LAB University of Applied Sciences

Real-Time IoT Platform for Smart Environmental Monitoring

A Case Study of Rakkaranta Resort

LAB University of Applied Sciences
Bachelor of Engineering, Industrial Information & Technology
2025
Ye Thu Hlaing

Abstract

Author(s)	Publication type	Completion year
Hlaing Ye Thu	Thesis, UAS	2025
	Number of pages	
	49	

Title of the thesis

Real-Time IoT Platform for Smart Environmental Monitoring

A Case Study of Rakkaranta Resort

Degree

Bachelor of Industrial Information & Technology

Name, title and organisation of the client

Abstract

This thesis presented the design architecture and implementation of an Internet of Things (IoT) analytics dashboard for Rakkaranta Resort, which monitors environmental metrics and enhances the overall customer experience. The project integrated a production-level DevSecOps pipeline for the iterative, secure, and scalable software development and deployment life cycle.

An integrated methodological approach was adopted, combining system architecture, software design patterns, and prototyping. The system was developed on Azure cloud infrastructure integrated with IoT sensor networks to collect real-time environmental data, such as temperature, humidity, and water level. Data acquisition was managed via the MQTT messaging protocol and Telegraf agent, with time-series data stored in the InfluxDB database. The backend was developed using a Node.js architecture, while the frontend leverages the Next.js framework for server-side rendering.

The DevSecOps pipeline was composed of vulnerability scanners and SonarCloud code quality checking, ensuring security compliance and maintainable code. The pipeline was built using Docker, GitHub Actions, and Azure Container Registry. This custom IoT application tailored to the Finnish Resort scenario can significantly facilitate both development and operational workflows. Future recommendations include the physical integration of sensor hardware and role-based authentication between guests and staff.

Keywords

Azure, Internet of Things, Docker, Containerization, MQTT, InfluxDB, Next.js, Supabase, Kinde, DevSecOps

Contents

Appendices

Appendix 1. Supplementary Materials

Appendix 2. Landing Page

Appendix 3. Authentication Pages

Appendix 4. Main Dashboard Page

Appendix 5. Analytic Dashboard Page

LIST OF ABBREVIATIONS (OR) SYMBOLS

DAST	Dynamic Application Security Test
DevOps	Development and Operations
DevSecOps	Development, Security, and Operations
ETL	Extract, Transform, Load
IoT	Internet of Things
ISR	Incremental Static Regeneration
MQTT	Message Queue Telemetry Transport
MQTT	Message Queuing Telemetry Transport
NLP	Natural Language Processing
ORM	Object–relational Mapping
PaaS	Platform-as-a-Service
SaaS	Software-as-a-Service
SAST	Static Application Security Test
SCA	Software Composition Analysis
SDLC	Software Development Life Cycle
SSR	Server-side Rendering
HTTP	Hypertext Transfer Protocol

1 Introduction

1.1 Background

The wave of IoT innovation has opened several pathways for enhancing the operational efficiency across industrial and community sectors. In particular, the tourism industry undergoes significant changes in how to fulfill the guest experience through innovated IoT systems. While the tourism business using IoT systems is increasingly acknowledged, limited attention has been given to the practical application in resort contexts – especially data-driven decision-making and optimizing resource management.

According to Astanakulov et al. (2025, pp. 153–154), the hotel chains have adopted the IoT-enabled energy management systems integrated with temperature, occupancy, and lighting sensors to autonomously regulate energy consumption. These smart IoT platforms have not only enhanced the operational efficiency but also reduces by up to 20% energy usage and 15% carbon emissions. Such outcomes underscore the sustainability potential of IoT adoption in the tourism industry offering a similar case in resort operations.

The Rakkaranta wilderness resort project addresses this research gap by implementing a comprehensive IoT solution specifically designed for Insight-led decision-making that are applicable to the broader tourism industry.

1.2 Objectives

The primary objective of this practice-based thesis is to implement a real-time IoT system specifically for Rakkaranta Resort in Finland. This tailored IoT architecture also supports resort's smart decision-making whilst maintaining reliable data transmission across the resort property. It is designed to establish a robust data pipeline that ingests, processes, and stores environmental metrics, transforming these raw data inputs into tangible takeaways.

The key outcome is a comprehensive analytic dashboard that allows the resort staff to monitor real time events and make well-informed decisions without needing a technical background. The dashboard is the primary user interface for all resort management operations, consolidating multiple data streams into a cohesive visualization platform.

Furthermore, the system consists of unique tools for predicting the northern lights, tracking real-time weather insights and recommending activities based on the weather. The inventory service is created for the staff to seamlessly manage resort assets, such as accommo-

dations, amenities, and facilities, ensuring proper maintenance and availability. The customer support system is implemented to resolve complains and capture feedback to create a continuous improvement loop for the resort.

1.3 Delimitations

Although the system must interface with physical IoT sensors, the current implementation employs a dummy server generating simulated real-time data for actual embedded components. This approach simplifies the process of testing the data pipeline components without tiresome hardware configuration. Additionally, the interface combines both administrative and user functionalities within a single unified dashboard rather than a role-based implementation. The thesis excludes the in-depth examination on the impact on the tourism industry. It is tailored to fit the unique requirements and challenges of the Rakkaranta Resort rather than enterprise-level deployment. For features such as inventory management and customer support, the data are stored persistently in the database since those features are not fundamental to the thesis objectives.

2 Technology Stack

2.1 MQTT Server

The MQTT broker, or MOM (message-oriented middleware), acts as the central hub to connect several different systems together so the software applications can share credentials with each other. It handles the messages by validating, translating, filtering, routing, and transforming between messaging protocols. This enables cross system communication regardless of the programming languages in which they are built in (Oliveira, 2021, p.10).

For the resort scenario where the communication is challenged by the remote location or harsh weather conditions, MQTT is favoured due to its light-weight design and well-suited for low-bandwidth utilization in constrained environments. Moreover, the publish-subscribe architecture allows one-to-many transmission that prevents direct coupling between subscribers and publishers. The client-side transparency is the key feature in which the data flow is hidden from the clients, meaning they are unaware of whether the data is local or global. The use of HTTP WebSockets allows interoperability that facilitates communication with many existing webservices (Azzedin and Alhazmi, 2023, p. 1- 2).

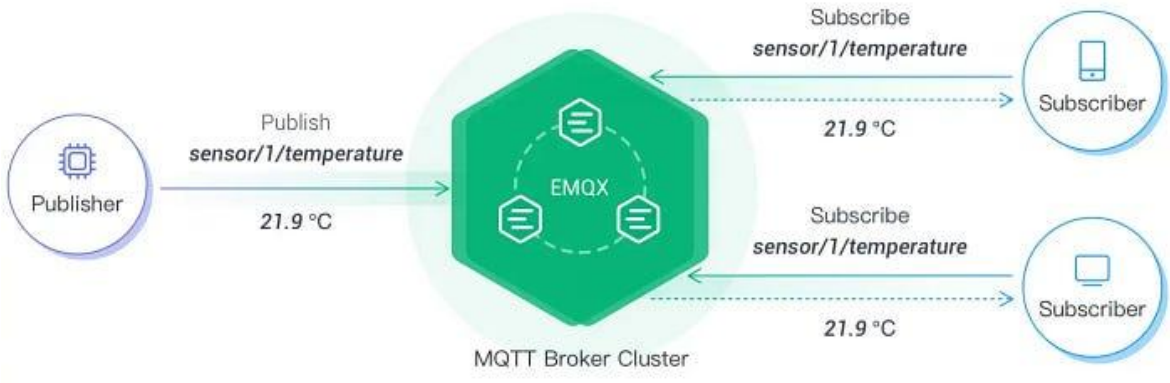


Figure 1. MQTT Broker Architecture (EMQ Technologies, 2025).

AMQP and Apache Kafka are alternatives message brokers in distributed systems. However, the resource overloading makes it less efficient for constrained environments. Kafka allows high volume throughput but introduces unnecessary complexity for the resort's scale.

2.2 Telegraf as Data Collection Framework

The ETL (Extract, Transform, Load) paradigm is the foundational approach for modern IoT data collection frameworks. Regarding the Rakkaranta Resort case, the ETL pipeline is

implemented by unifying the data from different sources into a cohesive format – temperature, humidity and other sensor readings are seamlessly combined into a single dashboard. Moreover, the data accuracy is maintained through validation and standardization.

Telegraf service is utilized as the primary ETL agent to retrieve accurate readings, optimize resource usage on limited system, and perform edge preprocessing such as filtering, aggregation and transformation at the collection stage to reduce network bandwidth usage. Telegraf, with its rich resources, provides extensive plugin ecosystem tailored to various sensors and data sources. Its low memory footprint makes Telegraf compatible with edge devices with limited resources. Although Logstash and Fluentd offer similar functionalities, Telegraf outperforms them due to its tight integration with InfluxDB and minimal resource requirements.

