IoT-Based Environmental Monitoring System for Real-Time Air Quality Assessment

Wangyue Xu
Zheshuo Zhang
Wenzhou-kean university
CPS3962
Wenzhou,China
zhanzhes@kean.edu

line 1: 2nd Given Name Surname
line 2: *dept. name of organization (of Affiliation)*
line 3: *name of organization (of Affiliation)*
line 4: City, Country
line 5: email address or ORCID

Abstract*-This paper presents the design and implementation of an IoT-based environmental monitoring system aimed at tracking pollution levels, temperature, humidity, CO_2, light intensity, and PM2.5. The system uses a sensor network to collect real-time environmental data, stores it in a MySQL database, and provides instant visual feedback with threshold-based alerts. The software architecture is object-oriented, with UML models including use case, class, sequence, state, and design diagrams. The final system evaluates air quality into three categories: good, moderate, or poor, offering practical insights for environmental monitoring.

Keywords—IoT, environmental monitoring, air quality, real-time system, PM2.5, CO2 sensor, object-oriented design, MySQL database, threshold alert, UML modeling.

I. INTRODUCTION (*HEADING 1*)

Urbanization and industrialization have significantly impacted air quality, making real-time environmental monitoring an essential task. The presence of pollutants like CO2 and PM2.5 poses a threat to human health, especially in densely populated areas. Traditional monitoring systems are often expensive and lack the capability for localized, real-time updates. The objective of this project is to develop a low-cost, scalable, and efficient IoT-based system that monitors temperature, humidity, CO2, light intensity, and PM2.5 in real time. This project also employs object-oriented design to ensure modularity, reusability, and clarity in system implementation. By integrating a database, sensor network, and alert mechanism, the system provides actionable insights into air quality and environmental conditions.

II. RELATED WORK

Several IoT-based environmental monitoring systems have been proposed in recent years. Most of them rely on microcontroller platforms and provide basic monitoring functionality. However, many lack scalability, integration with object-oriented programming, and structured database storage.

Our system improves upon these by adopting a fully modular design and utilizing UML diagrams for clearer system architecture. Moreover, the use of MySQL allows for efficient querying and long-term data analysis, which is missing in many lightweight implementation

III. UML

We have drawn a total of use case diagrams, sequence diagrams, state diagrams, deployment diagrams, analysis class diagrams, design class diagrams, MySQL. The following are some ideas and analysis of the main diagrams

A. Use case diagram

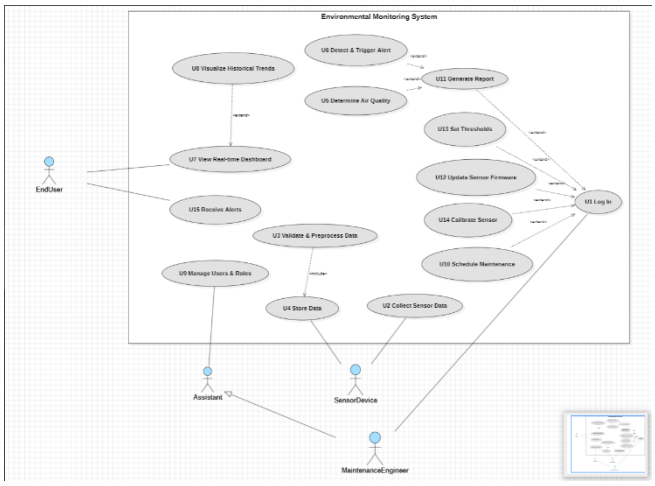


Figure 1

The use case diagram presented in Figure 1 illustrates the functional interactions between external actors and the Environmental Monitoring System. It defines the system's functional boundaries and outlines the primary responsibilities of different user roles.

The system supports four primary actors: EndUser, Assistant, MaintenanceEngineer, and SensorDevice.

The EndUser is permitted to view the real-time dashboard (U7), receive alerts (U15), and explore historical trends (U8).

The Assistant is granted extended privileges, such as managing users and roles (U9), in addition to viewing real-time information.

The MaintenanceEngineer is responsible for logging in (U1), updating sensor firmware (U12), calibrating sensors (U14), and scheduling maintenance (U10).

The SensorDevice automatically collects environmental

data (U2), initiating internal system processes.

The system's core use cases are as follows:

U1 Log In is a prerequisite for several administrative tasks, as indicated by the <<extend>> relationships leading to U11 (Generate Report), U12 (Update Sensor Firmware), U13 (Set Thresholds), and U14 (Calibrate Sensor).

U3 Validate & Preprocess Data includes U4 Store Data, forming the backbone of the data processing pipeline.

U5 Determine Air Quality and U6 Detect & Trigger Alert are central to the system's environmental intelligence, enabling automated classification and alerting mechanisms.

U7 View Real-time Dashboard is extended by U8 Visualize Historical Trends, enhancing the user's ability to explore long-term patterns.

