VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY

HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY

FACULTY OF COMPUTER SCIENCE AND ENGINEERING

REPORT

MULTIDISCIPLINARY PROJECT

YOLOHOME: SMART HOME

SUPERVISOR: Phan Trung Hiếu

STUDENT 1: Mai Quốc Thăng - 2252761

STUDENT 2: Lê Hoàng Thịnh - 2252775

STUDENT 3: Trần Hoàng Phúc - 2252647

STUDENT 4: Mai Hoàng Phúc - 2252635

STUDENT 5: Nguyễn Bình Nguyên - 2252545

STUDENT 6: Võ Hoàng Long - 2053192

STUDENT 7: Nguyễn Hồng Phúc - 2252639

HO CHI MINH CITY, 05/2025

Preface

In an era where living spaces increasingly become extensions of digital life, the boundary between technology and home gradually fades. The *IntelliHome* project embodies our vision for the future of living spaces—where homes are not merely shelters, but actively adapt to the needs and preferences of people.

This multidisciplinary project combines knowledge from various fields such as environmental monitoring, automation, user experience design, and data analytics, aiming to build a unified system that can transform ordinary living spaces into responsive environments. Our approach centers around three core principles: **accessibility**, **adaptability**, and **awareness**.

A modern home must be accessible to all residents regardless of technological proficiency. Through intuitive interfaces and thoughtful interaction design, *IntelliHome* provides easy control over the environment. The system learns established routines yet still allows manual control, striking a balance between automation and user agency. Most importantly, the system maintains awareness of environmental conditions, enabling users to make more informed decisions in managing their living spaces.

As households increasingly face rising energy costs and growing environmental consciousness, solutions like *IntelliHome* demonstrate how automation can enhance convenience while minimizing resource consumption. The scheduling functions reflect our understanding that modern life demands flexibility—the home should adapt to us, not the other way around.

This project is the result of countless hours of teamwork, collaboration among members with diverse backgrounds and expertise. We overcame many challenges to integrate various elements into a coherent user experience. The outcome is a prototype that showcases the potential of smart home solutions to practically improve daily quality of life.

Figure 1: *Smart Home*

Contents

1 Introduction

1.1 Project Overview

The IntelliHome project represents a comprehensive smart home solution, designed to transform traditional living spaces into intelligent environments that respond flexibly to the needs and preferences of occupants. In recent years, the concept of smart homes has evolved from basic automation systems into sophisticated ecosystems that integrate environmental sensing, device control, personalized scheduling, and data analytics. Our project addresses this evolution by developing a unified system that enhances comfort, convenience, and energy-saving potential for modern households.

IntelliHome focuses on creating a seamless experience where technology operates quietly yet delivers meaningful benefits to users. By monitoring environmental conditions such as temperature, humidity, and light, the system builds awareness of the living space. This awareness is combined with the intelligent control of connected devices to create adaptive environments that align with established habits while respecting user preferences.

1.2 Project Scope and Objectives

The scope of the IntelliHome project includes the development of a complete smart home management system with the following key objectives:

1. **Environmental Monitoring:** Deploy a reliable system for continuously monitoring indoor environmental parameters, giving residents real-time insights into their living space.
2. **Smart Device Control:** Develop an intuitive interface for monitoring and controlling household devices, supporting both manual control and automatic operation based on predefined conditions.
3. **Scheduling and Automation:** Enable flexible scheduling capabilities that allow devices to operate based on time, environmental conditions, or usage patterns.
4. **Data Visualization:** Design dashboards that present environmental data in an understandable manner, helping users identify trends and patterns in their homes.
5. **User-Centered Design:** Ensure the entire system is accessible to users with varying levels of technical expertise through a user-friendly interface and simple interaction models.
6. **System Integration:** Establish a synchronized framework where components work harmoniously, avoiding the fragmentation common in many current smart home solutions.

Advanced features such as voice control, integration of external weather data, and predictive capabilities based on machine learning are not within the current scope, but the system architecture supports the future extension of these functionalities.

1.3 Technologies and Components

The IntelliHome project integrates various technologies and components to create a unified smart home management system:

1.3.0.1 Hardware components:

- Environmental sensors for monitoring temperature, humidity, and light levels
- Device control modules
- A central processing unit to coordinate the system
- Network infrastructure for communication between devices

1.3.0.2 Software architecture:

- Frontend user interface with responsive cross-platform design
- Backend services for data processing, storage, and device communication
- A database system for storing and retrieving environmental data
- Communication protocols to ensure reliable device interaction

1.3.0.3 User interface components:

- Dashboard displaying environmental data
- Device control panel with manual override functionality
- Scheduling interface for automation configuration
- Historical data viewer with visual charts

The integration of these components results in a system that balances technological sophistication with user accessibility, enabling residents to enjoy the benefits of a smart home environment without requiring deep technical knowledge for daily operation.

2 System Architecture

2.1 Overall System Design

The IntelliHome system architecture follows a modern client-server model with a clear separation between the presentation, application logic, and data management layers. At its core, the system comprises a Vue.js-based frontend that communicates with a Flask-based API server, which connects to a local SQLite database and external IoT services.

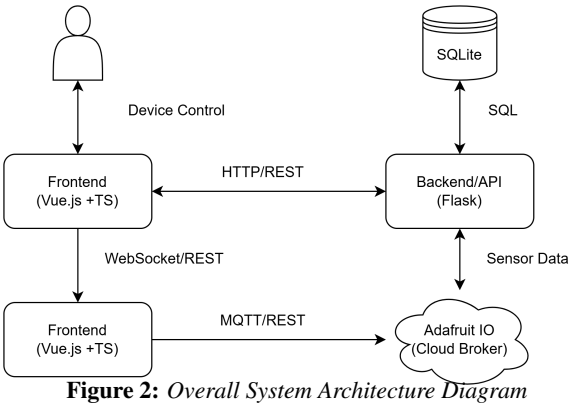

Figure 2: *Overall System Architecture Diagram*

This architecture emphasizes modularity, allowing for the independent development and testing of components while maintaining overall functionality through well-defined interfaces. The design supports key system requirements: real-time environmental monitoring, device control, and automated scheduling.

2.2 Frontend Architecture

The frontend is implemented using Vue.js 3 in combination with TypeScript, providing strong typing and an enhanced development experience. The component structure is logically organized, where specialized components handle specific parts of the interface:

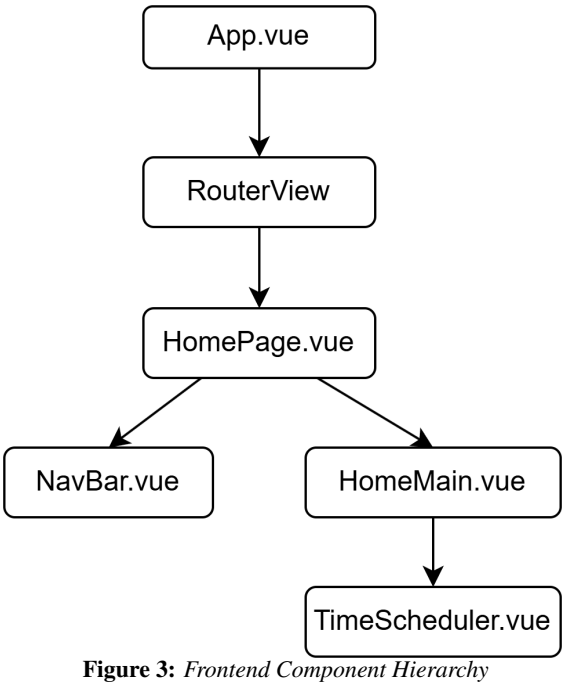

Figure 3: *Frontend Component Hierarchy*

Vue's reactivity system ensures the user interface is efficiently updated as environmental data or device status changes, creating a smooth interactive experience. The router manages navigation between system functionalities

(home control, security, settings) without full page reloads.