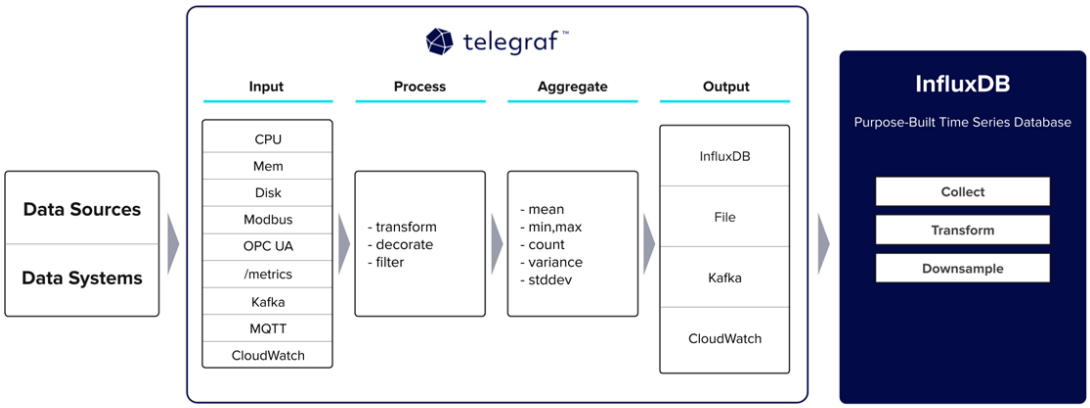


Figure 2. Pull-based Architecture of Telegraf (Clifford, 2021).

2.3 InfluxDB for Time-Series Data Storage

The time-stamped data are stored in dedicated database optimized for handling the high volume of time-series data collected at regular intervals. The time-series database is favoured over traditional databases because it is highly scalable across millions of IoT devices in a continuous data flow. The database architecture is practically designed to be resilient to any failures even in the network segmentation or hardware downtime. The retention policy is applicable to automatically manage the data lifespan.

One major benefit is the ability to retrieve fast and efficient range queries by grouping the correlated data and prevent memory overloads which are common in traditional database. The time-series databases are designed for high write performance without compromising the speed and availability under heavy loads (Naqvi, Yfantidou and Zimányi, 2017, p.6).

Among time-series databases – TimescaleDB, Prometheus, and QuestDB, InfluxDB is selected because it is lightweight and compatible with TICK stack IoT ecosystems (Telegraf, InfluxDB, Chronograf, Kapacitor). It is an open source schemeless database written in Go programming language developed by InfluxData. The closed-source versions provide extra functionality for high availability, scalability and ability to run either on premises or in the cloud on a single node (Naqvi, Yfantidou and Zimányi, 2017, p.8).

2.4 Supabase for Relational Data Management

No-code and serverless relational databases such as Firebase or AWS Amplify encounter scalability issues and complex queries in NoSQL environments. In contrast, modern alternatives like Supabase and PlanetScale adopt relational models, positioning them well-suited for data-intensive applications.

The comparison of both Supabase and Firebase have different data models. Firebase's Firestore offers a schema-less NoSQL database which is why developing complex systems with advanced transactions or JOINs become tedious. Firebase raises the vendor lock-in concern as its counterparts particularly for larger organizations. Conversely, Supabase's codebase is completely open source, making migrating or integrating with other services much easier if the need arises.

Compared to AWS Amplify, it is evident that Amplify facilitates the seamless integration of multiple Amazon Services to build complex systems, but its steep learning curve and potential for overengineering are the future concern. Supabase, on the other hand, is straightforward: all backend services are easily navigable through a unified user interface enabling a drastic reduction in workload (Endo, 2025).

2.5 Kinde for Authentication

Kinde has been selected over alternatives such as Auth0 and AWS Cognito due to its developer-friendly implementation and transparent pricing model. Although Auth0 and Cognito are robust and feature-rich authentication platforms, their steeper learning curves and complex configurations lead to slower-paced implementation. Since the Rakkaranta Resort scenario has too small a scope to build as the enterprise-level orientation as Auth0 provides. In contrast, Kinde offers a modern, developer-friendly interface with concise documentation, especially appealing for the startups and small- to medium-sized applications.

2.6 Node.js for Backend Services

Node.js was selected as the backend platform for the Rakkaranta IoT architecture mainly due to its capability for real-time. Multiple concurrent sensor connections can be easily handled by its asynchronous I/O model without blocking operations. The JavaScript ecosystem makes frontend and backend unified, reducing the code complexity and development time. The primary reason Node.js is suitable especially for building IoT backend development is because of its event loop architecture featuring non-blocking I/O requests. This means the thread is not overloaded, making the event loop continuous even when multiple asynchronous API calls (including input/output operations) are made (Nkenyereye and Jang, 2016, p. 14).

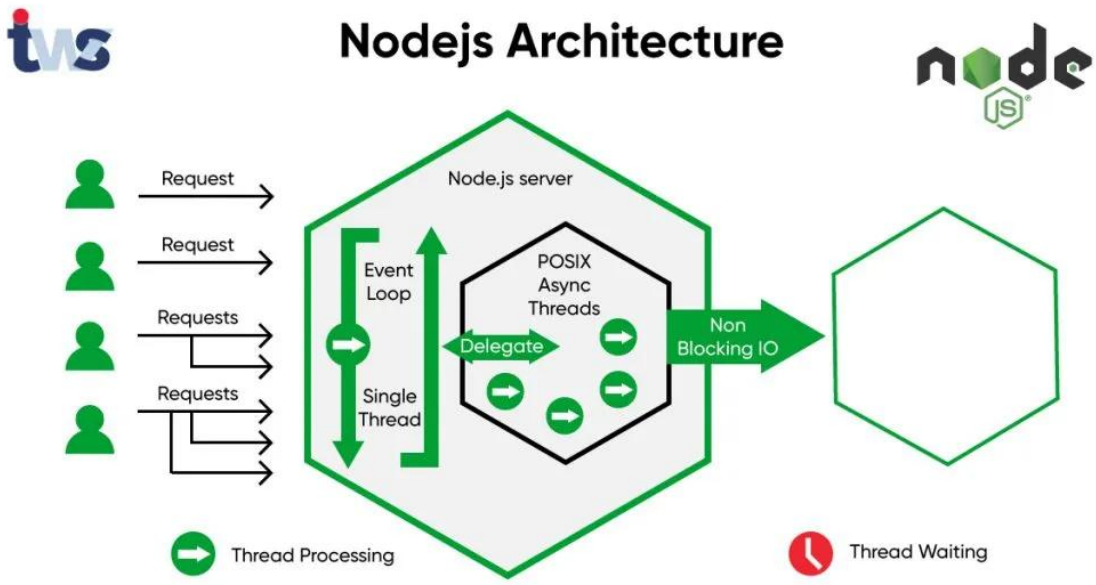


Figure 3. In-depth Node.js Architecture (Sood, 2024).

The alternative popular open-source JavaScript runtimes are Deno and Bun. The former uses the V8 engine like Node.js, while the latter uses WebKit's JavaScriptCore engine. Both runtimes provide drop-in replacement for Node.js but Node.js offers unmatched ecosystem maturity with millions of packages through npm, the world's biggest package registry. Moreover, Node.js is actively integrating the unique features that Deno and Bun offers—including experimental TypeScript support, built-in SQLite, and a permissions model. The vast contribution from developer community and the backing of the OpenJS foundation guarantee the long-term sustainability, stability, reliability and constant upgrades over a decade of production use (Ulili, 2024).

Figure 4. Comparison between popular JavaScript runtime (Ulili, 2024).

2.7 Next.js/React for Frontend

The decision to employ Next.js, React framework, was due to its server-side rendering, which is critical to the initial load performance of data-intensive dashboards. SSR makes pages fully pre-rendered on the server and sends the these rendered HTML to the client browser, enhancing the User experience. Complex UI can easily be implemented in a re-usable and modular manner without incurring concerns regarding maintainability, consistency, and scalability. The expanding React community provides numerous third-party UI libraries, accelerating product development without the necessity of developing from scratch. Next.js supports Progressive Web Applications (PWAs) which allows offline operation for remote service workers without a stable network connection.

By integrating optimization techniques, Next.js enhances the SEO ranking, page loading speed, and asset delivery with reduced network bandwidth. Code splitting and lazy loading can significantly reduce the initial JavaScript bundle size for smoother performance. This results in increased organic traffic and, consequently, potential revenue as the audience is engaged through effortless navigation and discoverable websites. The ISR strategy utilized its advanced architecture to balance data staleness and freshness. First, it pre-renders HTML files at build time and subsequently updates them in the background at certain intervals (Patel, 2023, p. 25).

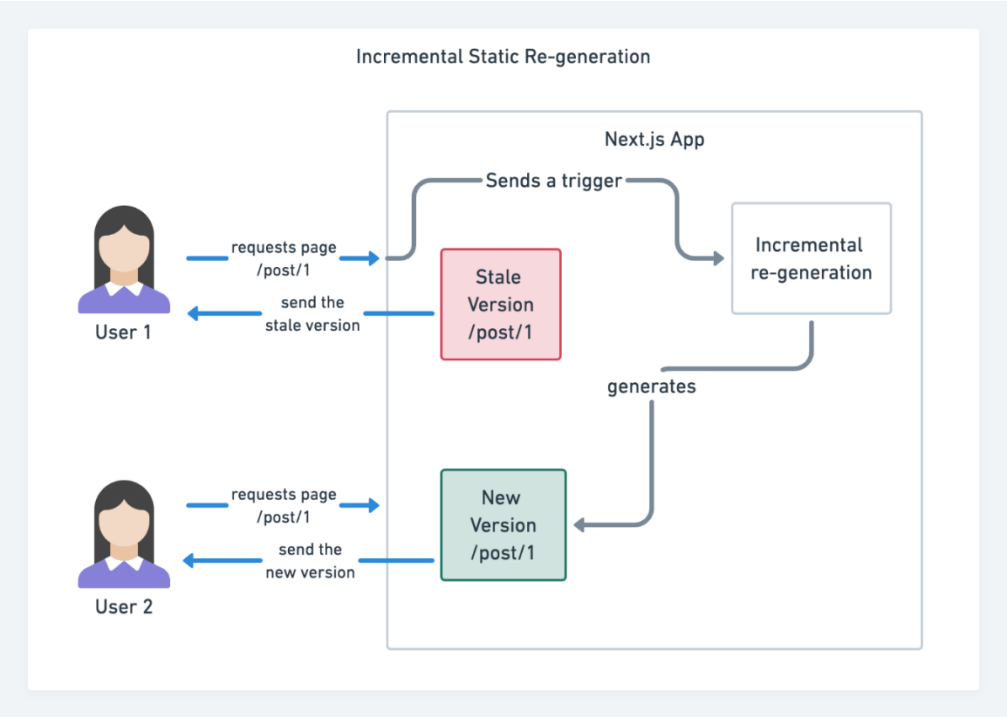


Figure 5. ISR feature of Next.js (Norman, 2020).