This diagram outlines the entire system functionality from a user interaction perspective and provides the foundational structure for subsequent architectural design, including class, sequence, and state diagrams. It emphasizes modularity, clarity, and role-specific access, all essential in an object-oriented IoT-based monitoring solution.

.

B. Sequence diagram

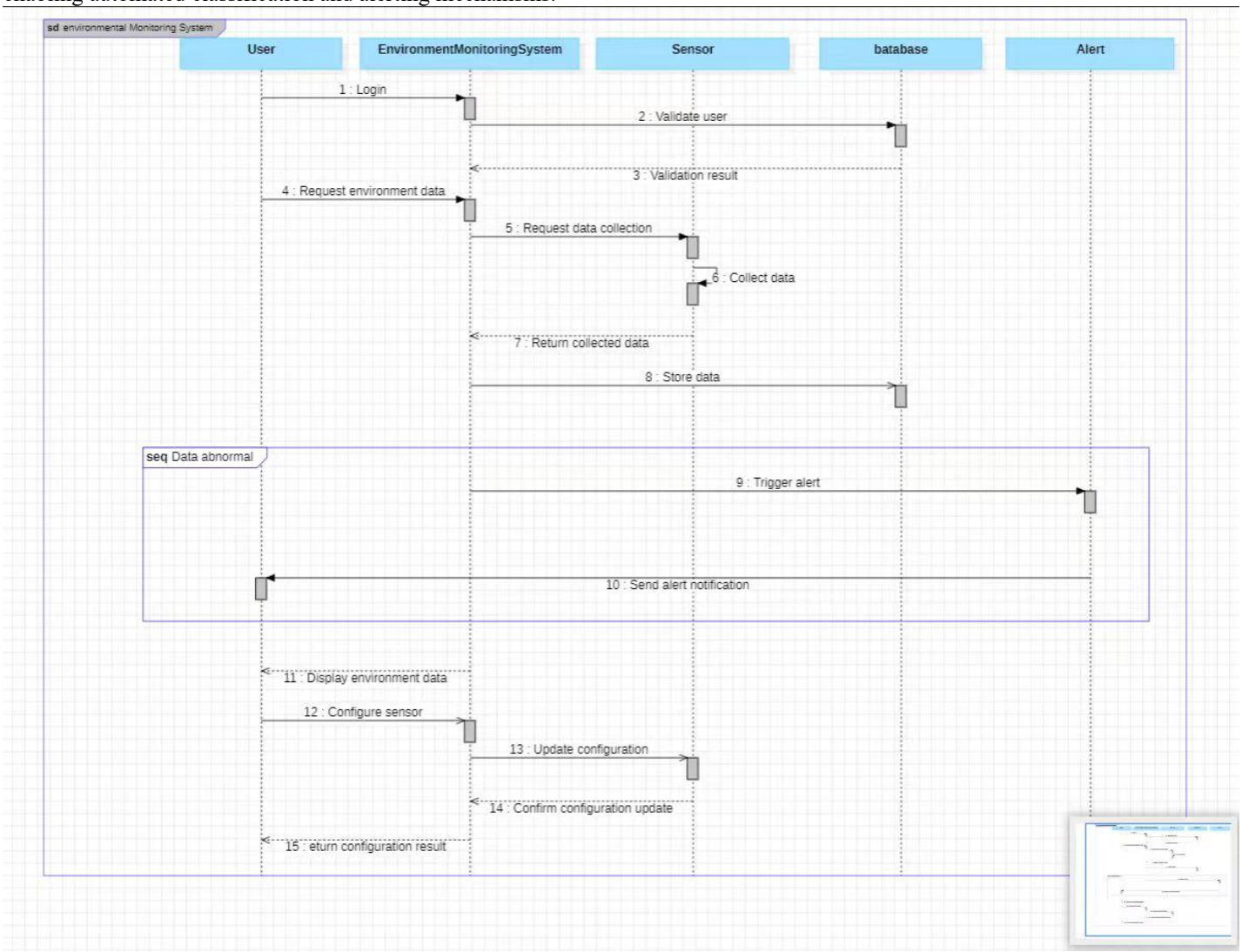


Figure 2

The sequence diagram illustrates the core workflow of the IoT-Based Environmental Monitoring System. It models the interactions between key system components including the user, the monitoring system, sensor devices, the database, and the alert module. The diagram emphasizes both standard operational flow and exceptional behavior when abnormal data is detected.

The sequence begins with the User initiating a login request, which the EnvironmentMonitoringSystem validates. Upon successful authentication, the user requests environmental data. The system then sends a request to the Sensor to collect data, which is subsequently returned and stored in the Database.

If the system detects that the collected data exceeds the predefined thresholds (e.g., for PM2.5 or CO_2), it triggers an

alert and delegates the Alert component to send a real-time notification to the user. The user may then inspect the displayed environmental information and optionally configure the sensor, such as adjusting calibration settings. These configuration commands are processed and confirmed through the system back to the user.