2.3 Backend Services

The backend architecture consists of two main concurrent Python services:

Data Collection Service: Implemented in `data_collector.py`, this service periodically fetches environmental data from IoT sensors via the Adafruit IO API. It processes parameters such as temperature, humidity, and light, storing them in a structured database to preserve historical records.

API Server: The `server.py` module implements RESTful endpoints that act as a communication bridge between the frontend and the data layer. This service handles data queries, time-series analysis, and delivers aggregate metrics through statistical processing.

2.4 Data Flow Architecture

Data flows through the system in a well-defined pattern:

$$\text{IoT Sensors} \rightarrow \text{Data Collector} \rightarrow \text{Database} \rightarrow \text{API Server} \rightarrow \text{Frontend} \tag{1}$$

The bidirectional nature of device control introduces an additional data flow, where user actions propagate from the frontend through the API to affect physical devices:

$$\text{User Interface} \rightarrow \text{API Server} \rightarrow \text{Device Control} \rightarrow \text{Physical Environment} \tag{2}$$

2.5 Storage Design

The storage architecture uses SQLite for its lightweight and self-contained characteristics. The database includes dedicated tables for each type of parameter:

The `humidity_readings`, `temperature_readings`, and `light_readings` tables store measured values along with timestamps and room identifiers, enabling spatial and temporal data analysis and visualization. This design supports efficient queries across time and location ranges.

2.6 Scheduler Architecture

The scheduling system implements time-based control, automatically toggling devices on or off based on user-defined schedules. This component:

- Stores schedule configurations in the browser's `localStorage`
- Compares time conditions against the current system time
- Triggers state changes when time thresholds are exceeded
- Provides visual feedback on the user interface

The scheduler runs entirely on the client side, leveraging JavaScript's timing capabilities to ensure precise control without continuous server communication.

2.7 Communication Protocol

System components primarily communicate via HTTP/RESTful APIs, using JSON as the data exchange format. This approach offers several advantages:

$$\text{Compatibility} + \text{Simplicity} + \text{Statelessness} = \text{Maintainable Architecture} \tag{3}$$

API endpoints follow a resource-oriented design pattern, with clear semantics for accessing environmental metrics, historical data, and room information.

2.8 Technology Integration

The system architecture smoothly integrates various technologies: Vue.js and TypeScript for frontend development, Flask and SQLite for the backend, and Tailwind CSS for responsive UI design. This tech stack strikes a balance between runtime performance and development efficiency, while ensuring high compatibility during deployment.

3 Design Pattern

3.1 Creational Patterns

Component Factory Pattern

The project leverages Vue's component-based architecture, which implements a factory pattern for component creation. Vue components are created and instantiated through Vue's built-in component factory mechanism. This is evident in the `HomePageComponents` directory, which contains reusable components like `TimeScheduler.vue`, `HomeMain.vue`, and `NavBar.vue`. These components are imported and instantiated in their parent components. For instance, `HomePage.vue` imports and instantiates `NavBar` and `HomeMain` components, creating a hierarchy of UI elements.

Lazy Loading Pattern

The project implements lazy loading through Vue Router's dynamic imports for component loading. In `router/index.ts`, components are loaded using dynamic imports such as `() => import("@/view/HomePage.vue")`. This approach significantly reduces the initial load time of the application by loading components only when needed, enhancing performance and user experience. This is particularly valuable for larger applications with many routes and components.

3.2 Structural Patterns

Composition Pattern

The application is structured following the composition pattern, where components are composed of smaller, more focused components to form a tree-like structure. `HomePage.vue` composes `NavBar.vue` and `HomeMain.vue`, while `HomeMain.vue` further includes `TimeScheduler.vue`. This compositional approach promotes code reusability and separation of concerns, allowing developers to build complex UIs from simpler, well-defined components.

Adapter Pattern

The `scheduleService.ts` file implements an adapter pattern, serving as an interface between raw `localStorage` data and the component-friendly format. Methods such as `shouldDeviceBeOn()` and `getAllSchedules()` transform raw `localStorage` data into structured application data that components can consume. This decouples the storage mechanism from UI components, enabling future changes without disrupting dependent logic.

3.3 Behavioral Patterns

Observer Pattern

Vue's reactivity system inherently implements the observer pattern, with reactive state being observed by components. The use of `ref()` and `computed()` in components creates reactive state that automatically triggers UI updates. In `TimeScheduler.vue`, `schedule` and `scheduleActive` are reactive states that update the UI whenever they change, simplifying state management and ensuring UI consistency.

Mediator Pattern

Parent components in the application act as mediators between child components. `HomePage.vue` mediates between `NavBar` and `HomeMain` components by controlling the sidebar visibility state. This mediator role ensures that components do not communicate directly with each other, reducing coupling and improving maintainability.

Command Pattern

Device operations in the application are encapsulated as commands. In `HomeMain.vue`, the `toggleDevice()` function encapsulates the logic to toggle a device's state. This abstraction allows actions to be extended, logged, or potentially undone without modifying the calling code, providing a clear and consistent interface for device operations.

3.4 Architectural Patterns

Repository Pattern

The `scheduleService.ts` module implements a repository pattern for schedule data, providing a centralized access point. Methods like `getAllSchedules()`, `shouldDeviceBeOn()`, and `initializeScheduledDevices()` abstract the data storage and retrieval logic, promoting a clean separation between business logic and data access.

Local Storage Cache Pattern

The application uses the browser's `localStorage` as a client-side cache for device schedules, forming a local storage cache pattern. This is evident in both `TimeScheduler.vue` and `scheduleService.ts`, where `localStorage` is accessed for persisting and retrieving data. This design enhances responsiveness and reduces the need for server interaction, especially across page reloads.

4 Hardware

4.1 Hardware components

In this project, we use several key hardwares components that works together to monitor environmental conditions and respond accordingly:

1. **Main controller: BBC micro:bit:** The BBC micro:bit is a compact and versatile microcontroller that serves as the brain of the system. It features a 32-bit ARM Cortex-M0 processor, built-in Bluetooth Low Energy (BLE), and various input/output (I/O) pins that facilitate communication with external sensors and actuators. The micro:bit is chosen for its ease of use, compact design, and extensive educational support, making it ideal for prototyping and learning embedded systems.

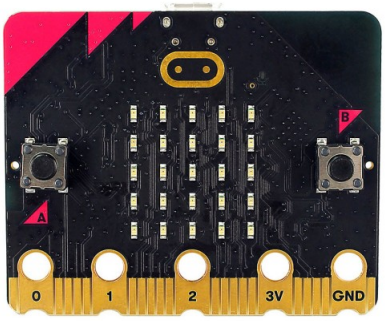

Figure 4: *BBC micro:bit* Photo by ElectroPeak

In this project, the micro:bit collects data from the connected sensors, processes the readings, and triggers outputs such as the fan or LED indicators based on predefined conditions.

2. **Sensors:** To monitor environmental conditions, we use 2 sensors:

   - **DHT11 (Temperature and Humidity Sensor):** The DHT11 is a low-cost digital sensor that provides accurate measurements of both temperature and relative humidity. It communicates with the micro:bit using a single-wire digital interface, which simplifies wiring. This sensor is essential for tracking climate-related changes in the environment and helps determine when to activate the fan.

