Software Engineering & Concepts

Lab Manual

Table of Contents

Overview of the Project

Objective:

Proactively predict and prevent system issues by analyzing system logs to forecast errors and warnings.

Key Operations:

- **Error/Warning Detection:**
  - Log Scanning
  - Log Parsing
  - Classification
- **Prediction:**
  - Data Analysis
  - Predictive Modeling
  - Validation
- **Reporting and Analysis:**
  - Reporting
  - Prediction Analysis
  - Recommendations

Benefits:

- **Enhanced Reliability:** Predict and address issues before they escalate.
- **Reduced Downtime:** Minimize unexpected system failures.
- **Cost Savings:** Lower maintenance and repair costs through early intervention.

Business Architecture Diagram: Predictive Maintenance

Business Need:

- **Optimize Asset Performance:**
  - Maximize equipment lifespan
  - Minimize unplanned downtime
- **Risk Reduction:**
  - Early failure detection
  - Avoid costly breakdowns
  - Improve safety
- **Efficiency:**
  - Leverage real-time data
  - Enhance production efficiency
  - Reduce costs

Current Process:

- **Reactive Maintenance:** Fixing failures when they occur
- **Preventive Maintenance:** Scheduled maintenance
- **Condition-Based Monitoring:** Using sensors and analytics to assess equipment health continuously
- **Predictive Maintenance (PdM):** Insights into actual asset conditions

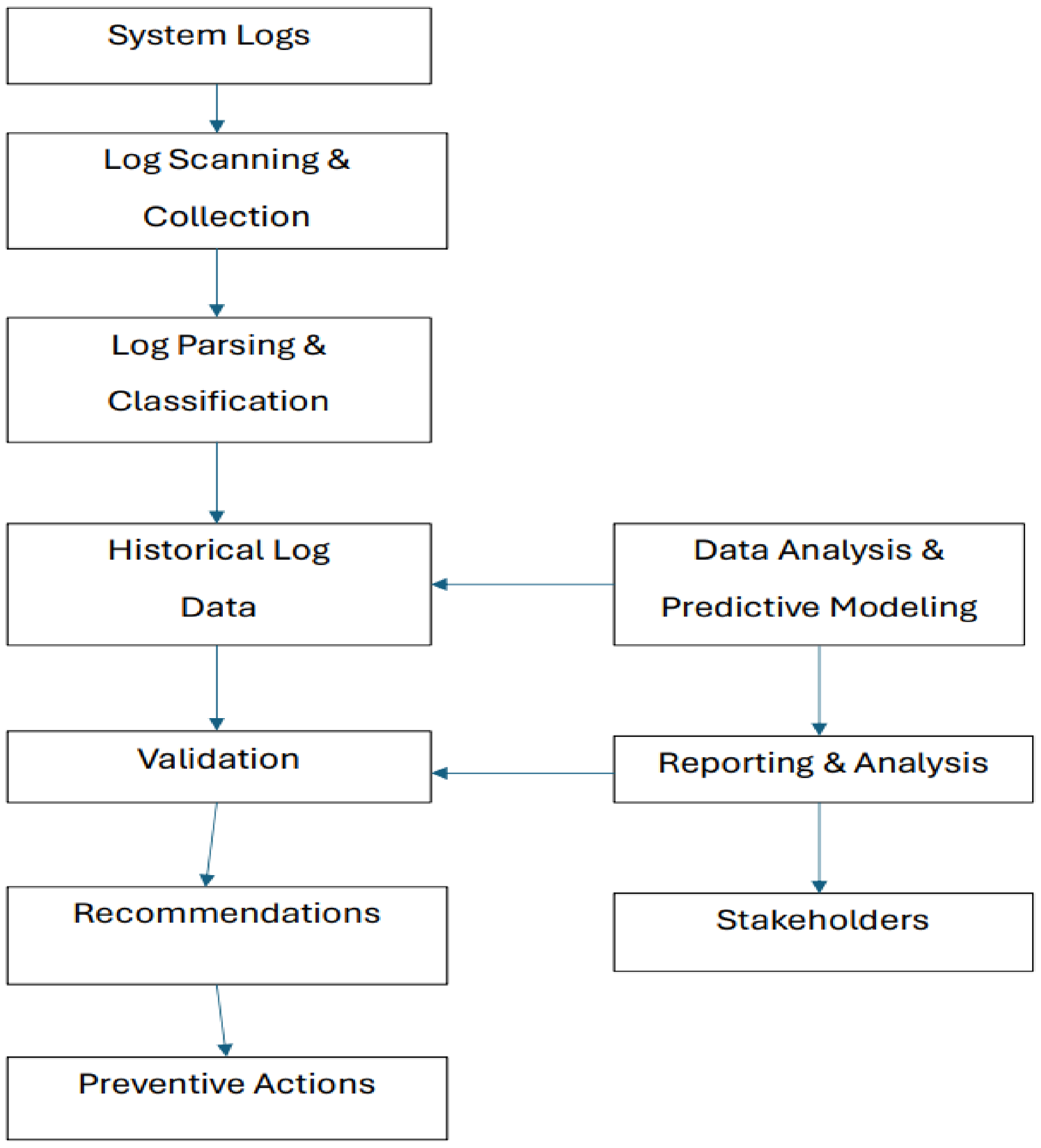
Personas:

