift Detection Process
    - Description: Analyzes the ingested data to detect any drift according to user-defined parameters.
    - Lightweight processes:
        - Preprocessed data is fetched for analysis.
        - Drift detection algorithms are applied.
        - Results are generated and sent to the visualization component.
    - Communication: Processes the data, and sends results to the visualization component.
- Visualization Process
    - Description: Updates the dashboard with the latest drift detection results.
    - Lightweight processes:
        - Results are received and processed.
        - Dashboard is updated with new visualizations.
    - Communication: Receives processed data through message queues or API calls.
- Alerting Process
    - Description: Sends notifications based on drift detection results.
    - Lightweight processes:
        - Drift detection results are evaluated for alerts.
        - Alerts are sent to the dashboard.
    - Communication: Uses interrupts to handle urgent notifications and message passing for regular alerts.



Activity Diagram



Sequence Diagram

7. Deployment View

The Drift Detection Dashboard System is designed to be deployed in a cloud-based environment. The following description outlines the physical nodes and their interconnections, along with a mapping of the processes from the Process View onto these nodes.

- User Devices
  - Users access the Drift Detection Dashboard through their web browsers over the internet.
- Web Server Node
  - Hosts the frontend application and backend API service, handling user requests and interactions.
- Data Processing Cluster
  - Composed of multiple nodes for data ingestion and real-time processing.
  - Connects to the web server node for data ingestion and to the drift detection node for processing results.
- Drift Detection Node
  - Runs drift detection algorithms on the ingested data and sends results to the visualization node.
  - Stores results in the PostgreSQL database and triggers the alerting service when significant drift is detected.
- Visualization Node
  - Updates the dashboard with the latest drift detection results for user viewing.
  - Receives processed data from the drift detection node and sends updated visualizations to the web server node.
- Database Node
  - Stores data and drift detection results in a database.
  - Provides persistent storage and can be accessed by all other nodes as needed.

8. Implementation View

The Drift Detection Dashboard System is implemented using a multi-layered architecture, with each layer handling specific responsibilities. The system is decomposed into several subsystems and components, each designed to address different aspects of drift detection, data processing, visualization, and alerting. This section describes the overall structure, layers, and architecturally significant components of the implementation model.

Layered Architecture

- Presentation Layer
  - Components: Frontend Application
  - Responsibilities: User interface, interaction handling, displaying visualizations and alerts
- Application Layer
  - Components: Backend API Service, Dashboard Service
  - Responsibilities: Business logic, API endpoints, data aggregation, and processing coordination.
- Processing Layer
  - Components: Data Ingestion Service, Drift Detection Service
  - Responsibilities: Data ingestion and streaming, drift detection algorithms, data processing.
- Data Layer
  - Components: Data Sources, Database
  - Responsibilities: Data storage, retrieval, and management.

Subsystems and Components

- Frontend Application
  - Description: A dashboard application implemented using the React framework.
  - Responsibilities: Provides an intuitive user interface for interacting with the dashboard, configuring parameters, and viewing drift detection results.
- Backend API Service
  - Description: A RESTful API service implemented using a framework such as Flask or Django.
  - Responsibilities: Handles user requests, manages communication between frontend and backend components, and orchestrates data processing.
- Dashboard Service
  - Description: A service responsible for generating visualizations and updating the dashboard.
  - Responsibilities: Processes drift detection results, generates visualizations, and provides data to the frontend application.
- Data Ingestion Service
  - Description: A service that ingests data from various sources using tools like Kafka.
  - Responsibilities: Connects to data sources, ingests and streams data for processing.
- Drift Detection Service
  - Description: A service that applies drift detection algorithms to the ingested data.
  - Responsibilities: Analyzes data to detect drift, processes results, and stores them in the database.
- Database
  - Description: A database used for persistent storage of data and drift detection results.
  - Responsibilities: Stores and manages data, provides efficient data retrieval and query capabilities.