This sequence demonstrates the system's ability to:

Perform user authentication and validation.

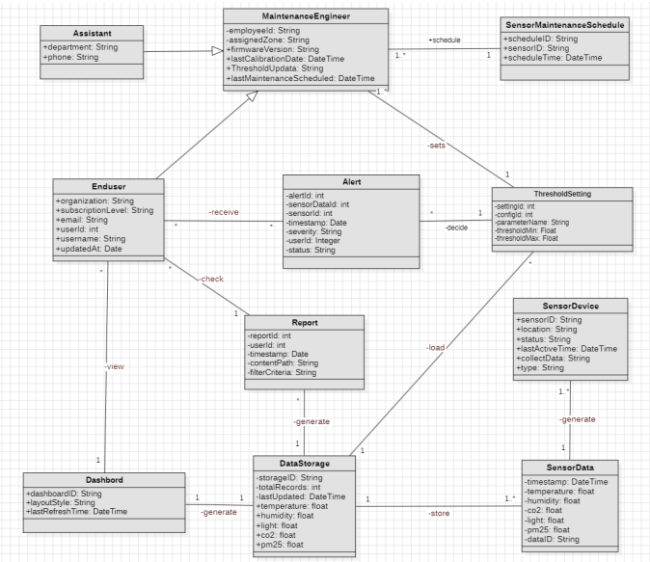
Enable real-time data acquisition and storage.

Support automated alert generation upon threshold violations.

Provide two-way interaction for sensor management and configuration.

C. Analysis Class Diagram

Figure 3



The analysis class diagram provides a comprehensive structural representation of the IoT-Based Environmental Monitoring System. It models the core domain entities and their interrelationships, offering a conceptual foundation for object-oriented design. Unlike earlier abstractions that relied on a common superclass for all user types, this refined version clearly separates user roles into three distinct classes: Assistant, Enduser, and MaintenanceEngineer. Each role encapsulates specialized attributes that reflect their functional responsibilities within the system. The MaintenanceEngineer class, in particular, holds operational data such as firmware version, calibration history, and threshold update timestamps—crucial for maintaining system accuracy and stability.

Environmental sensing is modeled through the SensorDevice class, which maintains information about device type, location, and operational status. These devices continuously generate SensorData entries that record timestamped environmental measurements including temperature, humidity, CO_2, light, and PM2.5. Each data entry is evaluated against a corresponding ThresholdSetting, which defines the acceptable minimum and maximum values for each parameter. When a reading exceeds its threshold, an Alert is generated and linked to both the relevant SensorDevice and the affected Enduser, ensuring timely notification and risk mitigation.

Collected sensor data is stored in the DataStorage class, which also tracks the total number of records and the last update time, serving as a central data repository. The system further supports analytical and decision-making functions

through the Report class, which allows users to summarize historical data and alerts based on customizable filters. Additionally, the Dashboard class enables real-time visualization of monitored data, enhancing user accessibility and system transparency. To ensure long-term reliability, scheduled sensor maintenance tasks are managed via the SensorMaintenanceSchedule class, which logs calibration appointments and responsible engineers.

Overall, the class diagram embodies core object-oriented principles such as modularity, encapsulation, and explicit role separation. This design promotes system scalability, maintainability, and clarity, making it well-suited for both future development and real-world deployment.

D. State Machine Diagram

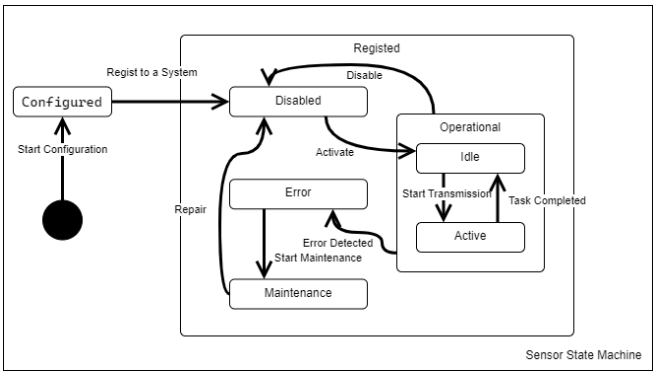


Figure 3

Figure 4 presents the state machine diagram for the Sensor entity within the proposed Environmental Monitoring System. This diagram is instrumental in formally specifying the discrete operational states of a sensor throughout its lifecycle, along with the events or conditions that precipitate transitions between these states. Such a model provides a precise definition of sensor behavior under various operational scenarios, including normal function, configuration, error conditions, and maintenance activities, thereby contributing to the verifiable and predictable operation of individual monitoring nodes within the IoT network.