- **Maintenance Teams:** Real-time insights for prioritizing tasks
- **Operations Managers:** Optimize production schedules and minimize disruptions
- **Finance and Safety Teams:** Cost savings from reduced downtime and improved safety

Business Problems Addressed:

- **Unplanned Downtime:**

  - Prevent unexpected equipment failures
  - Minimize production interruptions
- **Maintenance Costs:**
  - Avoid unnecessary maintenance
  - Save costs
- **Asset Lifespan:**
  - Extend the useful life of critical assets



Requirements as User Stories

User Stories for Predictive Maintenance :

User Story 1: As a data analyst, I want to integrate data from multiple sensors and systems into a single dashboard so that I can have a comprehensive view of equipment health.

User Story 2: As a maintenance manager, I want to track the effectiveness of predictive maintenance actions so that I can improve the accuracy of future predictions.

User Story 3: As a technician, I want to access step-by-step maintenance procedures and manuals digitally so that I can perform maintenance tasks efficiently.

User Story 4: As a maintenance manager, I want to forecast maintenance costs based on predicted failures so that I can budget more accurately.

User Story 5: As a technician, I want to log maintenance activities and outcomes digitally so that there is a record for future reference.

User Story 6: As a maintenance manager, I want to analyze trends in equipment failures across different sites so that I can identify common issues and address them proactively.

User Story 7: As a maintenance manager, I want to receive notifications of predicted equipment failures so that I can schedule maintenance before a breakdown occurs.

User Story 8: As a technician, I want to view detailed analytics and historical data of equipment performance so that I can understand the root cause of issues.

User Story 9: As a maintenance manager, I want to generate maintenance schedules based on predictive analytics so that I can optimize resource allocation.

User Story 10: As a technician, I want to receive real-time alerts on my mobile device about critical equipment issues so that I can take immediate action.

Estimation of User Stories using Poker Planning Methodology

User Story 1:

Integration of data from multiple sensors and systems

Estimate: 13 points (High complexity and effort)

User Story 2:

Tracking effectiveness of predictive maintenance actions

Estimate: 8 points (Moderate to high complexity and effort)

User Story 3:

Accessing step-by-step maintenance procedures digitally

Estimate: 3 points (Low complexity and effort)

User Story 4:

Forecasting maintenance costs

Estimate: 5 points (Moderate complexity and effort)

User Story 5:

Logging maintenance activities digitally

Estimate: 3 points (Low complexity and effort)

User Story 6:

Analyzing trends in equipment failures

Estimate: 8 points (Moderate to high complexity and effort)

User Story 7:

Receiving notifications of predicted equipment failures

Estimate: 8 points (Moderate to high complexity and effort)

User Story 8:

Viewing detailed analytics and historical data

Estimate: 5 points (Moderate complexity and effort)