2.8 Azure App Service for Cloud Hosting

For the cloud service, Azure was chosen for its developer-friendly UI, strong community support and scalable manner. Azure App Service is the fully managed PaaS offering from Azure that allows developers to build, deploy and scale web applications, serverless function apps and logic apps without underlying infrastructure configuration. Moreover, it provides several integration plugins such as Azure CDN for static file storage, Azure Redis for caching mechanisms and Azure Application Gateway for SSL termination. Azure Deployment slots allow multiple environments for staging deployment, A/B testing and blue-green deployments with minimal downtime.

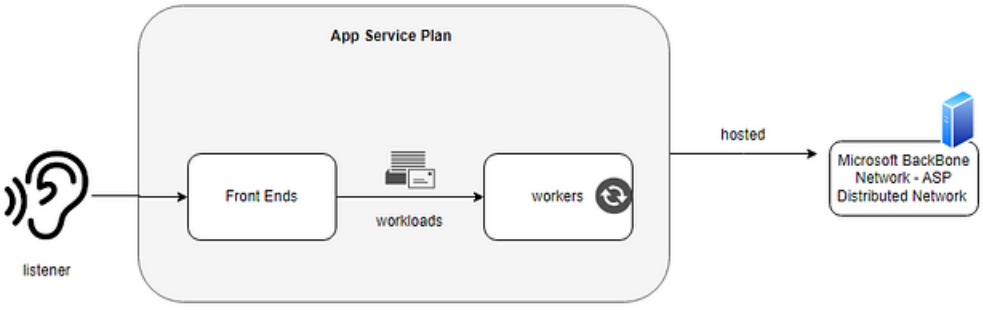


Figure 6. ASP underlying network flow and shared hosting.

Furthermore, Azure Vault manages the secret variables and Azure Active Directory provides the security and access control. Regarding the high availability and disaster recovery, Azure Front Door or Traffic Manager provides geo-replication and backup strategies. App Service diagnostics and Application Insights are helpful for regularly reviewing the system performance metric and diagnosing bugs in the application. The CI/CD pipeline can be effortlessly integrated with Azure DevOps, GitHub Actions, and other popular services allowing the development to be more reliable and time-efficient.

Although AWS Elastic Beanstalk and Google Cloud Run have their own strengths in providing deployment architecture, Azure's pricing model is more cost-effective for this project's scope. Moreover, Microsoft's widespread data center distribution across Nordic regions provides geographical advantages that boost application performance (Microsoft, 2024).

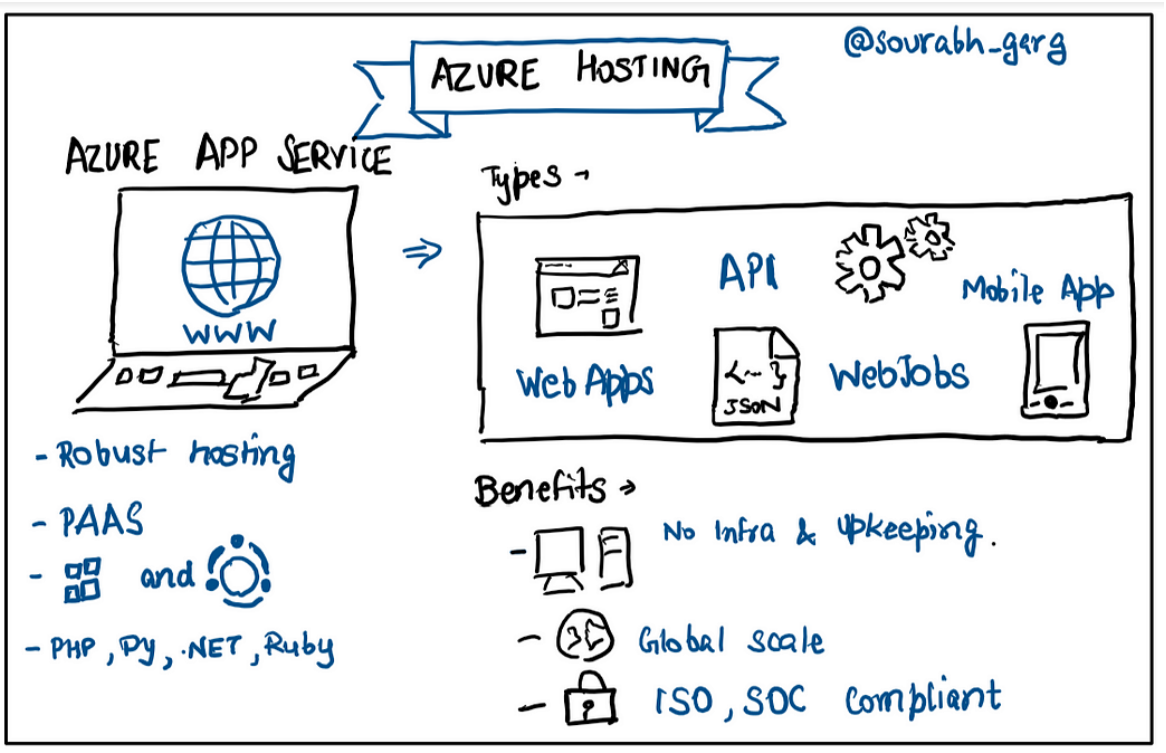


Figure 7. Benefit of Azure App Service Hosting (Garg, 2022).

3 System Architecture

The system architecture consists of multiple embedded sensors collecting the real-time environmental data distributed throughout the resort. However, the embedded system architecture falls outside the scope of the thesis. Instead, a background service is created to transmit real time sensor data to simulate the system's capabilities. The sensor metrics are transmitted to the centralized MQTT server to handle the incoming data streams.

A significant architectural pattern is the pull-based data collection using Telegraf rather than the traditional push-based method. As opposed to the push-based method where data is streaming continuously regardless of system availability, Telegraf acts as an agent to adjust the data ingestion based on the current system requirements. Thus, this intelligent scheduling adds to the benefits by mitigating the risk of system overwhelm. This approach improves the system's reliability and resource efficiency by actively polling the MQTT broker at predetermined intervals and routing it to InfluxDB, database optimized for high throughput sensor data.

Meanwhile, resort's business data is stored inside the Supabase PostgreSQL. The backbone of the system lies in the Next.js full-stack application enabling server-side and client-side rendering. Moreover, the Node.js backend server is implemented using WebSocket architecture to establish bidirectional real-time communication. This ensures sending instant sensor updates directly to the UI whenever new data arrives. Furthermore, it is fundamental for implementing business logic features such as notification system and analytics.

The DevSecOps pipeline integration ensures that security is maintained throughout the software development cycle rather than being treated as a final stage concern. These security tools include static code analysis, dependency vulnerability scanning, container image scanning, and Sonar cloud Code quality checking. Moreover, whole architecture is structured using Docker multi-containerization preventing security threats through container isolation and segmentation. To mitigate the potential vulnerabilities, container base images are used with minimal bundle size and all containers are configured with non-root users.

Regarding security measures, Kinde handles user Authentication providing OAuth 2.0 compliant identity management. To maintain clarity, the role-based access control has been excluded from the scope of this project. All the communication undergoes via TLS/SSL encryption for secure transmission.