The sensor lifecycle commences from an initial pseudostate, representing the point before the sensor object is formally initialized within the system's context. The first explicit state transition is triggered by the Start Configuration event, leading to the Configured state. This state represents the preliminary phase where essential parameters, such as network settings or basic identification, might be applied to the sensor hardware or its software representation, often preceding its full integration into the monitoring network. Subsequently, upon a successful registration event, denoted as Regist[er] to a System, the sensor transitions into the main operational boundary, initially entering the Disabled state. Within the system's management context, the Disabled state signifies that the sensor is recognized and provisioned but is currently inactive and not participating in data collection or transmission cycles. This state allows for administrative control over sensor activation.

Activation of the sensor is initiated by an Activate event, which moves the sensor from the Disabled state into the Operational composite state. The Operational state encapsulates the sensor's normal functioning modes. It contains two distinct substates: Idle and Active. Upon entering the Operational state, the sensor defaults to the Idle substate. In this substate, the sensor is powered, operational, and ready to perform its sensing task but is currently awaiting a trigger, such as a scheduled time event or a direct command. The Start Transmission event signals the transition from Idle to the Active substate. During the Active substate, the sensor is actively performing its core function, which typically involves acquiring sensor readings (e.g., temperature, humidity), potentially performing local processing, and transmitting the resulting data to the central system or gateway. Once this task cycle is finished, the Task Completed event occurs, transitioning the sensor back to the Idle substate to await the next activation trigger. Furthermore, the system retains the ability to administratively deactivate a functioning sensor; a Disable event received while in the Operational state will transition the sensor back to the Disabled state, effectively removing it from active duty.

The state machine also explicitly models pathways for handling non-nominal conditions. Should an internal fault or external condition leading to erroneous operation be detected while the sensor is in the Operational state (regardless of whether it is Idle or Active), an Error Detected event transitions the sensor to the Error state. This state indicates a malfunction requires attention and prevents the sensor from performing normal operations or providing potentially incorrect data. From the Error state, intervention is typically required. A Start Maintenance event, likely initiated by a system administrator or automated diagnostic routine, moves the sensor to the Maintenance state. This state represents the phase where diagnostics, troubleshooting, firmware updates, or physical repairs are conducted. Upon successful completion of these maintenance activities, a Repair event signifies the resolution of the fault, transitioning the sensor back to the Disabled state. From Disabled, it can then be reactivated via the Activate event to rejoin the operational pool.

In summary, Figure 4 provides a formal model of the Sensor's behavior through distinct states (Configured, Disabled, Operational [Idle, Active], Error, Maintenance) and well-defined transitions triggered by specific events. This detailed state modeling facilitates the implementation of sensor management logic within the Environmental Monitoring System, allowing for consistent handling of sensor activation, deactivation, data collection cycles, error reporting, and maintenance procedures. It forms a basis for developing reliable control software and understanding the potential status of any given sensor within the deployed IoT system at any point in time.

E. Class Diagram (Design level)

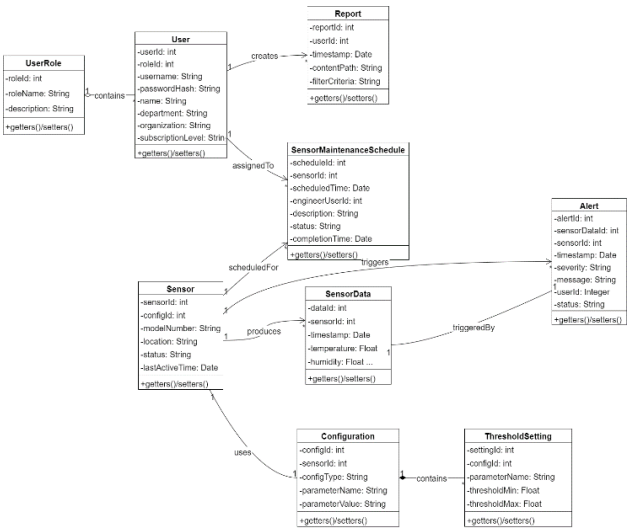


Figure 4a: Core Domain Model

Figure 5a provides a detailed representation of the Core Domain Model for the Environmental Monitoring System. This diagram meticulously outlines the fundamental data entities, including User, UserRole, Sensor, SensorData, Alert, Report, Configuration, SensorMaintenanceSchedule, and ThresholdSetting. Crucially, it displays the complete set of attributes and common operations (represented by +getters()/setters()) for each entity, offering a precise definition of the data structure managed by the system.

The diagram focuses on the structural relationships that define how these core entities are interconnected. These relationships, such as the one-to-many aggregation (o--) between UserRole and User, the one-to-many association (-->) indicating a Sensor produces many SensorData records, the one-to-one association (--) between Alert and SensorData, and the one-to-many composition (*--) showing ThresholdSettings as part of a Configuration, are clearly annotated with multiplicities. This detailed entity-relationship view serves as the foundational data schema for the application, directly informing database design and providing the data context for higher-level service interactions.