Figure 5: *DHT11 sensor*

   - **BH1750 (Light Intensity Sensor):** The BH1750 is a digital light sensor that measures ambient light levels in lux. It uses an I2C interface for communication, allowing it to send high-resolution light data to the micro:bit efficiently.

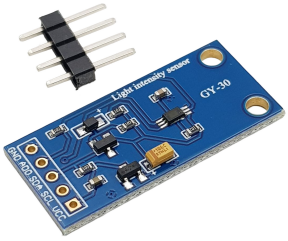

Figure 6: *BH1750 sensor*

3. **Actuators:** We use a fan to respond to the DHT11 sensor input, including:
   - **Relay Module (5V DC):** A relay acts as an electrically operated switch that allows the micro:bit to control high-current devices such as a fan. Since the micro:bit cannot directly power high-current components, the relay serves as an intermediary, enabling safe and effective operation of the fan using the low-power digital output from the micro:bit.
   - **5V DC Fan:** This cooling fan is connected to the relay and is turned on or off based on the temperature and humidity readings from the DHT11 sensor. It helps regulate environmental conditions by providing ventilation when necessary.
4. **Indicator:** We use the micro:bit's 5x5 LED as a virtual bulb, simulating the behavior of a physical light source. It's state is controlled through a remote switch on our web application.

4.2 Data communication and IoT Integration

In addition to local monitoring and control, we integrate IoT functionality to enable real-time data transmission and remote access through a custom-built **IoT gateway** and **Adafruit IO**.

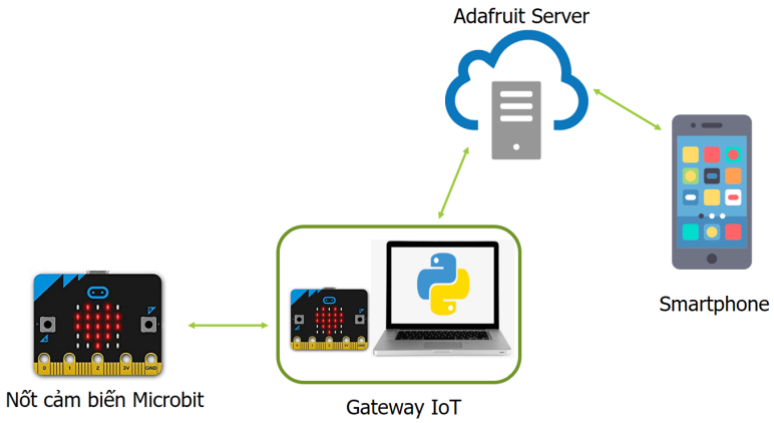

Figure 7: *Four-component architecture in IoT applications*

4.2.1 IoT gateway (Python based):

To bridge the BBC micro:bit with the internet, we developed a IoT gateway using Python. The micro:bit, which lacks native internet capabilities, communicates sensor data via a serial connection (USB in this project) to a host device (our laptop) running the Python gateway script.

The Python script performs the following tasks:

- Continuously reads data from the micro:bit via the serial port.
- Extracts sensor values (e.g., temperature, humidity, and light intensity) and formats them appropriately.
- Sends the data to Adafruit IO and receives its actions via MQTT

4.2.2 Adafruit IO

Adafruit IO is used as the central data hub for storing and visualizing the data collected from the micro:bit. It can provide dashboards to visualize live sensor data using graphs, gauges, and indicator, data storage, allow developers to set rules to trigger actions (turn on light, fan, sensors,...)

By using this integration, we are able to transform a simple sensor setup into a fully functional IoT solution capable of supporting smart applications and enabling informed decision-making processes.

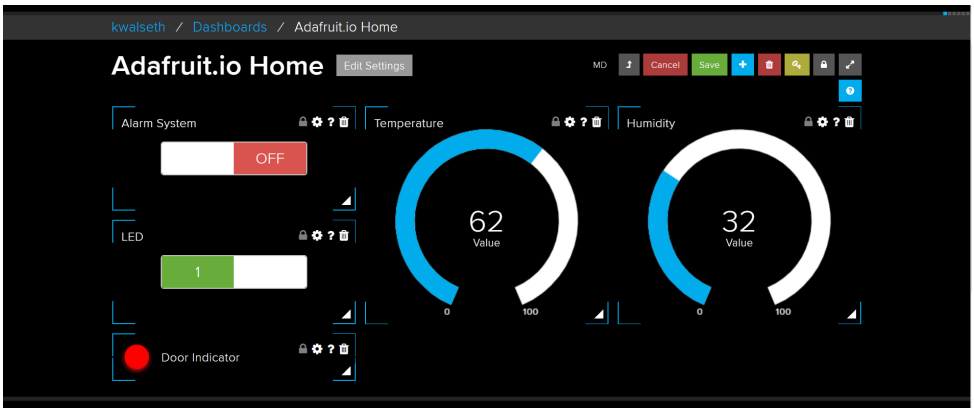

Figure 8: *Adafruit IO*

5 Frontend Implementation

5.1 Vue.js Architecture

The IntelliHome frontend is built using Vue.js 3 in combination with TypeScript, offering strong type-checking and an enhanced development experience. This architecture follows a component-based design pattern, where the user interface is divided into self-contained, reusable elements with clear responsibilities. The implementation uses Vue's Composition API, as seen in components like `HomeMain.vue` and `NavBar.vue`, which utilize `ref`, `computed`, and other reactive primitives to build interactive UIs.

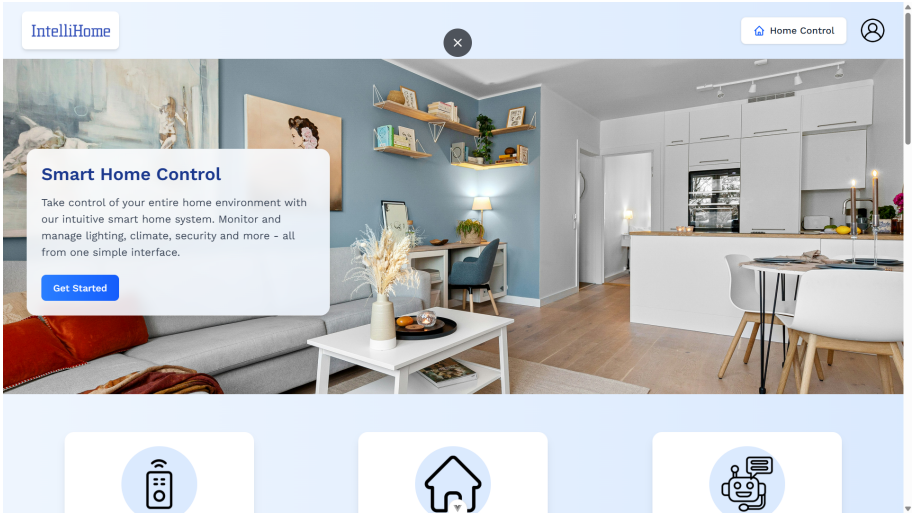

Figure 9: *Main Home Page*

The application structure organizes components hierarchically, with page-level components importing and arranging smaller, functional components. This modular approach supports independent development and testing while maintaining UI consistency. For example, the main dashboard interface in `HomePage.vue` includes specialized components such as `NavBar` for navigation and `HomeMain` for content display.

5.2 Data Reactivity and State Management

The frontend leverages Vue's reactivity system to maintain synchronized state across components without relying on external state management libraries. Environmental data such as temperature and humidity are stored in reactive references, allowing the interface to automatically update when values change. The function `updateEnvironmentMetrics` demonstrates this pattern by fetching API data and updating the state with new values.