8.1 Overview

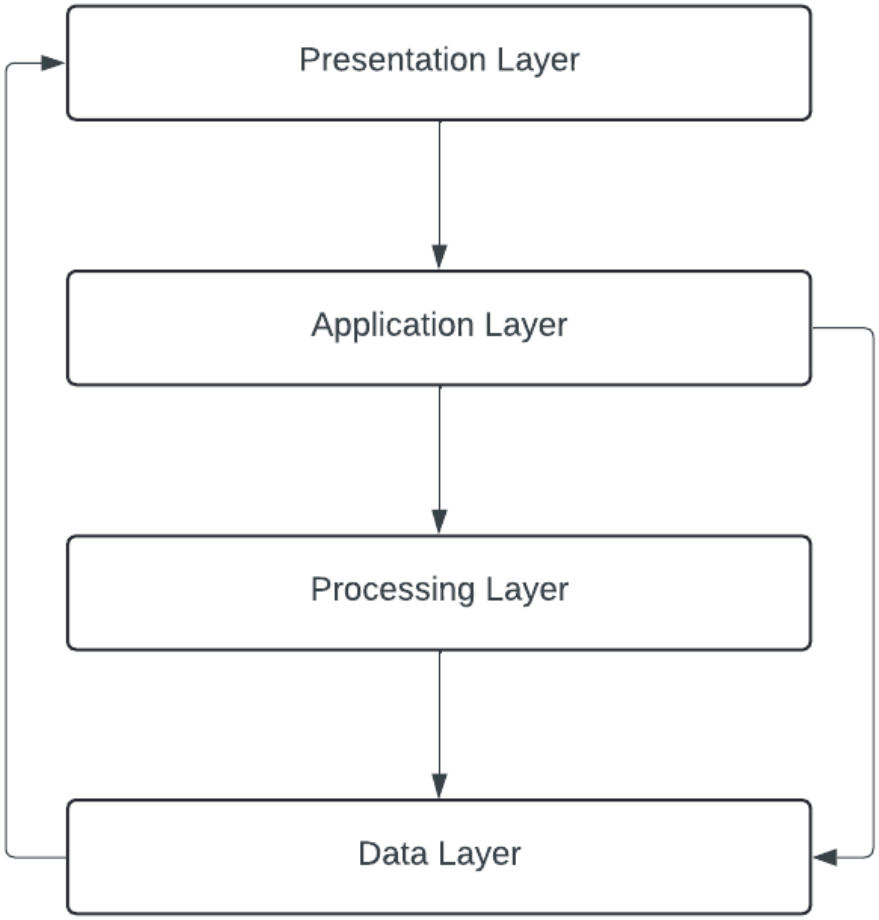
Overview of Layers

- Presentation Layer
  - Definition: This layer handles the user interface and interaction logic.
  - Contents: Frontend Application
  - Inclusion Rules: Contains components responsible for displaying data, receiving user inputs, and providing visual feedback.
  - Boundaries: Interacts with the Application Layer through well-defined API calls.
- Application Layer
  - Definition: This layer contains the business logic and orchestration of various services.
  - Contents: Backend API Service, Dashboard Service
  - Inclusion Rules: Contains components that implement business rules, manage data processing requests, and coordinate between different services.
  - Boundaries: Interacts with the Presentation Layer, Processing Layer, and Data Layer via API endpoints and service calls.
- Processing Layer
  - Definition: This layer is responsible for data ingestion, processing, and drift detection.
  - Contents: Data Ingestion Service, Drift Detection Service
  - Inclusion Rules: Contains components that handle real-time data ingestion, processing pipelines, and drift detection algorithms.
  - Boundaries: Receives data from the Application Layer and interacts with the Data Layer and Infrastructure Layer for processing.
- Data Layer
  - Definition: This layer manages data storage and retrieval.
  - Contents: Data Sources, Database
  - Inclusion Rules: Contains components that store and manage persistent data.
  - Boundaries: Interacts with the Processing Layer for data storage and retrieval and with the Application Layer for data access.