Figure 5b illustrates the crucial interactions between the Business Logic Layer (Services) and the Data Access Layer (Repositories). This diagram presents the full definitions, including key public methods, for major Service classes (e.g., UserService, SensorDataService) and their corresponding Repository implementations (e.g., UserRepository, SensorDataRepository). The inclusion of method signatures provides insight into the specific operations offered by each service and repository.

The dependencies (-->) clearly depict the application of the Repository pattern: Service classes depend on abstractions provided by Repositories to perform data operations (e.g., SensorDataService --> SensorDataRepository : persists). To provide necessary context for repository functions, the full definitions of the primary Domain Model entities managed by these repositories (e.g., User, Sensor, SensorData) are also included in this view. Note that Service and Repository classes typically lack explicit data attributes, as their state is managed through dependencies. This diagram emphasizes the decoupling achieved, where services orchestrate business logic using repositories that encapsulate data access details. However, including full entity definitions can lead to visual complexity, highlighting the trade-off between informational completeness and diagrammatic clarity in this specific view.

Figure 5c details the dependencies between the Presentation Layer (Controllers) and the Business Logic Layer (Services). This view provides the complete definitions, including method signatures, for all Controller classes (e.g.,

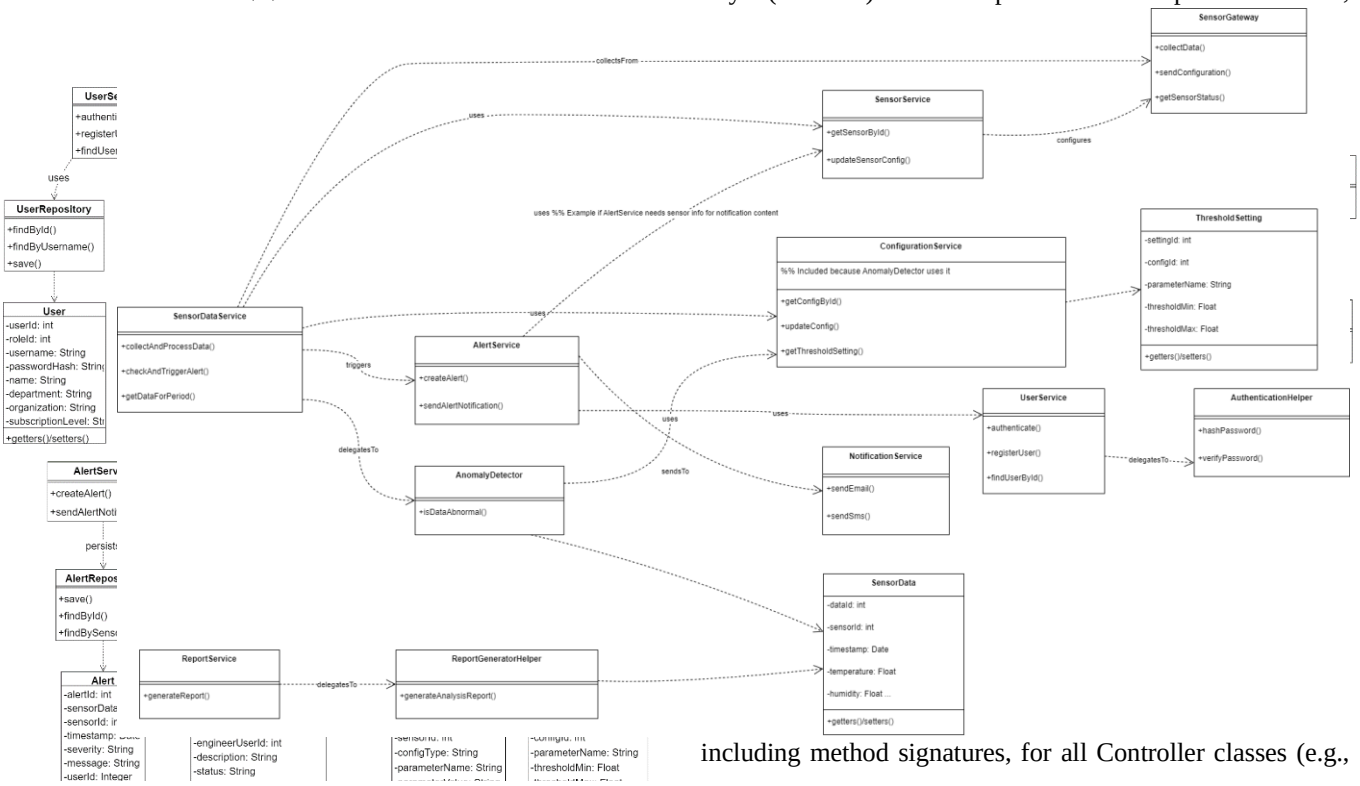


Figure 5b: Service Layer and Data Access Interaction

UserController, SensorController) and the Service classes they directly interact with (e.g., UserService, SensorService).