User Story 9:

Generating maintenance schedules based on predictive analytics

Estimate: 8 points (Moderate to high complexity and effort)

User Story 10:

Receiving real-time alerts on mobile devices

Estimate: 5 points (Moderate complexity and effort)

Final Estimation:

3 Points (Low complexity and effort)**:** User Stories **3, 5**

5 Points (Moderate complexity and effort)**:** User Stories **4, 8, 10**

8 Points (Moderate to high complexity and effort)**:** User Stories **2, 6, 7, 9**

13 Points (High complexity and effort)**:** User Story **1**

Non-Functional Requirements (NFRs) :

Performance: The app should load the menu within 2 seconds.

Scalability: The app should handle up to 10,000 simultaneous users.

Usability: The app should have an intuitive and easy-to-navigate user interface.

Security: User data should be encrypted and stored securely.

Reliability: The app should have an uptime of 99.9%.

Architecture Diagram

Architecture pattern used:

We are using a Microservices Architecture for our predictive maintenance system.

Why Microservices Architecture?

Independent Scaling: Each microservice can be scaled independently based on its workload. For instance, the Data Collection Service may need to handle high throughput from sensors, while the Predictive Analysis Service may require more computational resources.

Technology Agnostic: Different services can be developed using the most appropriate technology stacks. For example, we can use Python for data processing and analysis, while leveraging Node.js for real-time notifications.

Modular Structure: The architecture divides the application into smaller, manageable services, making it easier to update and maintain. Changes in one service do not directly affect others.

Independent Deployment: Each microservice can be deployed independently, facilitating continuous integration and deployment (CI/CD) practices. This enhances our ability to release new features and updates quickly.

Fault Isolation: Failures in one microservice do not cause the entire system to fail. For instance, if the Notification Service goes down, the rest of the system can continue to function.

Design principles Used:

In our predictive maintenance system using a microservices architecture, we adhere to several key design principles to ensure the system is robust, scalable, and maintainable. Here are the design principles we use and the reasons why:

1. Single Responsibility Principle (SRP)

Principle: Each microservice should have a single responsibility or focus.

Why: This ensures that each service is simple, focused, and easier to manage. It allows teams to develop, deploy, and scale services independently, improving maintainability and reducing complexity.

2. Loose Coupling

Principle: Microservices should be loosely coupled, meaning changes in one service should not require changes in another.

Why: This enhances flexibility and resilience. Services can evolve independently, and failures in one service do not cascade to others, improving the system's robustness.

3. Decentralized Data Management

Principle: Each microservice manages its own database.

Why: This avoids the pitfalls of a single monolithic database and ensures that services are independent. It allows for the use of different types of databases best suited for each service's needs (e.g., SQL, NoSQL).

4. Autonomous Services

Principle: Microservices should be independently deployable and scalable.