For persistent storage, the application uses a hybrid approach. Information like user preferences and device scheduling is saved in the browser's `localStorage`, as shown in the `TimeScheduler` component where schedules are stored per room and device. This strategy ensures data is retained across sessions without requiring server-side storage.

5.3 Component Design Philosophy

The component design philosophy emphasizes separation of concerns, with each component handling a specific part of the interface. Navigation elements are encapsulated in the `NavBar` component, which manages the collapsed state and a real-time clock. The `HomeMain` component focuses on displaying environmental metrics and controlling devices, while scheduling functionality is delegated to the `TimeScheduler` component.

This separation extends to data-fetching logic, which is abstracted into dedicated service modules. The `update_environment_metrics.js` module handles API calls for sensor data, while `scheduleService.ts`

manages scheduling logic. This abstraction allows components to focus on presentation instead of business logic or data handling.

5.4 Responsive Design Implementation

The responsive design strategy employs Tailwind CSS utilities throughout component templates, adjusting layout based on screen size. The navigation sidebar is a clear example, with a collapsible design that switches from a full sidebar on large screens to a mobile-friendly overlay on smaller devices. Dynamic class bindings control this transition, applying different CSS styles based on the `isMobile` and `isMinimized` state variables.

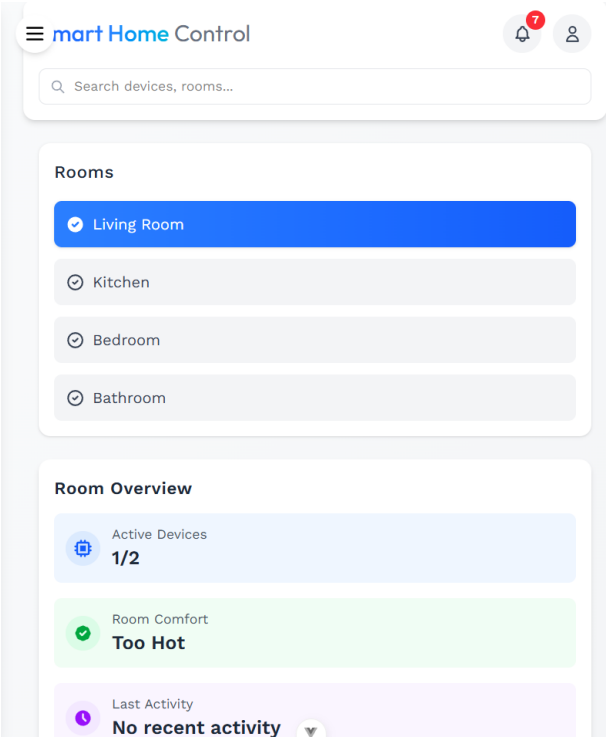

Figure 10: *Responsive Layout for Mobile*

Media queries are implemented using Tailwind's responsive prefixes, such as `max-lg:h-[30rem] h-[35.4rem]`, which adjusts element height according to screen resolution. This approach ensures a consistent UI across devices while optimizing user experience for each screen size.

5.5 User Interface Patterns

The UI design follows consistent patterns to enhance usability. Environmental parameters are presented in clear, card-based layouts, showing current values, status, and representative icons. Interactive controls maintain a cohesive visual style through shared toggle switches and buttons, providing immediate feedback on state changes.

Modal dialogs, such as those in the `TimeScheduler` component, adhere to standard design conventions with blurred backdrops, centered content, and standard header and action buttons. This consistency reduces cognitive load and improves user learnability when navigating the application.

5.6 Router Configuration

The application's navigation structure is defined in the router configuration, mapping URL paths to UI components. Dynamic imports are used to load components as needed, improving initial load performance by deferring the loading of less frequently visited views. The router supports both a welcoming landing page and the main dashboard interface, with commented routes for future expansions such as security, remote control, and settings.

5.7 API Integration

The frontend integrates with the Flask backend via API utility functions in the `update_environment_metrics.js` module. These utilities handle data fetching, error management, and data transformation for display. The implementation includes retry logic and error handling to ensure resilience during network instability or slow server responses.

Error states are captured and displayed to users via reactive properties, ensuring that data load failures do not break the interface. The system gracefully degrades when environmental data is unavailable, showing loading indicators or appropriate error messages.

5.8 Development and Build Configuration

The development environment is configured using Vite, enabling fast hot module replacement during development and efficient builds for production. TypeScript integration ensures type safety across the codebase, with custom type definitions in `global.d.ts` and `env.d.ts` supporting project-specific concepts such as device scheduling.

The build process, defined in `package.json`, includes type checks and bundling steps to produce optimized production assets. This configuration balances development convenience with deployment performance.

6 Backend Deployment

6.1 Flask API Server Architecture

The IntelliHome system employs a Flask-based API server that acts as a critical intermediary between environmental sensors and the frontend application. This architecture decouples data collection from presentation, allowing each component to evolve independently while maintaining a consistent interface through RESTful endpoints. The server enables Cross-Origin Resource Sharing (CORS) to allow secure communication from the Vue-based frontend, while preventing unauthorized access from external domains. Unlike monolithic smart home platforms that integrate both data collection and visualization, this separation enhances maintainability and reduces latency through specialized components.

The backend's error-handling strategy utilizes a comprehensive logging system that records exceptions at multiple levels—from network failures during data collection to database operations—and persistently logs them to `api_server.log`, while also displaying critical issues on the console. This dual approach supports real-time debugging during development and retrospective analysis in production environments.

6.2 Data Storage and Management

Environmental data is stored using a lightweight SQLite database architecture consisting of three dedicated tables—`temperature_readings`, `humidity_readings`, and `light_readings`—each capturing time-series measurements linked to specific rooms. This design optimizes query performance for time-series analysis while maintaining a lightweight footprint. Unlike cloud-dependent solutions, using a local database ensures continued system functionality during network outages and eliminates privacy concerns related to external data transmission.

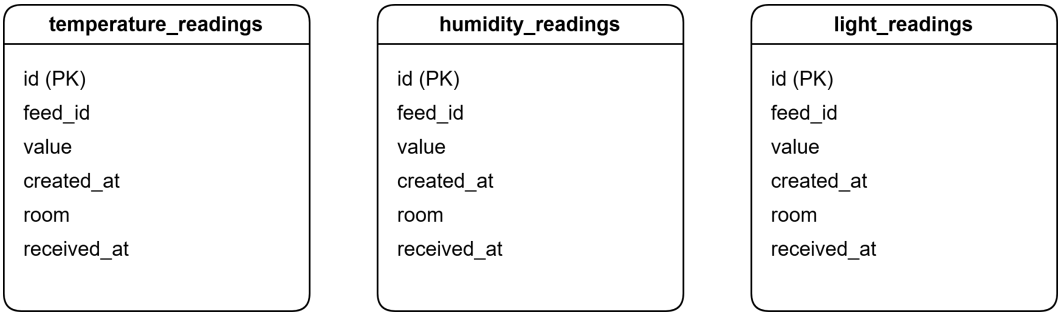

Figure 11: *Database Structure*

The database schema follows a uniform structure across measurement types, with each table containing identical fields: `id`, `feed_id`, `value`, `created_at`, `room`, and `received_at`. This standardized design simplifies code maintenance and enables consistent query patterns regardless of data type. Database connections employ a connection pooling pattern via the `get_db_connection` function, which sets row factories to return dictionary-style results, improving readability in data-processing functions.

6.3 Environmental Data Collection Pipeline

The data collection pipeline operates through a dedicated collector module that periodically retrieves environmental metrics from Adafruit IO feeds. The process uses authenticated REST requests on a configurable interval, currently set at three minutes, balancing data freshness with API usage constraints. Each collection cycle covers multiple rooms defined in the system, with parallel requests for temperature, humidity, and light data per room. The collector's robust error handling skips transient network issues without halting the entire process, ensuring resilience under unstable connections.