The diagram focuses solely on the dependency relationships (-->) flowing from Controllers to Services, illustrating the primary control flow for handling incoming requests. For example, UserController --> UserService : uses signifies that HTTP requests related to user actions (login, registration) received by UserController are delegated to UserService for processing. This layered interaction ensures that Controllers remain lightweight, focusing on request handling and response formatting, while the core application logic resides within the Service layer. The detailed class definitions clarify the specific service methods invoked by each controller, mapping directly to the system's exposed functionalities and use case entry points.

Figure 5d: Integration and Supporting Components Interaction

Figure 5d illustrates the collaborative aspects of the Service layer, focusing on its interactions with external systems, integration points, and key supporting utility or algorithmic components. It presents the full definitions for the relevant Service classes, the Utility classes (AuthenticationHelper, AnomalyDetector, ReportGeneratorHelper), and the Integration/External classes (SensorGateway, NotificationService). Key Domain Model entities (SensorData, Configuration, ThresholdSetting) directly utilised in these interactions are also fully defined for context.

The dependencies (-->) highlight several interaction patterns: Services utilizing Gateways for external communication (SensorDataService --> SensorGateway), Services delegating tasks to Utilities (UserService --> AuthenticationHelper), Utilities potentially relying on Services (AnomalyDetector --> ConfigurationService), and Utilities or Services directly using specific Domain Models (AnomalyDetector --> SensorData). Furthermore, critical service-to-service dependencies relevant to these integration or support scenarios (e.g., SensorDataService --> AlertService : triggers) are included. This diagram provides insight into how the system interfaces with its environment and leverages specialized components, showcasing the modularity and distribution of responsibilities beyond the primary service-repository or controller-service interactions.

F. MySQL Model

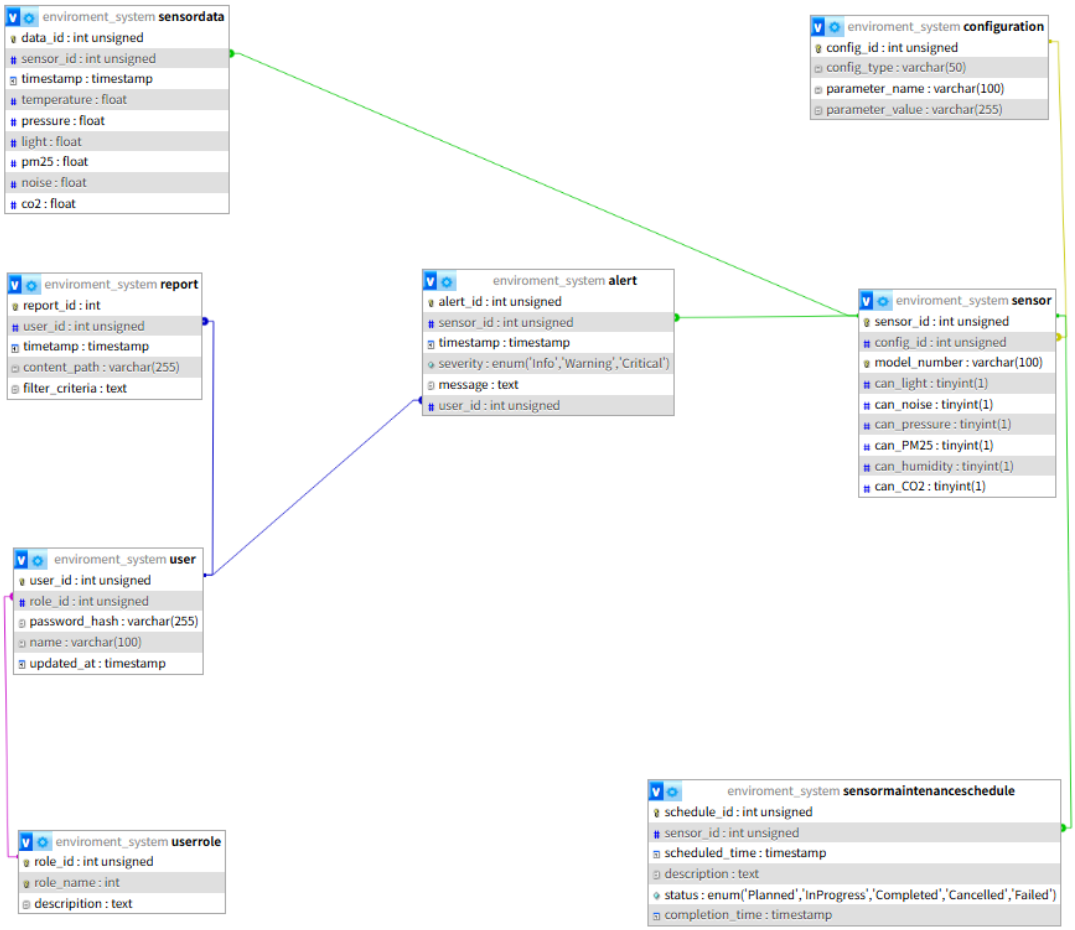


Figure 5

Figure 6 illustrates the physical database schema designed to support the data persistence requirements of the IoT-Based Environmental Monitoring System. This schema is implemented using MySQL and translates the logical domain model (previously shown in Figure 5a) into a concrete set of relational tables, defining the specific data types, constraints, and relationships necessary for efficient storage and retrieval.