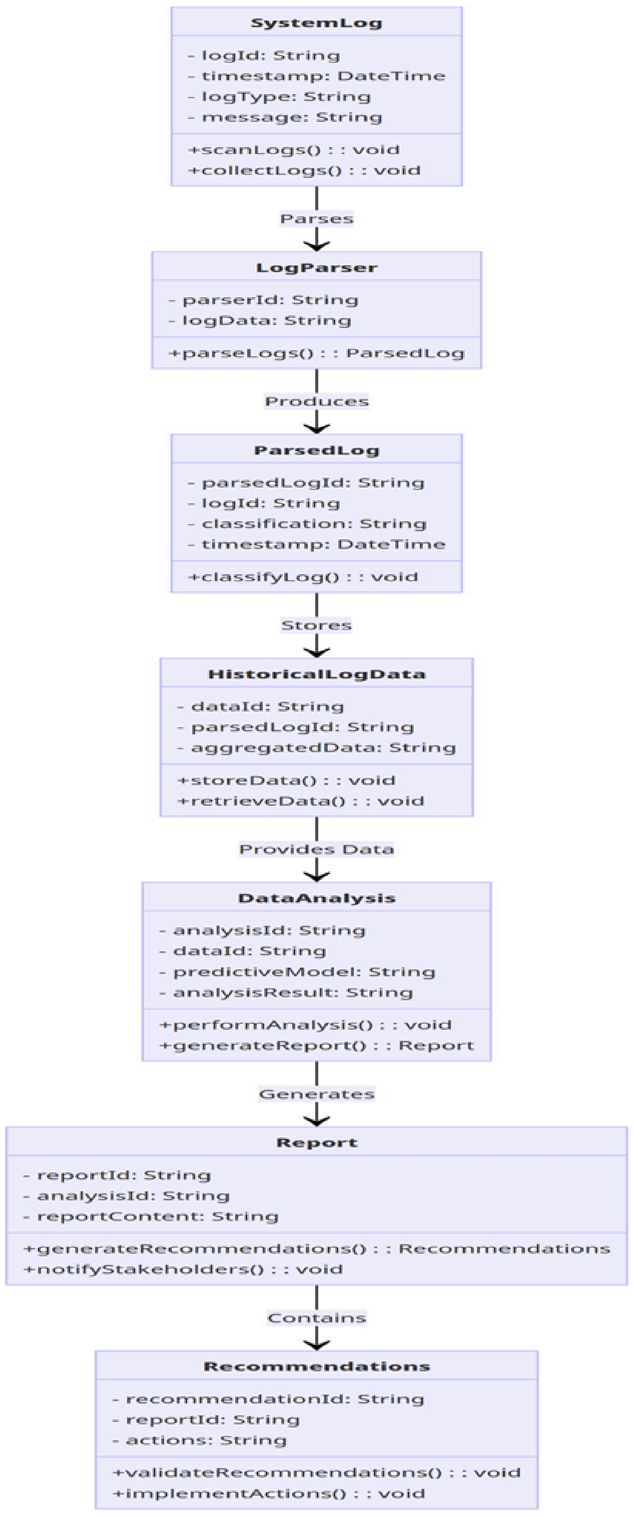
Why: This supports continuous integration and deployment (CI/CD) practices. Services can be scaled based on their specific requirements, enhancing overall system scalability and resource utilization.