Incoming data undergoes immediate validation, including type checks and value verification before storage, to ensure database integrity. Each measurement is timestamped twice—once at the sensor platform's origin and once

upon local receipt—allowing latency and synchronization issues in the sensor network to be analyzed.

6.4 RESTful API Design

The API implements resource-oriented endpoints consistent with REST principles, organizing data access by room and measurement type. Key endpoints include `/api/metrics/<room>` for aggregated data, `/api/history/<room>/<metric_type>` for time-series analysis, and `/api/timerange/<room>/<metric_type>` for time-bounded queries. This structure provides intuitive access patterns while maintaining a clear separation between data types.

Each endpoint returns standardized JSON responses containing success indicators, requested data, timestamps, and error messages if applicable. This uniform format simplifies frontend integration and provides comprehensive context for error handling and data validation. The API supports query parameters for customizing data retrieval, allowing the frontend to specify time ranges, aggregation intervals, and result limits without requiring additional dedicated endpoints.

6.5 Data Aggregation and Processing

Historical data retrieval incorporates intelligent aggregation by grouping data over time intervals, reducing volume while preserving meaningful trends. For long-range queries, data is automatically grouped into configurable time buckets—typically two hours—with average values computed to represent each segment. This approach optimizes payload size and rendering performance while maintaining insight into key environmental fluctuations.

Statistical processing extends beyond aggregation to include derived metrics and trend analysis. Each historical data query returns minimum, maximum, and average values over the requested period, providing deeper context without requiring separate API calls. The `calculateBasicStats` utility performs these calculations, supporting multiple metric types and automatically parsing numeric values from temperature and humidity readings.

6.6 Error Handling and System Reliability

The backend implements a multi-layered error handling mechanism that categorizes exceptions into connection errors, data validation errors, and database errors. Each category is handled specifically to maximize system reliability—connection timeouts are automatically retried, invalid data falls back to the last known good value, and database errors trigger transaction rollbacks to ensure data integrity. A health-check endpoint at `/api/status` offers operational transparency, reporting current status and recent error counts to monitoring systems.

HTTP status codes are carefully mapped to specific error scenarios, with 400-series codes indicating client-side fixable errors (e.g., invalid parameters) and 500-series codes signaling server-side issues requiring administrator intervention. Error responses include detailed messages that aid troubleshooting without revealing sensitive system details.

6.7 System Integration Considerations

The backend maintains loose coupling with frontend components through clearly documented API contracts rather than tight integration, allowing independent development of each system. This architectural decision supports future expansion to other frontend applications or third-party integrations. The system is designed with modular patterns that separate data collection and API provisioning, enabling components to be scaled or replaced independently as needs evolve.

Security considerations are addressed through API key authentication for external data sources and planned JWT-based authentication for user-related features. While the current focus is on environmental monitoring, the architecture is prepared for future expansion into device control through additional API endpoints, converting frontend-issued commands into device-specific control protocols.

7 Data Collection and Analysis

7.1 Environmental Monitoring Architecture

The IntelliHome system implements a multi-layer environmental monitoring architecture, focusing on continuous data acquisition from physical sensors via the Adafruit IO platform. This approach introduces a logical separation between physical devices and the application layer—allowing sensor hardware to evolve independently from visualization components. A collector daemon operates as a background process with a configurable sampling interval, currently set to three minutes to balance data granularity against bandwidth and API rate limits.

Environmental data passes through three distinct processing stages: acquisition from external sources via RESTful requests to Adafruit IO, structured storage in a relational database, and consumption via an API by the frontend application. This staged approach enables data normalization techniques at each transition point, ensuring consistent measurement formats prior to display. Unlike monolithic IoT platforms that tightly couple data acquisition and visualization, this decoupled structure allows each component to be independently optimized—the collector prioritizes reliability using reconnection and connection pooling techniques, while the API layer emphasizes query performance.

7.2 Sensor Integration Framework

The system's sensor integration strategy applies a unified abstraction layer to synchronize various environmental metrics—temperature, humidity, and light—into a consistent data model, regardless of differences in units, scales, or sensor characteristics. Each metric type follows dedicated acquisition routines tailored to the sensor's specifics but exposes a standardized interface for the storage layer. This abstraction also incorporates room-based classification, treating rooms as primary entities rather than mere attributes, allowing consistent multi-room analysis across sensor types.

Feed-level authentication ensures secure access to external sensor platforms through token-based authorization, with credentials managed separately from the source code to enhance security. The collection framework supports graceful degradation when specific sensors are unavailable, maintaining partial functionality instead of triggering a system-wide failure. This feature is particularly beneficial for temperature and humidity monitoring, enabling independent operation under localized failures—reflecting an architectural choice that favors partial availability over complete system interruption.

7.3 Data Processing Pipeline

Environmental measurements are systematically transformed through a data processing pipeline, including validation, normalization, and enrichment of raw data. Input values are checked for data types and value ranges based on predefined environmental thresholds—for example, temperatures outside habitable ranges trigger warnings but are not discarded, allowing anomaly detection without losing potentially critical information.

Temporal data is enriched with two timestamps—the original recorded time and the server receive time—to support latency analysis in the collection network. The pipeline performs threshold-based state classification, converting numeric values into qualitative indicators such as ”Normal”, ”Low”, or ”High” for humidity, and ”Cold”, ”Normal”, or ”Hot” for temperature. These transformations enhance user accessibility while preserving raw data for technical analysis.

The processing architecture employs a producer-consumer model, with the collector and API server operating independently, communicating via the database layer. This design creates a natural buffer under high load conditions and allows each component to scale or restart independently without data loss. In contrast to real-time processing architectures that require immediate handling, this approach optimizes for reliability in unstable home network environments, where connection drops are common.

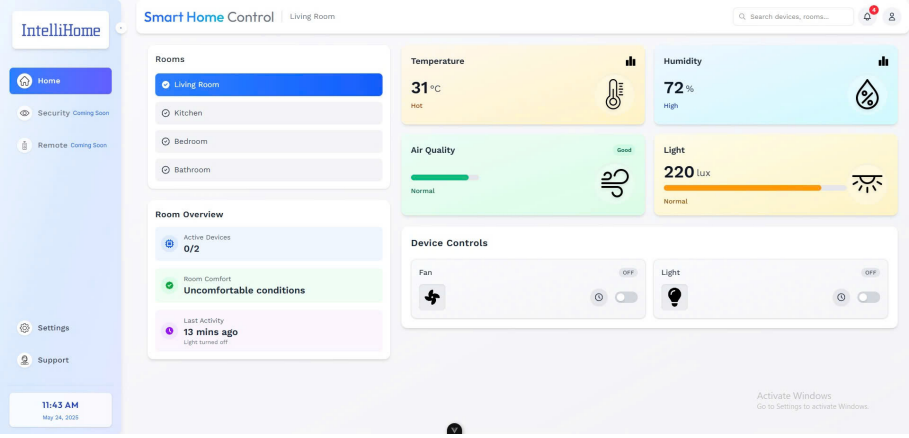

Figure 12: *Home Management*

7.4 Historical Data Management

The system implements an adaptive storage strategy for managing time-series data, retaining long-term measurements without overloading memory. Raw measurements are stored permanently at their original resolution, while the API generates aggregated charts in real time based on the query period—with hourly averages for recent data and broader intervals for older records. This approach differs from fixed-resolution systems or those that limit retention or consume excessive memory by maintaining high resolution at all times.