The schema comprises several key tables corresponding to the core entities of the system:

environment_system_sensordata: Stores the time-series environmental readings collected from sensors. It includes columns for various parameters such as temperature, humidity, pm25, etc. (all typed as float), along with a timestamp and a foreign key sensor_id linking each reading back to its source sensor. data_id serves as the primary key.

environment_system_sensor: Represents the physical sensor devices. It contains attributes like model_number and flag columns (e.g., can_light, can_noise, typed as tinyint(1)) indicating the sensor's measurement capabilities. It uses sensor_id as its primary key and includes a foreign key config_id referencing the configuration table.

environment_system_user and environment_system_userrole: Manage user accounts and roles, respectively. The user table stores user credentials (using password_hash), name, and links to a role via the role_id foreign key. The userrole table defines the available roles.

environment_system_alert: Records alert events generated when sensor data exceeds predefined thresholds. It includes a severity level stored efficiently using an enum type, a descriptive message (text), and foreign keys sensor_id and user_id associating the alert with the triggering sensor and potentially a relevant user.

environment_system_report: Stores metadata about generated reports, including the user_id of the user who generated it, a timestamp, the content_path where the report file is stored, and the filter_criteria used.

environment_system_sensormaintenanceschedule: Tracks scheduled maintenance activities for sensors. It contains scheduling details (scheduled_time, description), the status of the task (using an enum type for states like 'Planned', 'Completed'), and foreign keys like sensor_id.

environment_system_configuration: Stores configuration parameters referenced by sensors via the sensor.config_id

foreign key. It holds key-value pairs identifiable by config_type and parameter_name.

The relationships between tables are enforced using foreign key constraints, clearly depicted by the connecting lines in Figure 6. Key relationships include the one-to-many links from sensor to sensordata, alert, and sensormaintenanceschedule (via sensor_id), from user to report and alert (via user_id), and from userrole to user (via role_id). These relationships ensure data integrity and allow for efficient querying across related entities.

Overall, Figure 6 provides the concrete data persistence structure for the system. The choice of appropriate data types (e.g., int unsigned, timestamp, float, varchar, text, enum, tinyint(1)) and the establishment of relational integrity through primary and foreign keys ensure that the application's data is stored reliably and can be effectively managed by the data access layer components (Repositories) outlined in the design diagrams.

G. User Interface

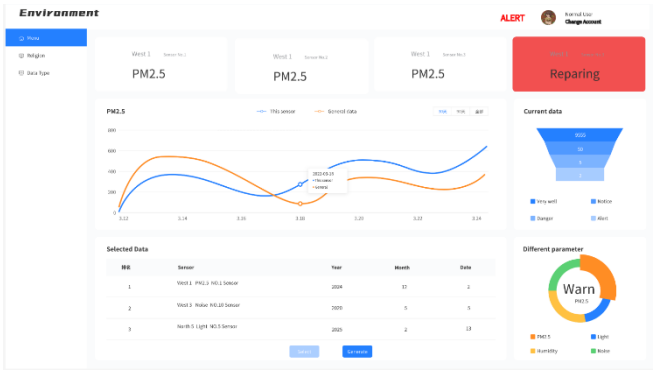


Figure 6

Figure 7 illustrates the primary user interface (UI) designed for the Environmental Monitoring System, serving as the main interaction point for users such as EndUsers and Assistants. The UI adopts a dashboard layout, prioritizing intuitive access to real-time data, historical trends, and sensor status information.

The interface is organized into distinct functional areas. A persistent left-hand navigation sidebar provides access to different system views or data types (e.g., 'Region', 'Data Type'). The main content area prominently features summary cards displaying key parameters (e.g., PM2.5) for individual sensors, including visual indicators for sensor status (e.g., 'Repairing'). Below the cards, a central line graph facilitates the visualization of historical trends, allowing comparison between a specific sensor's data ('This sensor') and general baseline data over a selectable time period. Interactive elements, such as tooltips displaying specific data points, enhance usability. Further down, a tabular section ('Selected Data') enables detailed examination of specific data records, with functionalities for selection and report generation ('Generate').