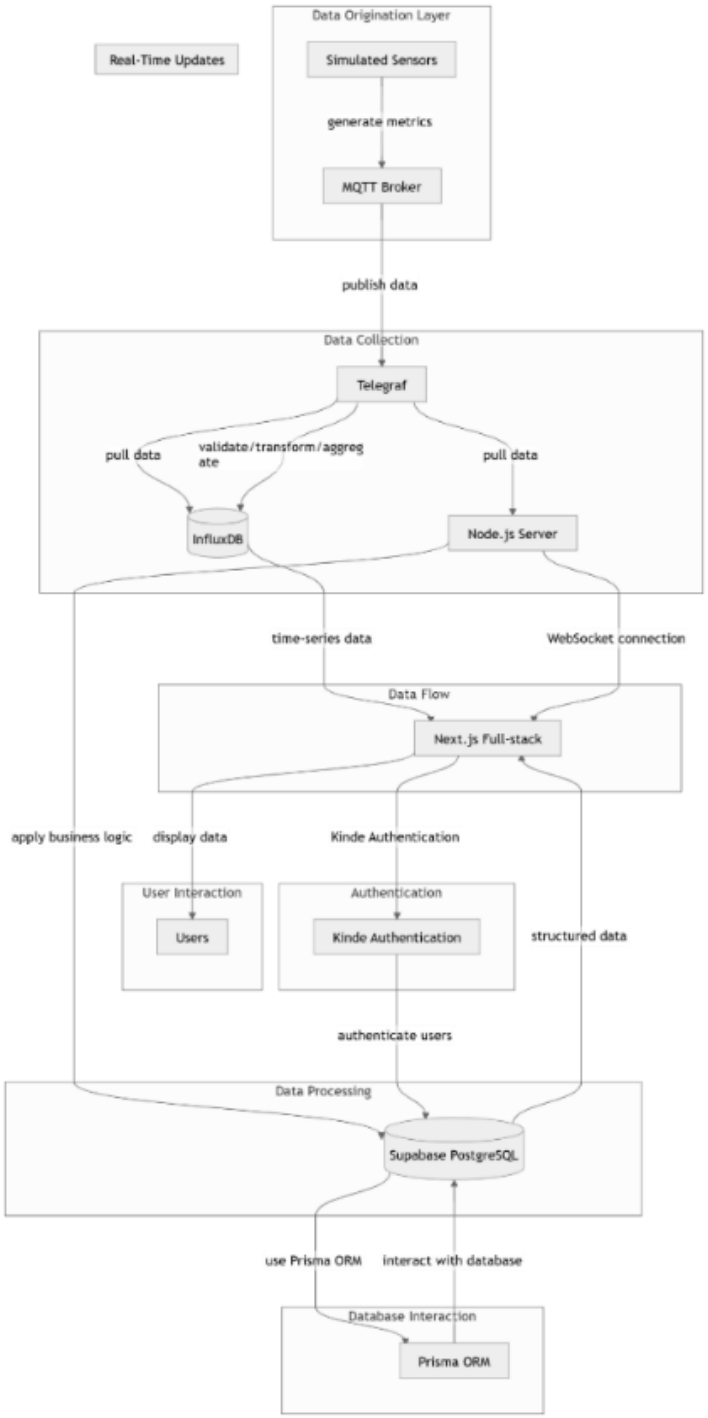


Figure 8. Overall System Architecture.

Supabase's row-level security policies prevent unauthorized data access directly at the database layer, even if application-layer protections are breached. InfluxDB cloud is configured with token-based authentication and separate read/write policies to maintain the principle of least privilege. Security auditing processes are regularly conducted to diagnose vulnerabilities in dependencies and maintain the code security.

4 Implementation

The project repository is structured based on multi-container architecture that separates each service into its own container for scalability and maintainability. It follows the containerization principles, allowing for independent development, testing, and deployment of individual components.

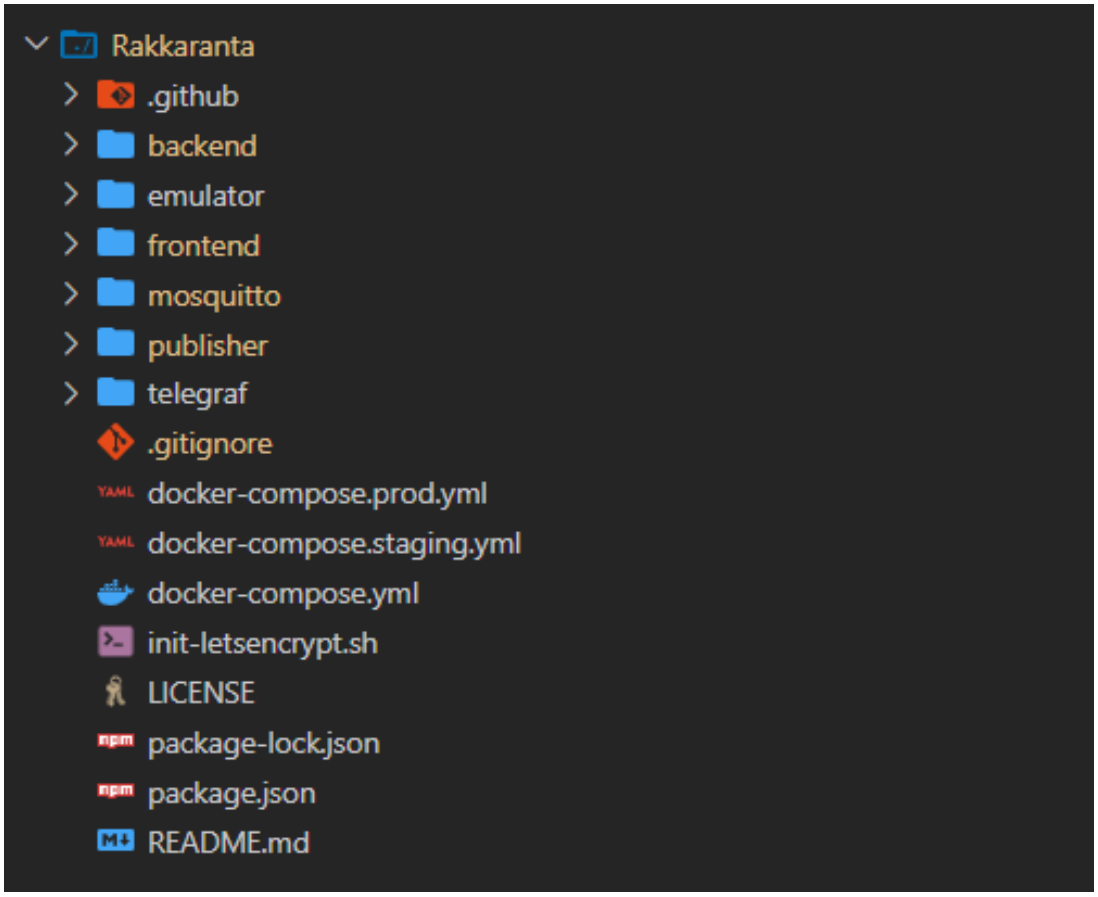


Figure 9. Folder Structure of Rakkaranta Project.

4.1 MQTT Server

The initialization script is created with authentication parameters through environment variables to externally modify the container configuration. It uses default credentials (e.g., "user") if no environment variables are provided to ensure the MQTT broker remains functional in the development stage. The script also creates the necessary files and configures least privilege permissions (600) which grants read and write only to the owner.

```
#!/bin/sh

# Get username/password from environment variables
MQTT_USERNAME=${MQTT_USERNAME:-"yethuhlaing"}
MQTT_PASSWORD=${MQTT_PASSWORD:-"yethuhlaing"}

echo "Setting up Mosquitto password file for user: $MQTT_USERNAME"

# Ensure password file exists and has proper permissions
touch /mosquitto/config/passwd
chmod 600 /mosquitto/config/passwd

# Set wide permissions to ensure we can write
chmod -R 600 /mosquitto
chmod 600 /mosquitto/log/mosquitto.log
chmod 600 /mosquitto/config/passwd
chmod 600 /mosquitto/data/mosquitto.db

# Create/update the user credentials
mosquitto_passwd -b /mosquitto/config/passwd "$MQTT_USERNAME" "$MQTT_PASSWORD"


echo "File permissions:"
ls -la /mosquitto/config/passwd
ls -la /mosquitto/log/mosquitto.log

exec /usr/sbin/mosquitto -c /mosquitto/config/mosquitto.conf

echo "Password file created successfully. Starting Mosquitto..."
```

Figure 10. Bash Script to Initialize Mosquito MQTT broker.

The Mosquitto Dockerfile uses the official eclipse-mosquitto:2.0 as the base image. Next, the mosquitto.conf is copied into the image to replace the default configuration. It contains broker-specific settings, persistence and logging configurations and listening ports. Then, the chmod +x is used to make the script executable by the operating system before the broker launches itself. The health check mechanism verifies the Mosquitto process is running smoothly.

```
FROM eclipse-mosquitto:2.0

# Create directories if they don't exist
RUN mkdir -p /mosquitto/config /mosquitto/data /mosquitto/log

# Copy configuration
COPY ./config/mosquitto.conf /mosquitto/config/mosquitto.conf
COPY ./init.sh /init.sh

# Make the initialization script executable
RUN chmod +x /init.sh

# Set proper ownership for all Mosquitto directories and files
RUN chown -R root:root /mosquitto && \
    touch /mosquitto/config/passwd && \
    chown root:root /mosquitto/config/passwd && \
    chmod 600 /mosquitto/config/passwd

# Health check to verify the MQTT broker is running
HEALTHCHECK --interval=30s --timeout=10s --retries=3 \
    CMD pgrep mosquitto || exit 1

# Expose MQTT and WebSocket ports
EXPOSE 1883 9001 8883

# Run as root (note: this is not recommended for production)
USER root

# Set environment variables
ENV MOSQUITTO_CONFIG_FILE=/mosquitto/config/mosquitto.conf

# Command to run Mosquitto
ENTRYPOINT ["/init.sh"]
```

Figure 11. Production Dockerfile for Mosquito MQTT broker.

4.2 InfluxDB

Setting up InfluxDB starts with creating an account on the official InfluxData platform. Upon successful registration, the most geographically proximate cloud provider region (AWS, Azure or Google Cloud) towards the system's deployment is selected to reduce the latency.