The query interface provides flexible parameters via RESTful API, allowing the frontend to request specific time ranges with customized resolution. The `fetchMetricHistory` and `fetchMetricBetweenDates` functions illustrate this design, enabling time limitation, interval grouping, and result limiting—balancing detail and performance based on visualization needs. This system supports both general overviews and detailed analysis without requiring specialized storage systems.

The historical data system uses statistical aggregation functions to compute meaningful temporal indicators such as minimum, maximum, and average values—transforming raw time series into actionable insights. These computations are performed at query time rather than preprocessed, ensuring that metrics always include the most recent measurements without requiring periodic refresh routines. This design reflects a balance between query efficiency and data freshness, with reasonable response times for residential monitoring scenarios.

7.5 Data Visualization Methodology

Environmental metric visualization follows a multi-resolution display strategy, presenting current conditions as prominent numeric values with supplemental status text, while supporting interactive access to historical trends through linked charts. The current state is clearly presented using color coding—cool green for optimal humidity, orange warnings for high temperature—conveying information through both numeric values and visual design.

Time-series charts employ a progressive disclosure model, showing simple trend indicators in the overview mode and enabling detailed charts on demand. The display architecture supports asynchronous historical data loading, maintaining a responsive interface regardless of the viewed time range. This approach conserves network resources by only loading historical data when requested by the user, instead of fetching it all during interface startup.

Status determination is based on context-aware thresholds rather than absolute values, recognizing that optimal environmental conditions vary by room and season. The system's classification translates raw values into meaningful state information, helping users understand the data without needing deep environmental science knowledge. This approach reflects the system's overarching philosophy: presenting technical data in a user-friendly manner while retaining analytical depth for expert use.

Figure 13: *Graph Data Visual*

7.6 Multimetric Analysis Capabilities

Beyond analyzing individual metrics, the system architecture supports correlation analysis between environmental factors, enabling the discovery of relationships among temperature, humidity, and light. The unified endpoint at `/api/metrics/<room>` provides synchronized access to all environmental metrics in a given space, supporting aggregate temporal analysis. This design allows the detection of environmental change patterns that might be missed when examining metrics in isolation—such as the inverse relationship between rising temperature and falling humidity when heating systems are active.

Statistical processing extends to correlation analysis, computing not only individual averages but also correlation coefficients between related metrics. Time synchronization across measurements, achieved through consistent timestamp processing, ensures analytical validity without requiring complex synchronization logic. Unlike monitoring systems that treat each factor in isolation, this integrated approach acknowledges the interconnectedness of living environments, where changes in one aspect often affect others.

The system's analytical architecture is designed for future extensibility, with a modular structure that accommodates new sensor types through the same acquisition, storage, and display processes. This approach ensures the monitoring system can evolve alongside smart home technologies without requiring a complete structural overhaul. The combination of flexible data collection, adaptive storage, and multi-level visualization results in an integrated environmental monitoring system that balances technical complexity with user accessibility—suitable for practical deployment in household settings.

8 Core Features

8.1 Environmental Monitoring System

The IntelliHome system implements a comprehensive environmental monitoring architecture based on three key metrics—temperature, humidity, and ambient light intensity. Rather than treating these measurements as isolated data points, the system conceptualizes them as interrelated components of environmental quality, crucial for assessing the livability of a space. Data collection occurs through a continuous acquisition cycle managed by `data_collector.py`, which interfaces with external IoT platforms via authenticated APIs. This process maintains a persistent connection to Adafruit IO feeds, retrieving real-time sensor data at configurable intervals while adhering to API rate limits.

The monitoring architecture adopts a room-contextual model, recognizing differences in environmental requirements across household spaces. This approach assigns semantic meaning to numeric values based on spatial context—acknowledging, for instance, that ideal humidity levels for a bathroom differ markedly from those of a living room, and that temperature thresholds vary between kitchens and bedrooms. This contextual awareness extends to the presentation layer, where color coding provides visual cues of environmental status: cool green for optimal humidity, orange for high temperatures, and neutral tones for standard conditions.

The system goes beyond mere threshold monitoring by incorporating comparative analytics to detect significant deviations from established patterns. This approach distinguishes between expected environmental fluctuations—such as midday temperature rises—and potentially problematic changes requiring user attention. Unlike binary alert systems that only flag threshold violations, this nuanced strategy reduces alert fatigue by recognizing that contextual environments dictate appropriate intervention thresholds. The monitoring framework also enhances understanding by timestamping data, enabling users to correlate environmental shifts with household activities or external conditions.

8.2 Device Control System

The device control architecture employs a reactive state management model, maintaining real-time synchronization between the user interface and device states through unidirectional data flow. This approach, exemplified in `HomeMain.vue`, establishes a clear relationship between user actions and system feedback, ensuring interface elements accurately reflect the true state of devices—regardless of control source, whether it be direct user interaction, scheduled automation, or environmental response. Devices are organized hierarchically by room and function, forming natural groupings that mirror common perceptions of domestic spatial organization rather than technical categorization.

Interaction with devices is facilitated through intuitive binary toggles, combining clear state imagery with refined design features—contrasting colors provide instant feedback while consistent positioning supports spatial memory. This interaction model intentionally reduces cognitive load through simple controls while supporting complex automation underneath. The control architecture enforces an access prioritization model, allowing devices to respond to multiple control sources with established precedence—manual user commands temporarily override scheduled activities without disrupting long-term automation.

The system maintains continuity of distributed device states through both server-side database storage and client-side local persistence, ensuring operational continuity even during connection interruptions. This distributed persistence approach enables offline functionality while maintaining eventual consistency upon reconnection. Control reliability is further enhanced by transactional state changes—either all aspects of a device state update successfully or none do, preserving consistency. This atomic model is especially useful for complex devices with multiple control parameters, ensuring systems remain in a predictable and complete state.

8.3 Automated Scheduling

The scheduling system implements a calendar-based temporal control framework with complex repeating patterns that align with household routines. This architecture, defined in `TimeScheduler.vue` and `scheduleService.ts`, uses a weekday-organized system with independent triggers for each day—acknowledging that household routines often follow weekly rather than daily cycles. Schedule configuration occurs via a modal interface with full-context time controls, enabling users to visualize full weekly patterns rather than isolated time fragments.