A right-hand sidebar provides summarized, at-a-glance information. This includes widgets visualizing the distribution of current data across predefined status levels ('Current data', indicating levels like 'Notice', 'Danger', 'Alert') and summarizing the status or distribution of different monitored parameters ('Different parameter', highlighting a 'Warn' state for PM2.5 in the example). The header bar consistently displays the system title, an alert indicator, and user account information.

H. Deployment Diagram

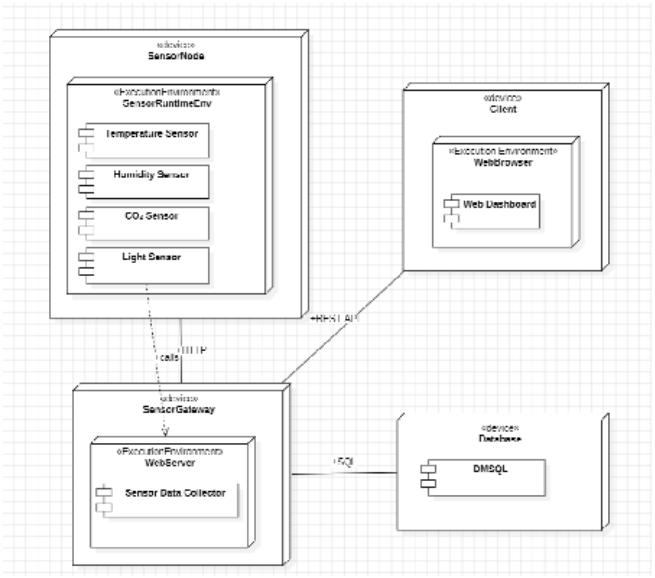


Figure 8

The deployment diagram models the physical architecture of the IoT-Based Environmental Monitoring System, illustrating how software components are distributed across hardware nodes and how they interact within the execution environment. This structure reflects a modular and scalable deployment strategy that enables real-time environmental data collection, processing, and presentation.

At the sensing layer, environmental data is captured by a SensorNode device, which hosts an execution environment labeled SensorRuntimeEnv. This environment integrates multiple embedded sensors including temperature, humidity, CO_2, and light sensors. These sensors operate autonomously to gather environmental parameters and transmit the collected data to a central processing node. Communication between the SensorNode and the processing layer is handled via HTTP requests.

The data is received by a separate device known as the SensorGateway, which contains a WebServer running within its own execution environment. Inside the WebServer, a dedicated Sensor Data Collector module processes incoming sensor data and acts as the system's core data handler. The gateway then pushes the processed data to the Database device using SQL. This database, labeled DMSQL, stores historical sensor readings and supports complex queries for monitoring, reporting, and alert generation.

On the user interaction side, clients access the system through a Client device that runs a WebBrowser. The browser hosts a Web Dashboard application, which communicates with the WebServer using REST API calls. This allows users to retrieve real-time environmental data, view trends, and receive alerts without directly accessing the backend infrastructure.

This deployment design ensures that each layer of the system—sensing, processing, storage, and user interaction—is cleanly separated and independently scalable. It enables

robust performance, supports fault isolation, and allows for future enhancements such as adding new sensors or expanding dashboard functionalities. Overall, the deployment architecture reflects a service-oriented and loosely coupled design, capable of supporting reliable environmental monitoring in real-time scenarios.

IV. Conclusion

This paper has presented the comprehensive design of an IoT-based system tailored for real-time environmental monitoring, focusing on critical air quality parameters including temperature, humidity, CO_2, light intensity, and PM2.5. By systematically applying an object-oriented methodology, the system's architecture, behavior, and data structures were rigorously defined using a suite of UML diagrams. These included use case diagrams (Figure 1) to capture functional requirements and actor interactions, sequence diagrams (Figure 2) to illustrate key operational workflows, state machine diagrams (Figure 4) to model sensor lifecycle dynamics, detailed multi-layered design class diagrams (Figures 5a-d) specifying the software components and their collaborations, and a corresponding physical MySQL database schema (Figure 6) for persistent data storage.

The proposed design successfully integrates essential functionalities: automated sensor data acquisition, validation, and storage; real-time data processing and air quality assessment (categorized as good, moderate, or poor); configurable threshold-based alerting; and user-friendly visualization through a web-based interface (Figure 7) offering dashboards and historical trend analysis. The adoption of a layered architecture (Controllers, Services, Repositories, Domain Models, Utilities, Gateways) ensures modularity, maintainability, and scalability, addressing limitations found in some existing systems.

In summary, this work provides a robust and well-documented blueprint for developing an effective, low-cost IoT solution for localized environmental monitoring. The detailed object-oriented design and database schema offer a solid foundation for subsequent implementation, testing, and deployment. The resulting system has the potential to provide valuable, actionable insights into environmental conditions, contributing to public health awareness and facilitating timely responses to air quality issues. Future work would involve the physical implementation of this design, performance evaluation under real-world conditions, and potentially extending the system with advanced data analytics or machine learning capabilities for predictive modeling.