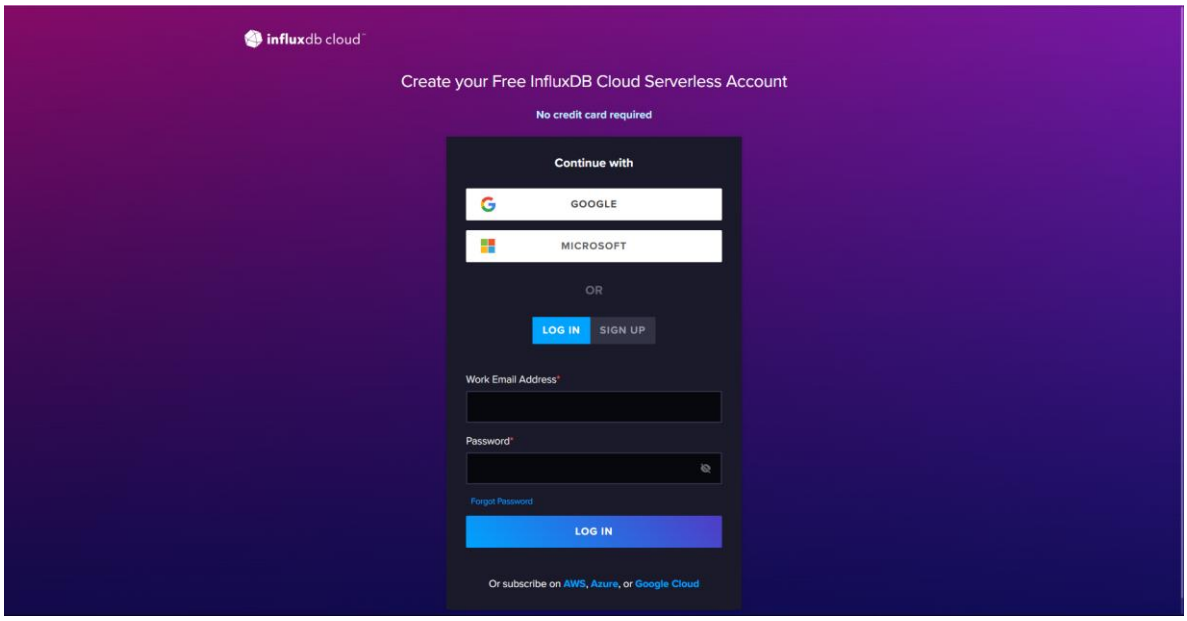


Figure 12. Login In Page of InfluxDB Cloud.

The next step is to create a bucket - the primary storage location for time-series data. The retention policy is set to 24 hours because of limited resources. If historical data is critical to the business analysis, a maximum of 30 days is preferable.

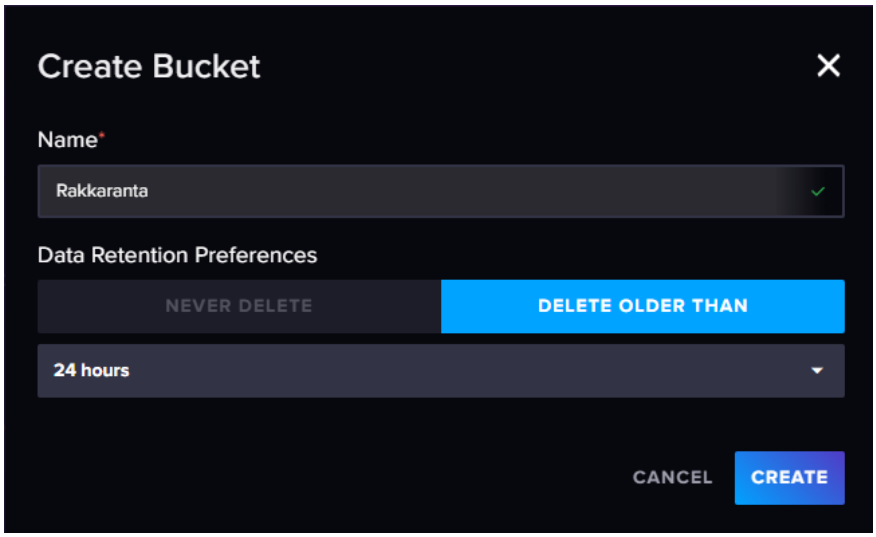


Figure 13. Creating Bucket for Time-series Sensor Data.

After this, an API token must be generated for secure and authenticated communication with the database. It is recommended to set the minimum privilege to read from and write to specific buckets. Following this, the API key is copied to the repository.

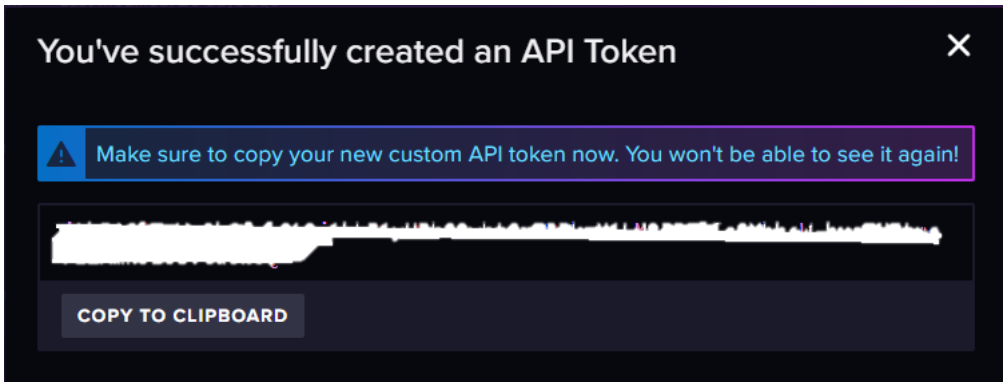


Figure 14. InfluxDB API Token Generation.

4.3 Telegraf

The Telegraf configuration has similar build patterns to the MQTT one. First, the necessary files and proper ownership are set for the “telegraf” user and group. Following this, the custom configuration file is copied into the container with non-root user ownership during operation. The port 5050 is the Telegraf standard port for communication through the actual port. Finally, the Telegraf is launched with the config flag to specify the above-mentioned configuration file's location. This structured method adopts balanced security and maintainability as industry best practice.

```
FROM telegraf:latest

# Create directory structure with proper permissions
RUN mkdir -p /etc/telegraf/telegraf.d && \
    chown -R telegraf:telegraf /etc/telegraf

# Copy our custom configuration file with proper ownership
COPY --chown=telegraf:telegraf ./config/telegraf.conf /etc/telegraf/telegraf.conf

# Set the permissions
RUN chmod 644 /etc/telegraf/telegraf.conf && \
    chmod -R 755 /etc/telegraf

# Expose necessary port
EXPOSE 5050

# Set environment variables
ENV TELEGRAF_CONFIG_PATH=/etc/telegraf/telegraf.conf

# Switch to non-root user for security
USER telegraf

# Run Telegraf
CMD ["telegraf", "--config", "/etc/telegraf/telegraf.conf"]
```

Figure 15. Production Dockerfile for Telegraf.

If the MQTT server and InfluxDB are configured and running successfully, the Telegraf service loads its configuration and connects to the MQTT broker by pulling the sensor data to push it towards InfluxDB as shown in the logging Figure 16.

```
2025-04-30T14:33:26Z I! Loading config: /etc/telegraf/telegraf.conf
2025-04-30T14:33:26Z I! Starting Telegraf 1.34.2 brought to you by InfluxData the makers of InfluxDB
2025-04-30T14:33:26Z I! Available plugins: 239 inputs, 9 aggregators, 33 processors, 26 parsers, 63 outputs, 6 secret-stores
2025-04-30T14:33:26Z I! Loaded inputs: mqtt_consumer
2025-04-30T14:33:26Z I! Loaded aggregators:
2025-04-30T14:33:26Z I! Loaded processors:
2025-04-30T14:33:26Z I! Loaded secretstores:
2025-04-30T14:33:26Z I! Loaded outputs: influxdb_v2
2025-04-30T14:33:26Z I! Tags enabled: host=5f1cd7a39b99
2025-04-30T14:33:26Z I! [agent] Config: Interval:5s, Quiet:false, Hostname:"5f1cd7a39b99", Flush Interval:10s
2025-04-30T14:33:26Z W! [agent] The default value of 'skip_processors_after_aggregators' will change to 'true' with Telegraf v1.40.0! If you need the current default behavior, please
explicitly set the option to 'false'!
2025-04-30T14:33:26Z D! [agent] Initializing plugins
2025-04-30T14:33:26Z D! [agent] Connecting outputs
2025-04-30T14:33:26Z D! [agent] Attempting connection to [outputs.influxdb_v2]
2025-04-30T14:33:26Z D! [agent] Successfully connected to outputs.influxdb_v2
2025-04-30T14:33:26Z D! [agent] Starting service inputs
2025-04-30T14:33:26Z I! [inputs.mqtt_consumer] Connected [mqtt://mosquitto:1883]
2025-04-30T14:33:26Z D! [inputs.mqtt_consumer] Session found [mqtt://mosquitto:1883]
```

Figure 16. Bash Script to Initialize Mosquito MQTT broker.

4.4 Backend

The server infrastructure is instantiated with Express.js – the Node.js web application framework, serving HTTP requests. With the WebSocket server integrated into the pre-existing web server, both HTTP and WebSocket communication are enabled on the same port. The heart of this integration lies in the 'upgrade' event listener intercepting the transition between HTTP and WebSocket. This path '/api/realtime/sensors' is acknowledged as the only legitimate route for orchestrating the WebSocket handshake operation. Conversely, if a different route is hit by an 'upgrade' request, it is automatically destroyed as shown in Figure 17.

```
const app = express();
const server = http.createServer(app);

// Create a single WebSocket server
const wss = new WebSocketServer({ noServer: true })

// Store client subscriptions and their data
const clientSubscriptions = new Map()

wss.on('connection', (ws) => {···
})

server.on('upgrade', (request, socket, head) => {
    const { pathname } = url.parse(request.url)

    if (pathname === '/api/realtime/sensors') {
        wss.handleUpgrade(request, socket, head, (ws) => {
            wss.emit('connection', ws, request)
        })
    } else {
        socket.destroy()
    }
})

// Route to initialize WebSocket connection
app.get('/api/realtime/', (req, res) => {
    res.send('WebSocket server is running. Connect to ws://localhost:5000/api/realtime/sensors');
});

const PORT = 5000
server.listen(PORT, () => {
    console.log(`Server running on port ${PORT}`)
})
```

Figure 17. Node.js Backend server Implementation.