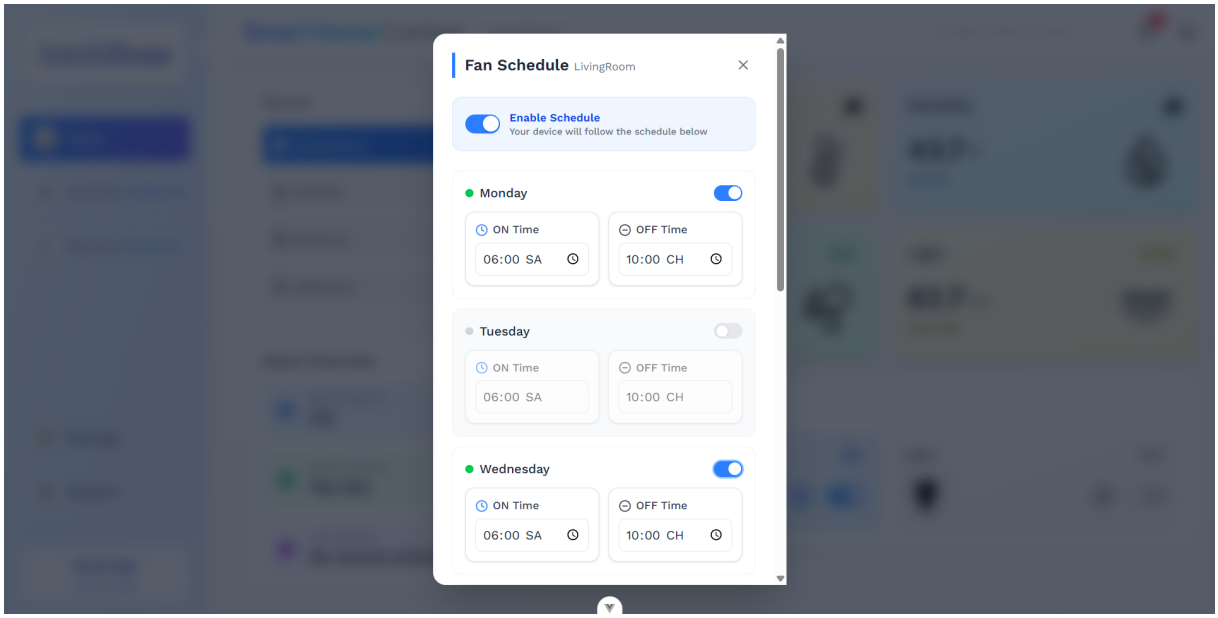

Figure 14: *Time Schedule*

Schedule persistence employs a hybrid storage strategy, maintaining schedules in local browser memory for immediate access while synchronizing with server-side components to ensure cross-device consistency. This design prioritizes fast, responsive interface interactions while accommodating the distributed access models typical in home environments. The scheduling engine evaluates in per-minute cycles, determining appropriate device states by comparing the current time to all active rules using efficient time-comparison algorithms that minimize system resource usage despite continuous operation.

The architecture handles rule conflicts gracefully through a prioritization system, yielding consistent outcomes when multiple rules apply simultaneously. It establishes clear precedence relationships—device-specific schedules override room defaults, while manual commands temporarily suspend automation without disrupting long-term scheduling patterns. Unlike simple timers that operate independently, the scheduling system integrates with the environmental monitoring architecture, enabling context-aware actions that can dynamically adjust timing patterns based on environmental conditions or presence states—though this capability remains partially implemented in the current version.

8.4 Data Visualization and Trends

The visualization system adopts a multi-resolution approach in presenting environmental data, deploying different visual models optimized for varying analytical contexts. Current conditions are displayed through prominent numeric indicators with supplementary status cues, enabling immediate comprehension without requiring technical interpretation. Historical data visualization employs time-series charts with adaptive resolution, automatically scaling according to the time range—providing detailed patterns for recent intervals while aggregating older data to reveal long-term trends without overwhelming the display.

The `fetchMetricHistory` function in `update_environment_metrics.js` demonstrates this adaptive method through configurable time windows and interval parameters that enable progressive data revelation—from

overviews to detailed analysis of specific periods. The visualization architecture implements a consistent visual language across environmental metrics, using parallel presentation structures for temperature, humidity, and light intensity to support comparative analysis while respecting each metric's unique characteristics.

Statistical aggregations supplement raw data visualization with auto-computed values such as minimum, maximum, and average, providing analytical context without requiring manual calculation. These derived metrics form interpretive frameworks, transforming raw measurements into actionable insights and highlighting noteworthy patterns that might be obscured in simple time plots. The system also enhances analysis via time filters implemented in the `fetchMetricBetweenDates` function, allowing focused examination of periods corresponding to household activities or external conditions.

In contrast to purely aesthetic data displays, the visualization architecture emphasizes functional clarity through consistent color schemes that retain semantic meaning across interface contexts—using color as an informational dimension rather than a decorative element. This extends to state indicators that translate numeric values into contextual assessments such as "Normal," "High," or "Cold" based on threshold evaluations tailored to each metric type and room context. This translation layer bridges technical readings with human understanding, enabling effective environmental management without requiring expert knowledge.

8.5 User Customization

The customization architecture employs a multi-layered approach to adapt system behavior to specific household characteristics while preserving consistency in core interactions. Room-based organization forms the primary customization framework, enabling unique device collections and environmental thresholds per space without redundant configuration templates. This approach, reflected throughout the interface structure, acknowledges that homes comprise distinct functional areas with unique operational requirements rather than homogeneous environments.

Interface customization occurs through responsive layouts that adapt to both device capabilities and user preferences, implementing responsive design patterns that preserve functional consistency across varying display environments. The sidebar navigation component exemplifies this approach with a collapsible structure that transitions between expanded and minimized states based on available screen space or explicit user preference, retaining navigational capability while adapting to context. Unlike rigid interface variants that present wholly different structures across devices, this approach preserves spatial consistency—maintaining muscle memory and reducing cognitive load during cross-device transitions.

The system architecture anticipates notification preferences through configurable thresholds that define alert conditions based on individual sensitivity rather than universal standards. This approach acknowledges that comfort preferences vary significantly between households—and even among individuals within a single home. Customization extends to temporal patterns via the scheduling system, adapting to established household routines instead of imposing fixed operational models. This user-centered design philosophy prioritizes system adaptation to existing behaviors rather than requiring users to conform to system constraints—distinguishing IntelliHome from less flexible smart home implementations.

While current customization primarily centers on room-based organization and device scheduling, the architecture has established extensibility points for advanced personalization through modular design and preference management. Future versions may incorporate explicit user profiles with individual environmental preferences, behavior-learned pattern recognition, or context-aware adaptations responsive to presence patterns without explicit configuration. The foundation for these enhancements already exists in the current architecture through the separation of user preferences from core functionality and the use of consistent state management practices.

9 User Experience Design

9.1 Design Philosophy

The IntelliHome project follows a user experience philosophy based on simplicity and accessibility—designing interfaces that require minimal cognitive effort while still providing comprehensive monitoring and control capabilities. This philosophy is reflected in the system's approach to grouping functions by context, where features appear within natural conceptual boundaries rather than technical categories. The interface architecture is organized by room, reflecting the user's mental model of physical space, allowing environmental data and device controls to be displayed within the same spatial context. This integration helps reduce the abstraction gap between digital controls and real space, supporting users in maintaining spatial awareness while interacting with the interface.

The design language prioritizes cognitive efficiency through progressive disclosure, displaying complexity only when necessary. This approach is evident in the `TimeScheduler` component, which offers complex time automation capabilities without overwhelming the main interface. The system maintains a simple main interface with quick toggle controls, while complex parameters are placed in separate dialogs. Unlike designs that simplify by removing features, IntelliHome achieves simplicity while remaining feature-rich thanks to a rational information architecture.

The system also applies a deliberate visual consistency strategy, keeping interface states uniform across contexts. For example, a blue gradient background is maintained for the navigation bar, binary toggle designs are consistent, and icons are unified between desktop and mobile views. Instead of designing discrete states, the system creates a "visual grammar" where interactive elements have consistent behaviors and appearances, enabling users to learn once and apply across many contexts.

9.2 Interface Flow

The interface flow is designed with a spatial hierarchical navigation model, organizing the user journey through progressively specific levels. The interface starts with a welcome page introducing the system and key features, then transitions to the main interface. The navigation architecture includes two types: a fixed sidebar for functional navigation, and a horizontal room selector for navigation within the same functional category. This dual approach supports both vertical movement across main functions and horizontal movement within categories.

The routing structure reflects frequency of use considerations: frequently used features like the main monitoring screen are placed at shallow navigation depth, while features like settings reside deeper. The `HomeMain` component implements responsive horizontal navigation, allowing users to view available rooms while focusing on the current room—helping maintain global spatial awareness rather than focusing narrowly on the selected item.