Relations Between Layers

- Presentation Layer to Application Layer
  - Communication: The Frontend Application communicates with the Backend API Service via HTTP requests.
  - Purpose: To send user inputs and receive processed data for display.
- Application Layer to Processing Layer
  - Communication: The Backend API Service sends data requests to the Data Ingestion Service and receives processed data from the Drift Detection Service.
  - Purpose: To manage data ingestion and processing, coordinating the flow of data between layers.
- Processing Layer to Data Layer
  - Communication: The Data Ingestion Service streams data to Kafka, which is then processed by Flink and analyzed by the Drift Detection Service. The results are stored in the Database.
  - Purpose: To handle data ingestion, processing, and storage efficiently.

- Application Layer to Data Layer
    - Communication: The Backend API Service accesses the Database for data retrieval and storage.
    - Purpose: To manage and query persistent data for application logic.

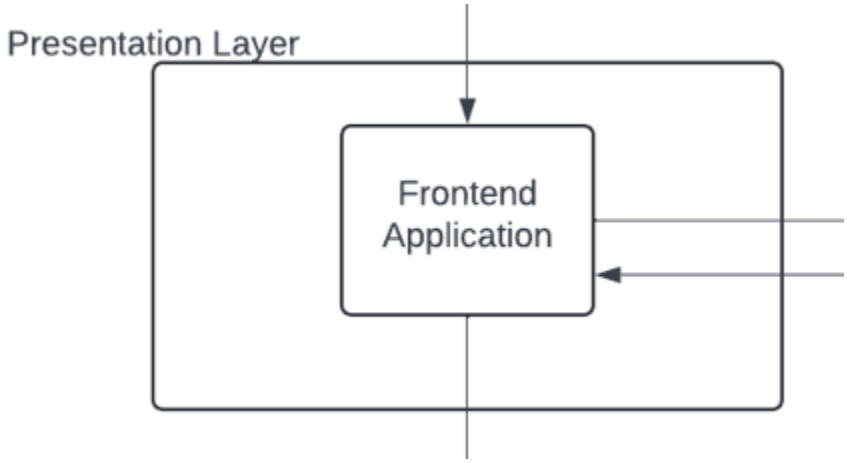


8.2 Layers

8.2.1 Presentation Layer

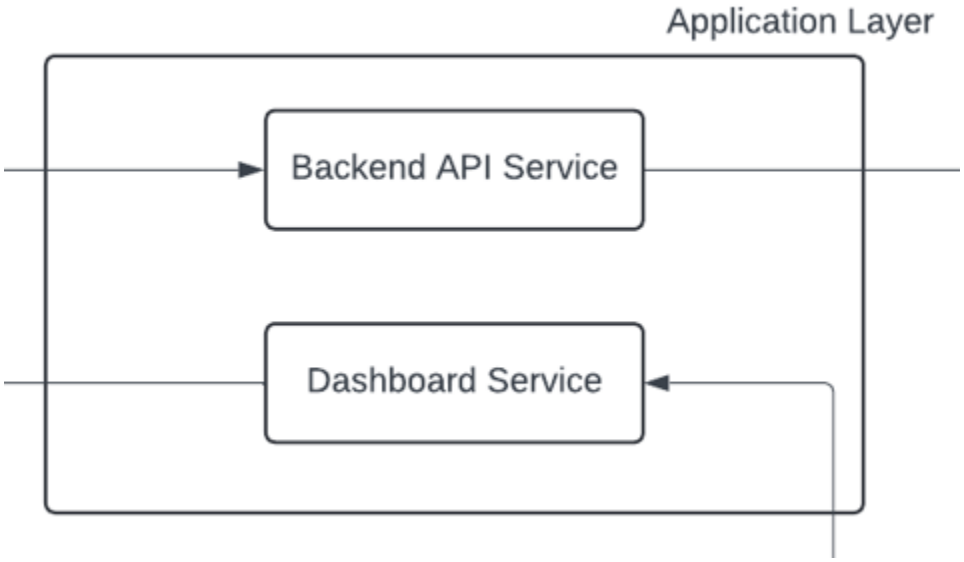
- Subsystems

    - Frontend Application
- Description
    - The Presentation Layer contains the `Frontend Application`, which is responsible for the user interface, handling user interactions, and displaying visualizations and alerts. This layer communicates with the Application Layer through well-defined API calls.