For interaction with the InfluxDB database, the InfluxDB client is instantiated with authentication parameters and gzip compression for incoming requests. Then, the queryApi method is created on the pre-existing InfluxDB instance, and this utility executes Flux queries to fetch time-series sensor data.

```
import { InfluxDB, Point } from "@influxdata/influxdb-client";

const QUERY_DURATION = "1d";

const token = process.env.DOCKER_INFLUXDB_INIT_ADMIN_TOKEN;
const influx_url = process.env.INFLUXDB_URL;
const org = process.env.DOCKER_INFLUXDB_INIT_ORG;
const bucket = process.env.DOCKER_INFLUXDB_INIT_BUCKET;

const influxDB = new InfluxDB({
  url: influx_url,
  token,
  transportOptions: {
    gzipThreshold: 1, // Compress all requests
  },
});
const queryApi = influxDB.getQueryApi(org);
```

Figure 18. InfluxDB initialization.

The "querySensorData" is a modular abstraction layer on top of InfluxDB to retrieve time-series data with three parameters – sensor type, duration, and optimal last timestamp. The Flux query is intelligently constructed to retrieve either the recent data or data within a broader time window. Then it processes the raw data and transforms it into structured data or rejects it if errors are found.

```
export async function querySensorData(sensor, timeRange, lastTimestamp = null) {
  let fluxQuery;

  if (timeRange === "last") {
    fluxQuery = `
            from(bucket: "${bucket}")
                |> range(start: -20s)
                |> filter(fn: (r) => r.topic == "sensors/${sensor}")
                |> keep(columns: ["_value", "_time", "_field"])
                |> pivot(rowKey:["_time"], columnKey: ["_field"], valueColumn: "_value")
        `;
  } else {
    const start = lastTimestamp ? lastTimestamp : `-${timeRange}`;
    fluxQuery = `
            from(bucket: "${bucket}")
                |> range(start: ${start})
                |> filter(fn: (r) => r.topic == "sensors/${sensor}")
                |> keep(columns: ["_value", "_time", "_field"])
                ${
                  timeRange > "1h"
                    ? "|> aggregateWindow(every: 5s, fn: mean)"
                    : ""
                }
        `;
  }
  return new Promise((resolve, reject) => {
    let sensorData = [];
    queryApi.queryRows(fluxQuery, {
      next: (row, tableMeta) => {
        const o = tableMeta.toObject(row);
        sensorData.push(o);
      },
      error: (error) => {
        reject(error);
      },
      complete: () => {
        resolve(sensorData);
      },
    });
  });
}
```

Figure 19. Sensor Data Retrieval method.

4.5 MQTT Publisher

The MQTT publisher is a simulated service to demonstrate the data flow since the physical embedded system is excluded from this scope. This publisher adopts the publish-subscribe architecture through an event-driven approach. The option object is primarily created to connect to the MQTT Broker, and it is composed of MQTT username, password, and connection resilience parameters. The 'clean' parameter decides whether to collect stateless or stateful data. The PUBLISH_INTERVAL stands for writing data to InfluxDB every five seconds. The MAX_RETRIES (3) minimizes the overhead of high-volume data streaming.

```
import mqtt from 'mqtt'
import { generateSensorData } from './data.js'

const mqtt_url = process.env.MOSQUITTO_BROKER_URL || "tcp://mosquitto:1883";
const mqtt_username = process.env.MQTT_USERNAME
const mqtt_password = process.env.MQTT_PASSWORD;
const PUBLISH_INTERVAL = 5 * 1000 // 5 seconds
const MAX_RETRIES = 3
const RETRY_DELAY = 1000 // 1 second

const options = {
  // MQTT connection options
  clientId: "mqtt_js_client_" + Math.random().toString(16).substr(2, 8),
  username: mqtt_username, // Use your MQTT username
  password: mqtt_password, // Use your MQTT password
  clean: true, // Set to false if you want to receive missed messages when reconnecting
  reconnectPeriod: 5000, // Try to reconnect every 5 seconds
  connectTimeout: 30 * 1000, // Timeout period of 30 seconds
};

class MQTTPublisher {
    constructor() {
        this.client = mqtt.connect(mqtt_url, options);

        this.client.on('connect', () => {
            console.log('Connected to MQTT broker')
        })

        this.client.on('error', (error) => {
            console.error('MQTT client error:', error)
        })

        this.isRunning = false
        this.publishCount = 0
        this.errorCount = 0
        this.lastPublish = null
        this.publishInterval = null
    }
```

Figure 20. MQTT Publisher Class Initialization.

The MQTT Publisher Class needs to generate the array of Point objects, each correspond to a collection of sensor readings with required tags and fields for InfluxDB syntax. The simulated sensors include air quality (CO, NO2, O3, SO2), gas leaks (methane, propane, ammonia, hydrogen, ozone), emergency status, population in different location, temperature, noise, vibration, motion detectors, light intensity, humidity, water flow, and water level.

```
export const generateSensorData = () => {
  const now = new Date();
  return [
    // Gas
    new Point("airSensors")
      .tag("sensor_type", "airSensor")
      .tag("sensor", "co")
      .floatField("co", Math.random() * 100)
      .timestamp(now),
    new Point("airSensors")
      .tag("sensor_type", "airSensor")
      .tag("sensor", "pm10")
      .floatField("pm10", Math.random() * 250)
      .timestamp(now),
    new Point("airSensors")
      .tag("sensor_type", "airSensor")
      .tag("sensor", "no2")
      .floatField("no2", Math.random() * 300)
      .timestamp(now),

    // Temperature
    new Point("airSensors")
      .tag("sensor_type", "airSensor")
      .tag("sensor", "temperature")
      .floatField("temperature", Math.random() * 50 + 20)
      .timestamp(now),

    // Noise Sensors
    new Point("noiseSensors")
      .tag("sensor_type", "noiseSensor")
      .tag("sensor", "noise-level")
      .floatField("noise-level", (50 + Math.random() * 40).toFixed(1))
      .timestamp(now),

    // Vibration Sensors
    new Point("vibrationSensors")
      .tag("sensor_type", "vibrationSensor")
      .tag("sensor", "vibration")
      .floatField("vibration", Math.floor(Math.random() * 8) + 1)
      .timestamp(now),

    // Motion Detectors
    new Point("motionSensors")
      .tag("sensor_type", "motionSensor")
      .tag("sensor", "motion_detected")
      .booleanField("motion_detected", Math.random() > 0.7)
      .timestamp(now),

    // Light Intensity Sensors
    new Point("lightSensors")
      .tag("sensor_type", "lightSensor")
      .tag("sensor", "light-intensity")
      .floatField("light-intensity", (Math.random() * 1000).toFixed(2))
      .timestamp(now),

    // Gas Leakage Detectors
    new Point("gasLeakage")
      .tag("sensor_type", "gasSensor")
      .tag("sensor", "gas")
      .floatField("methane", Math.random() * 50)
      .floatField("propane", Math.random() * 50)
      .floatField("hydrogen", Math.random() * 50) // Hydrogen concentration
      .floatField("ammonia", Math.random() * 50) // Ammonia concentration
      .floatField("ozone", Math.random() * 30)
      .timestamp(now),

    // // Emergency Status
    new Point("emergency")
      .tag("sensor_type", "emergencySensor")
      .tag("sensor", "emergency")
      .floatField("emergency", (Math.random() * 100).toFixed(2))
      .timestamp(now),

    // // Visitors
    new Point("population")
      .tag("sensor_type", "populationSensor")
      .tag("sensor", "population")
      .intField("reception", Math.floor(40 + Math.random() * 20)) // Lobby population
      .intField("sauna", Math.floor(20 + Math.random() * 15)) // Storage population
      .intField("woodshed", Math.floor(60 + Math.random() * 20)) // Office population
      .intField("restaurant", Math.floor(60 + Math.random() * 20)) // Security population
      .intField("office", Math.floor(60 + Math.random() * 20)) // Cafeteria population
      .intField("lakeside", Math.floor(60 + Math.random() * 20)) // Inspection population
      .intField("cottage", Math.floor(60 + Math.random() * 20)) // Automation population
      .intField("firepit", Math.floor(60 + Math.random() * 20)) // Maintenance population
      .timestamp(now),

    // humidity
    new Point("airSensors")
      .tag("sensor_type", "airSensor")
      .tag("sensor", "humidity")
      .floatField("reception", Math.random() * 100) // Lobby humidity
      .floatField("sauna", Math.random() * 100) // Storage humidity
      .floatField("woodshed", Math.random() * 100) // Office humidity
      .floatField("restaurant", Math.random() * 100) // Security humidity
      .floatField("office", Math.random() * 100) // Cafeteria humidity
      .floatField("lakeside", Math.random() * 100) // Inspection humidity
      .floatField("cottage", Math.random() * 100) // Automation humidity
      .floatField("firepit", Math.random() * 100) // Maintenance humidity
      .timestamp(now),

    // Water Flow
    new Point("water-flow")
      .tag("sensor_type", "water-flow_detector")
      .tag("sensor", "water-flow")
      .tag("location", "reservoir")
      .floatField("water-flow", Math.random() * 60) // Flow rate in liters per minute
      .timestamp(now),

    // Water Level
    new Point("waterLevel")
      .tag("sensor_type", "water-level_detector")
      .tag("sensor", "water-level")
      .tag("location", "reservoir")
      .floatField("water-level", Math.random() * 10) // Water level in meters
      .timestamp(now),
  ];
```

Figure 21. Simulated InfluxDB Data Points.