5. Event-Driven Communication

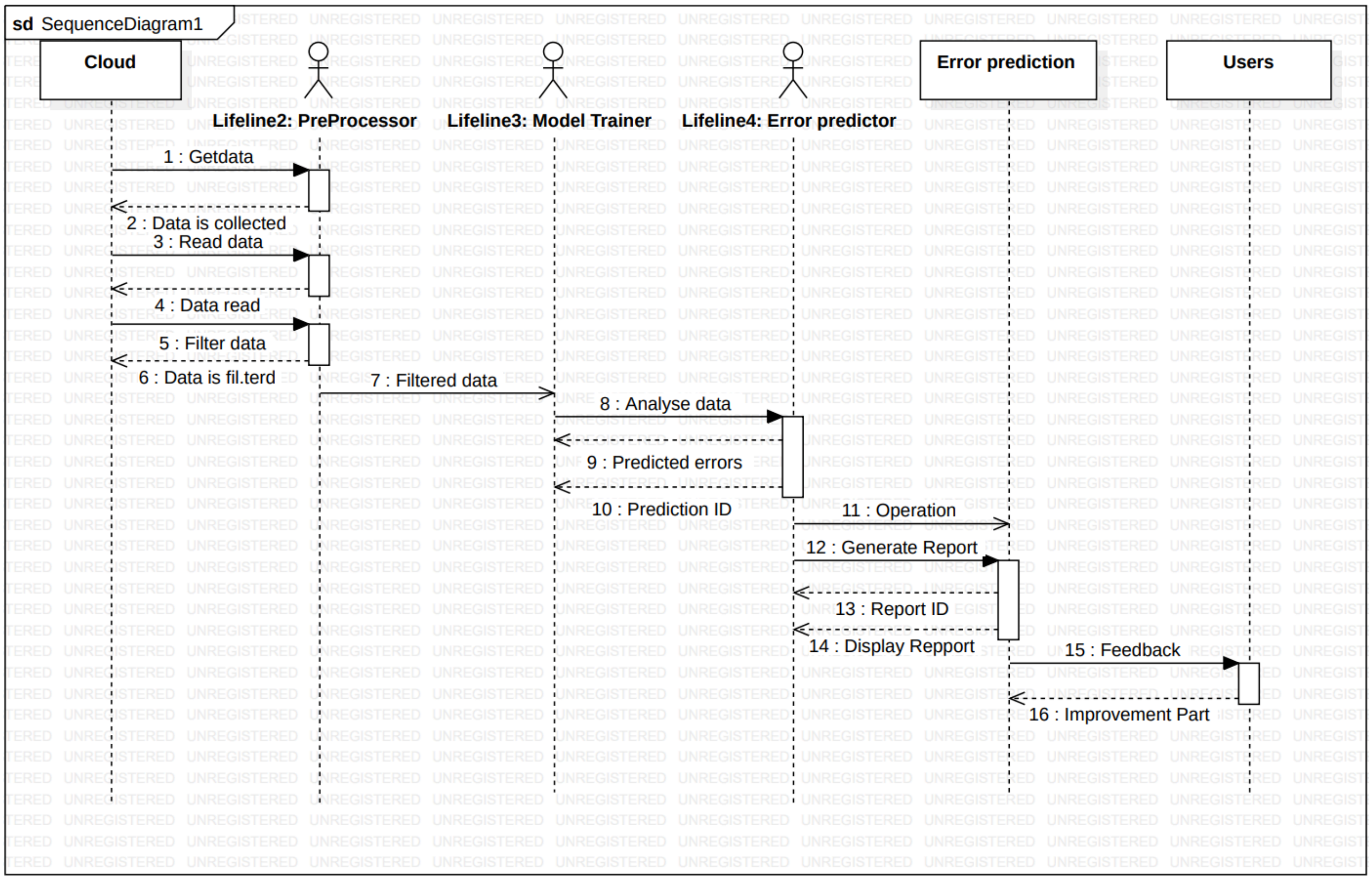
Principle: Use asynchronous, event-driven communication where appropriate.

Why: This enhances scalability and decouples services, allowing them to react to events as they occur. It supports real-time data processing and improves responsiveness.

CLASS DIAGRAM :



SEQUENCE DIAGRAM :



Test Strategy

a) Test Plan Document

Testing Type

Unit Testing: Verify individual components and functions.

Integration Testing: Ensure modules work together seamlessly.

System Testing: Validate the complete system against requirements.

Performance Testing: Assess the system's performance under various conditions.

User Acceptance Testing (UAT): Confirm the system meets user requirements.

Test Environment:

Development Environment: Local machines for unit and integration tests.

Staging Environment: Replicates production environment for system and UAT.

Production Environment: Live system for final validation.

Tools:

Unit Testing: pytest, unittest (Python)

Integration/System Testing: Selenium, Postman

Performance Testing: JMeter

CI/CD: Azure DevOps, GitHub Actions

b) Test Cases for 5 User Stories

User Story 1: Log Scanning

- **Happy Path:** Logs are successfully scanned and parsed.

    - **Test Case 1.1:** Verify that system logs are being scanned at scheduled intervals.
    - **Test Case 1.2:** Validate correct parsing of log entries.
- **Error Scenario:** Log files are missing or corrupted.
    - **Test Case 1.3:** Check the system's response to missing log files.
    - **Test Case 1.4:** Ensure the system handles corrupted log entries gracefully.

User Story 2: Error Classification

- **Happy Path:** Errors and warnings are correctly classified.
    - **Test Case 2.1:** Validate that errors are categorized by type and severity.
- **Error Scenario:** Unrecognized error types.
    - **Test Case 2.2:** Ensure the system logs unrecognized errors for further analysis.

User Story 3: Prediction Modeling

- **Happy Path:** Predictive models generate accurate forecasts.
    - **Test Case 3.1:** Verify that the system correctly predicts the recurrence of errors/warnings.
- **Error Scenario:** Inaccurate predictions.
    - **Test Case 3.2:** Test the system’s ability to learn from prediction inaccuracies and improve.