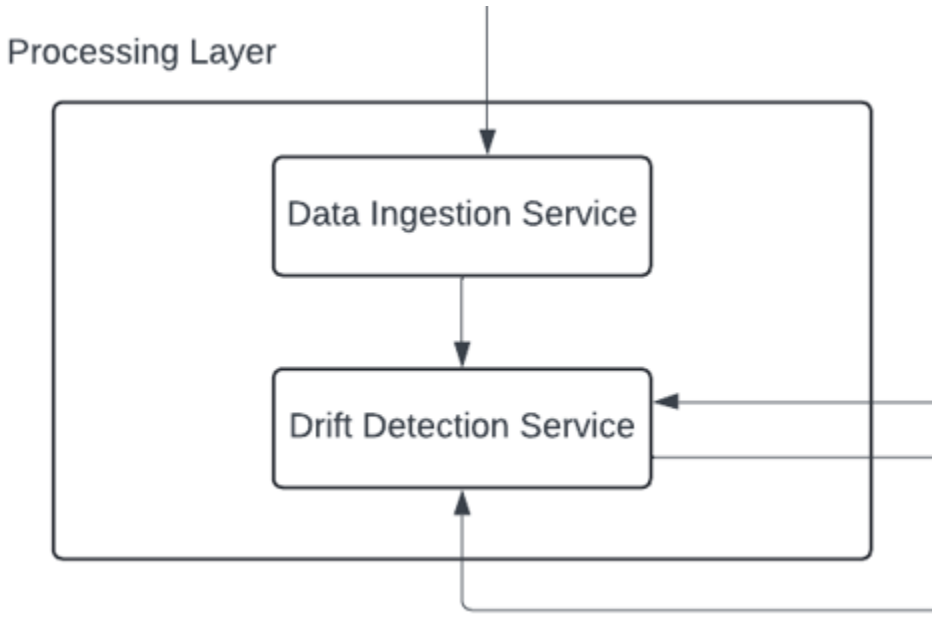
8.2.2 Application Layer

- Subsystems
    - Backend API Service
    - Dashboard Service
- Description
    - The Application Layer includes the Backend API Service and Dashboard Service. The Backend API Service handles user requests, manages communication between frontend and backend components, and orchestrates data processing. The Dashboard Service processes drift detection results and generates visualizations for the frontend.