The asynchronous publish method encapsulates the ‘generateSensorData’ method to generate the InfluxdDB points. Afterwards, the publisher instance sends each generated data to the corresponding topic with Promise. The publish counts are tracked with incremental variable for the logging process. The method will be retried until it reaches the MAX_RETRIES. This backoff strategy implicitly mitigate the network problems or temporary database downtime, improving the overall data ingestion process.

```
async publishDataWithRetry(retryCount = 0) {
    try {
        const sensorData = generateSensorData()

        // Add timestamp if not present
        sensorData.forEach(data => {
            if (!data.timestamp) {
                data.timestamp = new Date().toISOString()
            }
        })

        const publishPromises = sensorData.map(data => {
            return new Promise((resolve, reject) => {
                const topic = `sensors/${data.measurement}`;
                this.client.publish(topic, JSON.stringify(data), (err) => {
                    if (err) reject(err)
                    else resolve()
                })
            })
        })

        await Promise.all(publishPromises)

        this.publishCount += sensorData.length
        this.lastPublish = new Date()

        console.log(`Successfully published ${sensorData.length} messages to MQTT at ${this.lastPublish.toISOString()}`)
        console.log(`Total publishes: ${this.publishCount}, Total errors: ${this.errorCount}`)

        return true
    } catch (error) {
        this.errorCount++
        console.error(`Error publishing to MQTT (attempt ${retryCount + 1}/${MAX_RETRIES}):`, error.message)

        if (retryCount < MAX_RETRIES) {
            console.log(`Retrying in ${RETRY_DELAY}ms...`)
            await new Promise(resolve => setTimeout(resolve, RETRY_DELAY))
            return this.publishDataWithRetry(retryCount + 1)
        } else {
            console.error('Max retries reached, skipping this batch')
            return false
        }
    }
}
```

Figure 22. MQTT Publisher Method.

The 'start' method utilizes a recursive 'setTimeout' pattern combined with async/await for efficient data transfer – a fundamental IoT data transmission mechanism under unstable network connections. The 'stop' method removes any pending timeouts and terminates the publishing operation with a Promise-based asynchronous pattern. It can also handle potential errors with a robust try/catch mechanism.

```
async start() {
    if (this.isRunning) {
        console.warn('Publisher is already running')
        return
    }

    this.isRunning = true
    console.log('Starting MQTT publisher...')

    const scheduleNextPublish = async () => {
        if (!this.isRunning) return

        await this.publishDataWithRetry()

        // Schedule next publish using setTimeout
        this.publishInterval = setTimeout(scheduleNextPublish, PUBLISH_INTERVAL)
    }

    // Start the first publish
    scheduleNextPublish()
}
```

Figure 23. Method to run MQTT Publisher service.

```
async stop() {
    console.log('Stopping MQTT publisher...')
    this.isRunning = false

    if (this.publishInterval) {
        clearTimeout(this.publishInterval)
        this.publishInterval = null
    }

    try {
        await new Promise((resolve, reject) => {
            this.client.end(false, {}, (err) => {
                if (err) reject(err)
                else resolve()
            })
        })
        console.log('Successfully closed MQTT connection')

        // Log final statistics
        console.log(`Final statistics:`)
        console.log(`- Total successful publishes: ${this.publishCount}`)
        console.log(`- Total errors: ${this.errorCount}`)
        console.log(`- Last successful publish: ${this.lastPublish ? this.lastPublish.toISOString() : 'never'}`)
    } catch (error) {
        console.error('Error while closing MQTT connection:', error)
        throw error
    }
}
```

Figure 24. Method to gracefully stop MQTT Publisher service.

Finally, the MQTT Publisher service utilizes signal handlers to receive SIGINT and SIGTERM system events for graceful shutdown — important for data consistency in IoT communication systems. This approach follows defensive programming principles by consolidating the shutdown logic into a single function.

```
// Create publisher instance
const publisher = new MQTTPublisher()

// Handle process termination
async function handleShutdown(signal) {
    console.log(`Received ${signal} signal`)
    try {
        await publisher.stop()
        console.log('Graceful shutdown completed')
        process.exit(0)
    } catch (error) {
        console.error('Error during shutdown:', error)
        process.exit(1)
    }
}

// Register shutdown handlers
process.on('SIGINT', () => handleShutdown('SIGINT'))
process.on('SIGTERM', () => handleShutdown('SIGTERM'))
process.on('uncaughtException', (error) => {
    console.error('Uncaught exception:', error)
    handleShutdown('uncaughtException')
})
process.on('unhandledRejection', (reason, promise) => {
    console.error('Unhandled rejection at:', promise, 'reason:', reason)
    handleShutdown('unhandledRejection')
})

// Start the publisher
publisher.start().catch(error => {
    console.error('Failed to start publisher:', error)
    process.exit(1)
})

console.log('MQTT publisher started. Press Ctrl+C to stop.')
```

Figure 25. Running the MQTT Publisher service.

For the production, the dockerfile uses the lightweight alpine-based node image for the resource efficiency. Then, the package.json file is copied to install the dependencies according to the sequential caching layer. Afterwards, the source code is copied and run the publisher script.

```
FROM node:23.11.0-alpine3.18

WORKDIR /app
COPY package*.json ./

RUN npm install
COPY . .

# Create a non-root user
RUN addgroup -S -g 1001 appgroup && \
    adduser -S -u 1001 -G appgroup appuser

# Set permissions for the application files
RUN chown -R appuser:appgroup /app

# Switch to the non-root user
USER appuser

CMD ["node", "seed.js"]
```

Figure 26. MQTT Publisher Production Dockerfile.

In the docker logs, it publishes constantly to the MQTT broker at certain intervals.

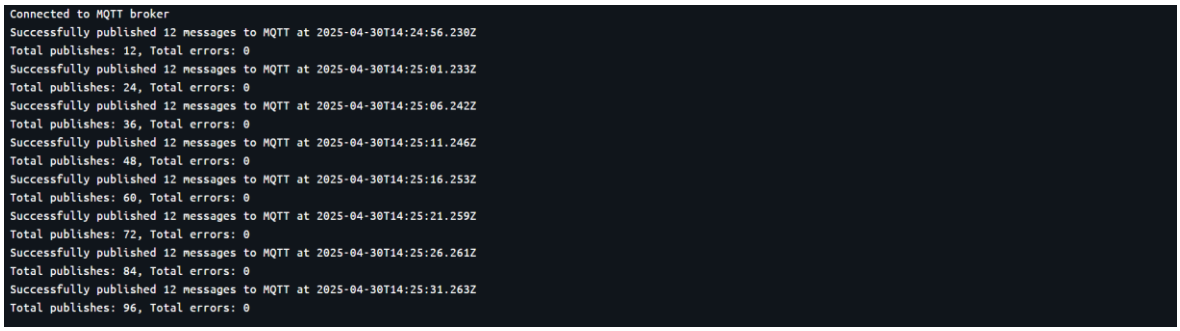


Figure 27. Logs from MQTT Publisher Container.

4.6 Frontend

4.6.1 Folder structure

The core frontend codebase is organized into the 'src' folder, excluding the 'public', 'prisma' folders and other configuration files. Next.js server actions are placed inside the 'actions' folder, and 'app' is the main directory in which all the route folders are constructed. The

shadCN UI components and custom UI are kept inside the 'components' folder. The 'config' folder contains static content data to be automatically embedded into the application. The 'middleware' file should be placed at the same level as the 'app' folder to check authentication at the edge level. The image files and all accessories are stored inside the 'assets' directory, and the 'lib' folder contains third-party software functionalities.

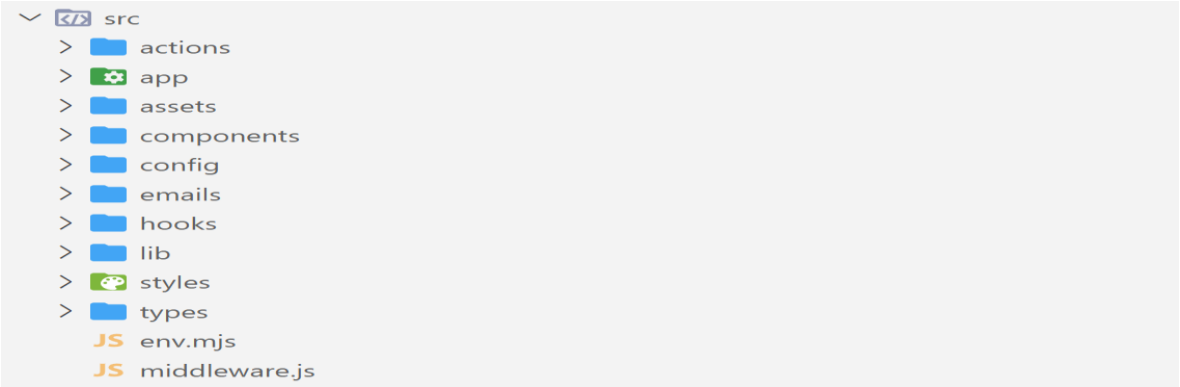


Figure 28. Folder Structure for Frontend Service.

4.6.2 Kinde Authentication

Kinde is the third-party service provider for authentication service to scale and comply with regulations. To obtain credentials, account registration must be completed on the Kinde platform. Then, all the following variables are pasted into the .env file.

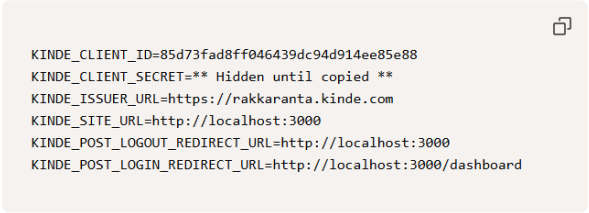


Figure 29. Storing Kinde Credentials.