User Story 4: Reporting

- **Happy Path:** Generate comprehensive reports.
    - **Test Case 4.1:** Validate the generation of reports listing all detected errors/warnings.
    - **Test Case 4.2:** Confirm reports include prediction analysis.

- **Error Scenario:** Report generation failure.
    - **Test Case 4.3:** Ensure the system logs report generation errors and retries.

User Story 5: User Notifications

- **Happy Path:** Users receive timely notifications about potential issues.
    - **Test Case 5.1:** Verify that users are notified when a critical prediction is made.
- **Error Scenario:** Notification system failure.
    - **Test Case 5.2:** Check the system's fallback mechanism when notifications fail.

c) GitHub Repository Structure

Repository Name: predictive-maintenance

Project Structure:

```
predictive-maintenance/
├── docs/
│   ├── test-plans.md
│   └── architecture.md
├─ src/
│   ├── log_scanner/
│   │     └── log_scanner.py
│   ├── error_classifier/
│   │       └── classifier.py
│   ├── prediction_model/
│   │      └── model.py
│   ├── report_generator/
│   │      └── report.py
│   └── main.py
├── tests/
│     ├── test_log_scanner.py
│     ├── test_classifier.py
│     ├── test_model.py
│     └── test_report.py
├── .github/
│      └── workflows/
│             └── ci.yml
├── README.md
└── requirements.txt
```

Naming Conventions:

- **Directories:** lowercase with underscores (e.g., log_scanner)
- **Files:** lowercase with underscores (e.g., log_scanner.py)
- **Tests:** prefixed with `test_` (e.g., test_log_scanner.py)
- **Variables/Functions:** `snake_case`
- **Classes:** `CamelCase`

d) DevOps Architecture

DevOps Architecture Overview:

Version Control:

- **Tool:** GitHub
- **Description:** Repository for source code, test plans, and documentation.

Continuous Integration/Continuous Deployment (CI/CD):

- **Tool:** Azure DevOps, GitHub Actions
- **Description:** Automated pipeline for building, testing, and deploying code.
- **Pipeline Steps:**
  - **Build:** Compile code and dependencies.
  - **Test:** Run unit, integration, and system tests.
  - **Deploy:** Deploy to staging/production environments.
  - **Monitor:** Use Azure Monitor for continuous performance and health checks.

Infrastructure:

- **Tool:** Azure App Services.
- **Description:** Host the application and manage resources.
- **Components:**
  - **Web App:** Hosts the main application.
  - **Database:** Azure SQL for storing logs and prediction data.
  - **Storage:** Azure Blob Storage for log files.

Monitoring and Logging:

- **Tool:** Azure Monitor, Azure Log Analytics

Description: Track application performance and log errors for further analysis.

Associated Tools in Azure:

Azure Pipelines: For CI/CD.

Azure App Service: Hosting web applications.

Azure SQL Database: Storing application data.

Azure Storage: For log storage.

Azure Monitor: Monitoring and alerting on application performance.

Azure Log Analytics: Analyzing logs and metrics.

Deployment Architecture of the application

Overview of Deployment Architecture

The deployment architecture consists of:

- Data Sources (Sensors/IoT Devices)
- Edge Devices/Gateways
- Cloud Infrastructure
- Data Processing and Storage
- Predictive Models and Machine Learning Infrastructure
- User Interface

Components and Architecture Diagram

- **Data Sources (Sensors/IoT Devices)**
  - **Description:** Physical devices collecting data (temperature, vibration, pressure).
  - **Deployment:** On machinery/equipment.
- **Edge Devices/Gateways**
  - **Description:** Collect and preprocess data from sensors.
  - **Deployment:** Close to data sources to reduce latency.
- **Cloud Infrastructure**
  - Description: Central hub for data collection, storage, and processing.
  - **Deployment:** Hosted on cloud platforms (AWS, Azure, Google Cloud).
- **Data Processing and Storage**
  - **Description:** Handles data ingestion, cleaning, transformation, and storage.
  - **Deployment:** Cloud-native solutions (Amazon S3, Azure Blob Storage, Amazon RDS).
- **Predictive Models and Machine Learning Infrastructure**
  - **Description:** Machine learning models for predicting maintenance needs.
  - **Deployment:** Cloud-based platforms (AWS SageMaker, Azure ML).
- **User Interface**