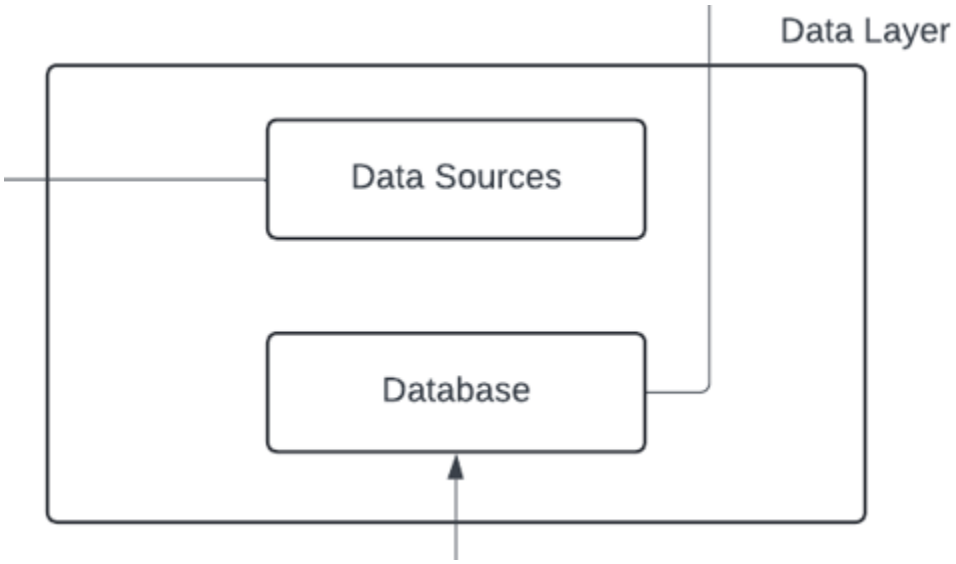
8.2.3 Processing Layer

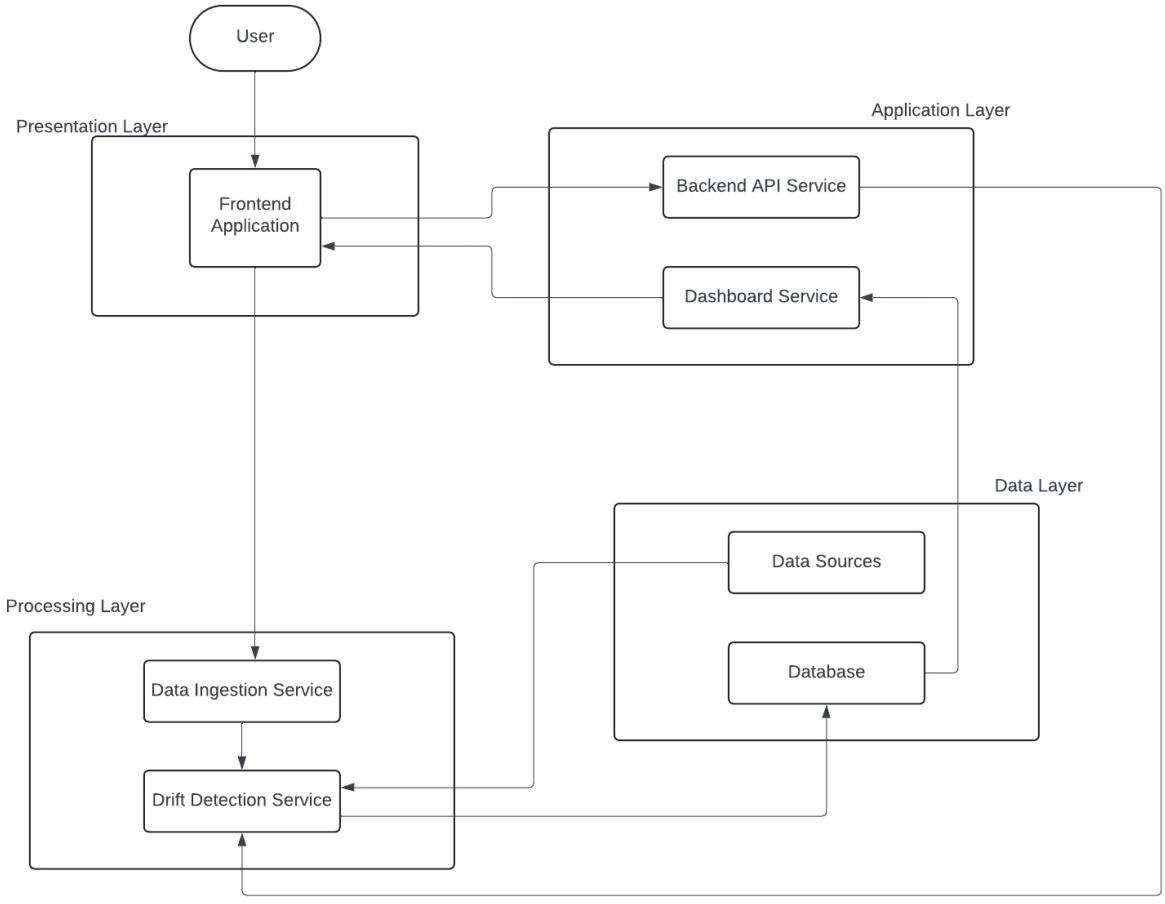
- Subsystems
  - Data Ingestion Service
  - Drift Detection Service
- Description
  - The Processing Layer contains the Data Ingestion Service and the Drift Detection Service. The Data Ingestion Service ingests data from various sources using tools like Kafka, and the Drift Detection Service applies drift detection algorithms to the ingested data.