The system also clearly distinguishes between navigation actions and operational actions through visual and behavioral cues. Navigation elements like the sidebar are designed to stand out and remain fixed, while control buttons such as device toggles have a subtler style. The device scheduling dialog smartly manages flow: it clearly indicates the device and room being configured, while focusing on the current task yet maintaining linkage to the overall context.

9.3 Accessibility Considerations

The accessibility strategy goes beyond minimum standards toward broad usability principles, serving all users regardless of specific needs. The interface uses semantic markup appropriately: labels are linked to controls, headings organize content hierarchically, and interactive elements have appropriate ARIA roles—ensuring compatibility with assistive technologies and enhancing overall usability. The color system maintains at least a 4.5:1 contrast ratio between text and background, ensuring readability and proper visual hierarchy.

The interface supports keyboard navigation with logical tab order and focus management for interactive elements. All primary controls are focusable and clearly display focus states, while navigation sections maintain a

consistent focus order matching visual layout. Important information is conveyed using both color and text: device states are shown with color and clear labels, environmental parameters with numeric values and descriptive statuses—supporting users with diverse preferences or limitations.

Text components maintain comfortable readability with a minimum size of 16px and appropriate spacing, while supporting scalable ratios adapting to device sizes via a flexible scale system rather than fixed units. Unlike systems treating accessibility as an add-on feature, IntelliHome integrates it into core design, creating an experience responsive to diverse users in a household context.

9.4 Mobile Device Compatibility

The responsive design architecture implements a continuous adaptation model rather than fixed breakpoints, allowing smooth transitions across device sizes through systematic layout transformations. This approach uses a combination of container queries and viewport queries, supporting component-level responsiveness. This is demonstrated by the navigation bar automatically collapsing on small screens and environmental metric cards switching from grid to vertical layouts based on display space.

The interface is optimized for touch with control elements sized at least 44×44 pixels, ensuring accurate operation on touch screens. Spacing between elements is increased to prevent accidental taps. The system also supports touch gestures such as swipe navigation alongside regular tap actions, aligning with mobile device interaction patterns.

Component structure prioritizes flexible display: search functionality is emphasized on large screens but appropriately collapsed on mobile; time and date displays adjust information density to suit screen space while retaining functionality. This approach not only adjusts layout but also adapts to user behavior depending on device type, ensuring the interface always serves effectively regardless of access medium.

10 Technical Challenges and Future Directions

10.1 Technical Challenges and Solutions

The development of IntelliHome faced several significant technical challenges that required careful consideration and innovative approaches. Chief among these was maintaining consistent data flow between the system's distributed components—physical sensors communicating through Adafruit IO, a Python-based data collection service, a SQLite database, and a frontend developed with Vue.js. Early versions experienced interruptions in sensor data transmission, leading to gaps in the environmental data history. This prompted the implementation of robust error handling and connection management within the `fetchRoomMetrics` function, including timeout limits and automatic retry mechanisms.

Database schema design also posed a complex challenge, particularly in representing time-series data in a normalized format. The solution was based on time normalization and interval-based aggregation strategies, implemented in the `get_metric_history` function in `data_collector.py`. This function groups measurements into configurable time intervals (default: 2 hours), ensuring consistent data representation despite varying sensor data transmission frequencies.

The frontend architecture had to balance real-time responsiveness with limitations in network and computational resources. Environmental metrics can change frequently, requiring efficient rendering strategies to avoid excessive DOM manipulation. This challenge was addressed through component-based design in Vue.js, particularly in `HomeMain.vue`, where environmental metric cards update individually without triggering a global re-render. Similarly, the scheduling functionality in `TimeScheduler.vue` required sophisticated state management to coordinate time settings across multiple devices and rooms while ensuring a smooth user experience.

Currently, the system uses a client-side polling model to update environmental data every few minutes, striking a balance between data freshness and system overhead. For real-world deployment, this model could be optimized using WebSocket or server-sent events to reduce unnecessary API calls while maintaining data timeliness. The current authentication system relies on manual Adafruit IO credential entry, which should be upgraded to a more secure model using safe credential storage mechanisms in production environments.

Resource usage analysis revealed potential for optimization in data retrieval and processing. The current `fetchMetricHistory` function loads the entire result set into memory before processing, which is suitable for expected data volumes but may become problematic in long-term deployments with years of historical data. Implementing server-side pagination or streaming data processing would improve scalability for long-term deployments.

10.2 Expansion Directions

The system architecture is intentionally designed for extensibility, clearly reflected in the modular nature of its components and router configuration with placeholders for modules such as security, remote access, and user profile management. The environmental monitoring system forms a natural foundation for evolving into intelligent automation—shifting from passive data collection to proactive control based on environmental conditions. This would transition the scheduling function from time-based to context-aware operation.

Data visualization capabilities also present a rich direction for expansion. The current version primarily displays environmental metrics as raw figures, but the data collection infrastructure already supports deeper analytics. Integration with specialized visualization libraries could enable pattern recognition across environmental indicators, turning raw data into actionable insights regarding building performance or user comfort levels.

Integration with the broader smart home ecosystem is another promising development path. The backend's API-first design with Flask is well-suited for integration with voice assistants, standard automation protocols like Matter or Zigbee, and even machine learning models for predictive environmental control. The modular design of both the frontend and backend facilitates incremental expansion without the need for a complete system redesign.

User personalization is also a highly promising development direction. The current system operates at the household level but could be expanded to support individual preferences, custom dashboards, and personalized

alert thresholds. This would transform IntelliHome from a generic home management tool into a system that adapts to the unique preferences of each family member, enhancing both utility and user satisfaction.

11 Conclusion

The IntelliHome project represents a multidisciplinary effort to develop an integrated smart home system, combining modern software technologies with sensor hardware engineering. By implementing a distributed system architecture—including a Vue.js user interface, a Flask API server, and a data collection system from the Adafruit IO platform—the project successfully built a unified platform for monitoring and controlling the living environment. The system enables users to track key environmental metrics and automate home device management while maintaining an intuitive user interface with responsive feedback.

Key challenges related to synchronizing data from multiple distributed sensor sources were addressed through optimized database design and standardized data collection processes. The `data_collector.py` system connects to the Adafruit IO API, storing raw data in an SQLite database, while the Flask server provides API endpoints for the user interface to access processed data. This strategy ensures data integrity even with unstable internet connections, a common issue in typical IoT systems.

The Vue.js structure employs a component model that clearly separates different functional parts of the user interface, from navigation and dashboard display to device scheduling management. This architecture facilitates parallel development and modular improvements, while ensuring a consistent user experience across multiple devices. The device scheduling system—one of the standout features—skillfully leverages local storage and time processing capabilities to automate device operation without requiring a continuous connection to the central server.

The project's integrative approach is also noteworthy for combining different yet compatible technologies into a unified system. Flask's RESTful API creates a clear abstraction layer between data collection and data presentation, allowing independent development of system components. This proves especially useful when adding new sensor types without major UI changes, while UI enhancements do not impact core data processing logic.

The system's data visualization, represented by `environmentMetrics` and `roomDevices`, provides visual insights into the current state of the living environment. Grouping algorithms and historical data analysis (`calculateBasicStats` and `get_metric_history`) generate valuable insights on trends and patterns, transforming raw sensor data into actionable information for end users.

Finally, IntelliHome's open architecture lays the foundation for future enhancements. Components commented within the router configuration (including security management, user profiles, and remote control) demonstrate a clear vision for a broader smart home ecosystem to be developed over time. The combination of solid software engineering and user-centered design philosophy forms the core value of IntelliHome—a system that not only monitors the living environment but also empowers users to enhance their quality of life through advanced yet accessible technology.