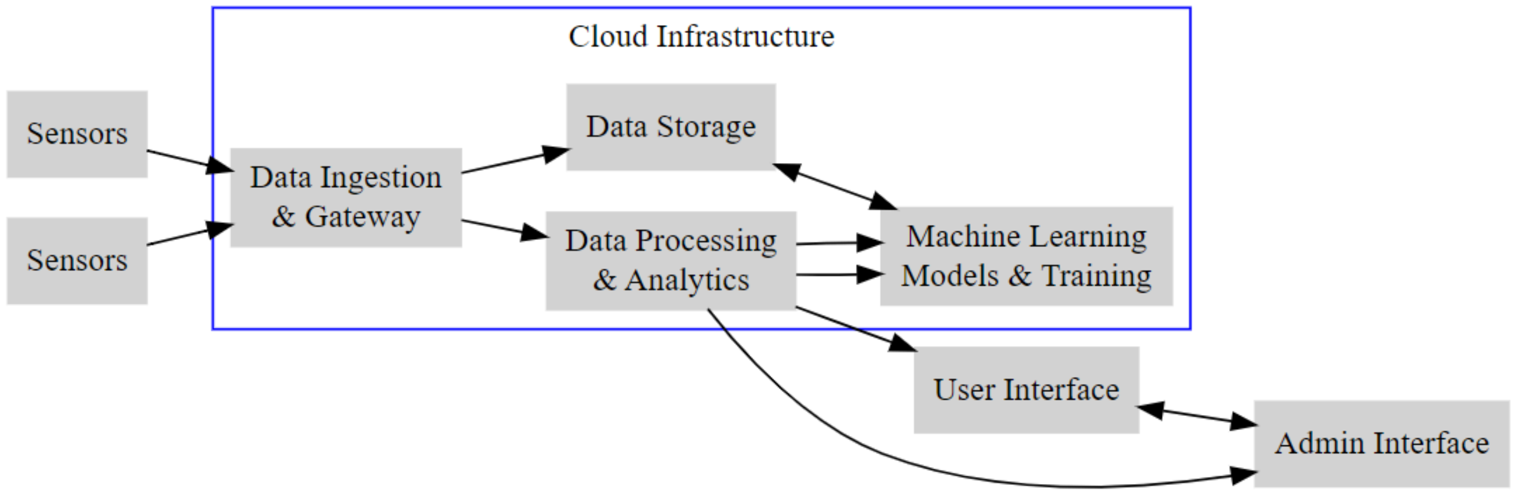
    - **Description:** Front-end applications for user interaction and data visualization.
    - **Deployment:** Hosted on cloud servers or web hosting platforms.

Deployment Considerations

- **Scalability:** Use scalable cloud services.
- **Reliability:** Design for high availability and fault tolerance.
- **Security:** Protect data in transit and at rest using encryption and secure communication.
- **Performance:** Optimize data pipelines and use edge processing.
- **Compliance:** Adhere to regulations (GDPR, HIPAA).

Example Technologies

- **Sensors/IoT Devices:** Bosch, Siemens, Honeywell
- **Edge Devices/Gateways:** Raspberry Pi, AWS IoT Greengrass, Azure IoT Edge
- **Cloud Infrastructure:** AWS, Azure, Google Cloud
- **Data Storage:** Amazon S3, Azure Blob Storage, Google Cloud Storage
- **Machine Learning:** AWS SageMaker, Azure ML, Google AI Platform
- **User Interface:** React.js, Angular, Vue.js; React Native, Flutter for mobile


Cloud Infrastructure
Sensors
Sensors
Data Ingestion & Gateway
Data Storage
Data Processing & Analytics
Machine Learning Models & Training
User Interface
Admin Interface