8.2.4 Data Layer

- Subsystems
  - Data Sources
  - Database
- Description
  - The Data Layer includes Data Sources and the Database. This layer is responsible for data storage, retrieval, and management, providing persistent storage for the drift detection results and other relevant data.





Package Diagram

9. Size and Performance

The size and performance of the Drift Detection Dashboard System are influenced by several key factors, including data volume, processing speed, storage capacity, and user concurrency. These factors impact the architecture's design choices and determine the system's scalability, responsiveness, and efficiency.

9.1 Dimensioning Characteristics

- Data Volume
  - Description: The system must handle large volumes of data, potentially in real-time, from various sources. The data could range from structured datasets to real-time streams.
  - Impact: The architecture must support scalable data ingestion, efficient processing pipelines, and storage mechanisms capable of handling high data throughput.
- Processing Speed
  - Description: The system should process incoming data quickly to detect drifts in near real-time. This includes the time taken for data ingestion, processing, and generating detection results.
  - Impact: The architecture should leverage efficient data processing frameworks to ensure low-latency processing.
- Storage Capacity
  - Description: The system must store large datasets and historical drift detection results. This includes raw data, processed data, and results of drift detection algorithms.
  - Impact: The architecture should use scalable and high-performance storage solutions, such as relational databases or distributed storage systems, to manage large volumes of data.
- User Concurrency
  - Description: The system should support multiple concurrent users who may perform various actions such as configuring parameters, initiating data processing, and viewing visualizations.
  - Impact: The architecture should ensure that the backend services can handle high concurrency and maintain responsive interactions with the frontend.

9.2 Target Performance Constraints

- Latency
  - Target: The system should detect and report data drifts within seconds to a few minutes of data arrival.
  - Constraint: The architecture must minimize processing delays by optimizing data pipelines and using real-time processing frameworks.
- Throughput
  - Target: The system should process thousands of data points per second to handle real-time streams.
  - Constraint: The architecture must ensure high throughput by leveraging efficient data ingestion and processing tools.
- Storage Scalability
  - Target: The system should scale to store terabytes to petabytes of data, accommodating both current and historical datasets.
  - Constraint: The architecture must support scalable storage solutions and optimize data management strategies.

- Availability
    - Target: The system should maintain high availability (e.g., 99.9% uptime) to ensure continuous operation and reliability.
    - Constraint: The architecture must incorporate redundancy, failover mechanisms, and robust monitoring to achieve high availability.
- User Experience
    - Target: The system should provide a responsive user interface with minimal load times for data visualizations and reports.
    - Constraint: The architecture must optimize API performance and frontend rendering to ensure a smooth user experience.

10. Quality

The software architecture of the Drift Detection Dashboard System is designed to ensure that it meets not only functional requirements but also non-functional requirements. These non-functional requirements, also known as quality attributes, are crucial for the system's overall effectiveness and user satisfaction. This section describes how the architecture contributes to various quality attributes, including extensibility, reliability, portability, security, and privacy.

10.1 Extensibility

- Description
    - Extensibility refers to the system's ability to incorporate new features and functionalities with minimal impact on existing components.
- Contribution
    - Modular Design: The architecture is divided into distinct layers (Presentation, Application, Processing, Data, Infrastructure), each with specific responsibilities. This modularity allows for independent development and enhancement of each layer.
    - Microservices Architecture: Key services (e.g., Backend API Service, Drift Detection Service) are implemented as microservices, facilitating the addition of new services without affecting the existing ones.
    - Plug-in Mechanism: The system supports plug-ins for new drift detection algorithms, making it easy to extend the processing capabilities without major architectural changes.

10.2 Reliability

- Description
    - Reliability ensures that the system operates correctly and consistently over time, even in the face of failures.
- Contribution
    - Redundancy: Critical components are replicated to provide redundancy, ensuring that the system remains operational even if some components fail.
    - Fault Tolerance: The use of distributed processing frameworks (e.g., Kafka, Flink) ensures fault tolerance, with mechanisms for automatic recovery from failures.
    - Monitoring and Alerts: Continuous monitoring and alerting mechanisms are in place to detect and respond to issues promptly, minimizing downtime.

10.3 Portability

- Description
    - Portability is the ease with which the system can be transferred from one environment to another.
- Contribution

- Containerization: The use of container technologies (e.g., Docker) encapsulates the application and its dependencies, making it easy to deploy across different environments (development, testing, production).
- Cross-platform Compatibility: The architecture leverages platform-independent technologies (e.g., Java, Python) ensuring compatibility across various operating systems and hardware configurations.

10.4 Security

- Description
  - Security involves protecting the system against unauthorized access, data breaches, and other threats.
- Contribution
  - Authentication and Authorization: Robust authentication and authorization mechanisms ensure that only authorized users can access the system and perform specific actions.
  - Data Encryption: Sensitive data is encrypted both in transit and at rest to protect against eavesdropping and unauthorized access.
  - Secure Communication: All communication between components and external systems is secured using protocols like HTTPS and TLS.

10.5 Privacy

- Description
  - Privacy ensures that users' data is handled in compliance with privacy regulations and best practices.
- Contribution
  - Data Anonymization: Techniques for data anonymization are employed to protect user identities in datasets.
  - Compliance: The architecture is designed to comply with relevant privacy regulations (e.g., GDPR), ensuring that data collection, storage, and processing adhere to legal requirements.
  - Access Control: Strict access control policies are in place to limit who can view or manipulate sensitive data.

10.6 Performance

- Description
  - Performance involves the system's responsiveness and throughput under various conditions.
- Contribution
  - Scalable Architecture: The system uses scalable components (e.g., Kafka for data streaming, Flink for processing) to handle high data volumes and processing loads.
  - Load Balancing: Load balancers distribute incoming requests evenly across servers, preventing any single server from becoming a bottleneck.
  - Optimization: Continuous performance tuning and optimization ensure that the system meets performance targets and can scale to accommodate growth.

10.7 Usability

- Description
  - Usability is the ease with which users can interact with the system and achieve their goals.
- Contribution
  - Intuitive UI: The frontend application provides an intuitive and user-friendly interface, making it easy for users to configure parameters and view drift detection results.
  - Responsive Design: The user interface is designed to be responsive, providing a consistent experience across various devices and screen sizes.

- User Feedback: Mechanisms for collecting user feedback help in continuously improving the usability of the system.

10.7 Maintainability

- Description
  - Maintainability is the ease with which the system can be maintained and evolved over time.
- Contribution
  - Clear Separation of Concerns: The layered architecture ensures a clear separation of concerns, making it easier to locate and fix issues.
  - Documentation: Comprehensive documentation is provided for each component and service, aiding in maintenance and onboarding of new developers.
  - Automated Testing: Automated testing frameworks are integrated into the development process, ensuring that changes do not introduce new bugs.

10.8 Scalability

- Description
  - Scalability is the ability of the system to handle increased load by adding resources.
- Contribution
  - Horizontal Scaling: The use of microservices and containerization allows for horizontal scaling, where additional instances can be deployed to handle increased load.
  - Elastic Resources: Integration with cloud services provides elastic resource allocation, enabling the system to scale up or down based on demand.