Next, the callback URLs of both development and production environment must be filled out in the Kinde platform.

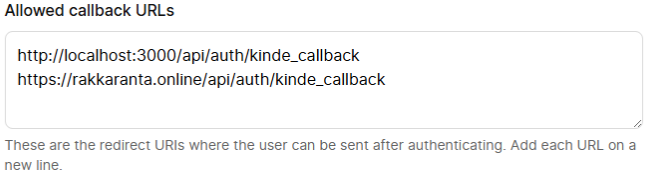


Figure 30. Adding the Kinde Callback URL.

Kinde provides a plethora of authentication methods including social connections, pass wordless connections, and password connection. The social connections operate through secure OAuth authorization protocol, allowing customers to authenticate frictionlessly without concerns associated with password management. Moreover, users tend to trust large enterprise corporations with their personal data. The Authentication UI pages can be found in Appendix 2.

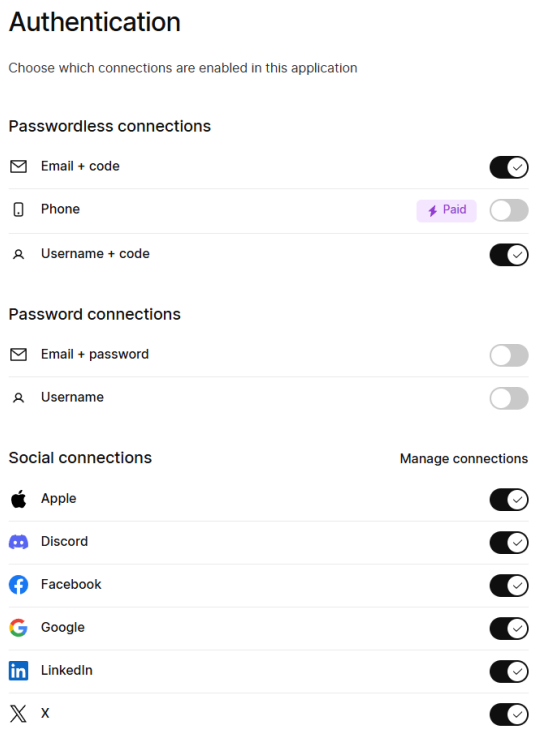


Figure 31. Toggling on Kinde Authentication Modes.

After setting up in Kinde platform, the Kinde SDK installation for the Next.js framework starts with **'npm install @kinde-oss/kinde-auth-nextjs'**. As shown in Figure 29, the middleware file is created including the Kinde built-in authentication check to protect routes restrict and unwanted requests.

```
import { withAuth } from "@kinde-oss/kinde-auth-nextjs/middleware";

export default function middleware(req) {
  return withAuth(req);
}

export const config = {
  matcher: ["/dashboard/:path*"],
}
```

Figure 32. Kinde Middleware for Route Protection.

4.6.3 Basic Configuration

The .env file is composed of third-party API keys, secret credentials, and configuration settings to execute the application. The Next.js configuration defines the URL where the application serves in development or production environments. The database configuration pertains to the parameter strings to connect with Supabase wrapped by Prisma ORM. The main API key is the OpenWeather secret which retrieves real-time environmental data. The rest of the third-party API keys such as Stripe and Resend are optional because these services play a minor role in this project's scope.

```
NEXT_PUBLIC_APP_URL=<"------------------------------">
NEXTAUTH_URL=<"------------------------------">
NEXT_PUBLIC_WEBSOCKET_URL=<"------------------------------">
# ------------------------------------------------------------------------------
# Authentication (Kinde)
# ------------------------------------------------------------------------------
KINDE_CLIENT_ID=<"------------------------------">
KINDE_CLIENT_SECRET=<"------------------------------">
KINDE_ISSUER_URL=<"------------------------------">
KINDE_SITE_URL=<"------------------------------">
KINDE_POST_LOGOUT_REDIRECT_URL=<"------------------------------">
KINDE_POST_LOGIN_REDIRECT_URL=<"------------------------------">

PRODUCT_DEMO_LINK=<"------------------------------">
# ------------------------------------------------------------------------------
# Database (Supabase - Prisma)
# ------------------------------------------------------------------------------

DATABASE_URL=<"------------------------------">
DIRECT_URL=<"------------------------------">

# ------------------------------------------------------------------------------
# Email (Resend)
# ------------------------------------------------------------------------------
RESEND_API_KEY=<"------------------------------">
EMAIL_FROM=<"------------------------------">
EMAIL_SERVER_HOST=<"------------------------------">
EMAIL_SERVER_PORT=<"------------------------------">
EMAIL_SERVER_USER=<"------------------------------">
EMAIL_SERVER_PASSWORD=<"------------------------------">
# ------------------------------------------------------------------------------
# Subscriptions (Stripe)
# ------------------------------------------------------------------------------
STRIPE_API_KEY=<"------------------------------">
STRIPE_WEBHOOK_SECRET=<"------------------------------">

NEXT_PUBLIC_STRIPE_PRO_MONTHLY_PLAN_ID=<"------------------------------">
NEXT_PUBLIC_STRIPE_PRO_YEARLY_PLAN_ID=<"------------------------------">

NEXT_PUBLIC_STRIPE_BUSINESS_MONTHLY_PLAN_ID=<"------------------------------">
NEXT_PUBLIC_STRIPE_BUSINESS_YEARLY_PLAN_ID=<"------------------------------">

NEXT_PUBLIC_STRIPE_ESSENTIAL_PLAN_ID=
NEXT_PUBLIC_STRIPE_ESSENTIAL_PLAN_ID=

OPENWEATHER_API_KEY=<"------------------------------">
```

Figure 33. Example of .env File.

The health check endpoint is recommended to exist in the Azure production environment. If the route is hit, it sends basic server metrics in JSON response including operational status, uptime, timestamp, application version, and comprehensive memory usage.

```
export async function GET(): Promise<NextResponse> {
    try {
        // Basic health status
        const healthStatus: HealthStatus = {
            status: 'ok',
            uptime: process.uptime(),
            timestamp: new Date().toISOString(),
            version: process.env.APP_VERSION || '1.0.0',
            memory: {
                rss: '0 MB',
                heapTotal: '0 MB',
                heapUsed: '0 MB',
            }
        };

        // Check memory usage
        const memoryUsage = process.memoryUsage();
        healthStatus.memory = {
            rss: `${Math.round(memoryUsage.rss / 1024 / 1024)} MB`,
            heapTotal: `${Math.round(memoryUsage.heapTotal / 1024 / 1024)} MB`,
            heapUsed: `${Math.round(memoryUsage.heapUsed / 1024 / 1024)} MB`,
        };

        return NextResponse.json(healthStatus, { status: 200 });
    } catch (error) {
        return NextResponse.json(
        {
            status: 'error',
            message: error instanceof Error ? error.message : 'Unknown error'
        },
        { status: 500 }
        );
    }
}
```

Figure 34. Health Check Route for Azure.

4.6.4 Prisma ORM Installation

Regarding database queries, Prisma, a next-generation ORM, is utilized to replace the cumbersome traditional SQL queries with JSON-like syntax and type-safety. Prisma generates the type-safe client according to the declared schema. Moreover, Prisma's abstraction removes database operations between data storage and business logic. The installation of Prisma starts with the command "npm install @prisma/client –save-dev". Afterwards, the command "npx prisma init" initialize the Prisma schema file and 'User' schema is defined in the following Figure 31.

```
model User {
  id            String    @id @default(cuid())
  name          String?
  email         String?   @unique
  image         String?
  emailVerified DateTime?
  createdAt     DateTime  @default(now()) @map(name: "created_at")
  updatedAt     DateTime  @default(now()) @map(name: "updated_at")
  role          UserRole  @default(USER)

  stripeCustomerId       String?   @unique @map(name: "stripe_customer_id")
  stripeSubscriptionId   String?   @unique @map(name: "stripe_subscription_id")
  stripePriceId          String?   @map(name: "stripe_price_id")
  stripeCurrentPeriodEnd DateTime? @map(name: "stripe_current_period_end")
}
```

Figure 35. User Schema for Prisma.

Upon successful schema creation, the following commands are used to apply migrations to synchronize the database with the declared schema and generate the Prisma client.

```
yethu@MSI MINGW64 ~/Development/Rakkaranta/frontend (hotfix/devsecops-pipeline)
$ npx prisma migrate dev --name init && npx prisma generate
Environment variables loaded from .env
Prisma schema loaded from prisma\schema.prisma
Datasource "db": PostgreSQL database "postgres", schema "public" at "aws-0-eu-central-1.pooler.supabase.com:5432"
```

Figure 36. Commands for Prisma Migration and Prisma Client Generation.

However, a singleton Prisma Client class must be created to prevent multiple database connections. In the development environment, the Prisma instance is shared to handle hot reloading, ensuring it is not re-initialized each time the application reloads.

```
import { PrismaClient } from '@prisma/client'

const prismaClientSingleton = () => {
    return new PrismaClient()
}

declare const globalThis: {
    prismaGlobal: ReturnType<typeof prismaClientSingleton>;
} & typeof global;

const prisma = globalThis.prismaGlobal ?? prismaClientSingleton()

export default prisma

if (process.env.NODE_ENV !== 'production') globalThis.prismaGlobal = prisma
```

Figure 37. Prisma Client for reusability.

4.6.5 OpenWeather API integration

As regards reliable access to meteorological information, the OpenWeather API offers compelling features and straightforward documentation. The OpenWeather API token can be generated on the platform after account registration.

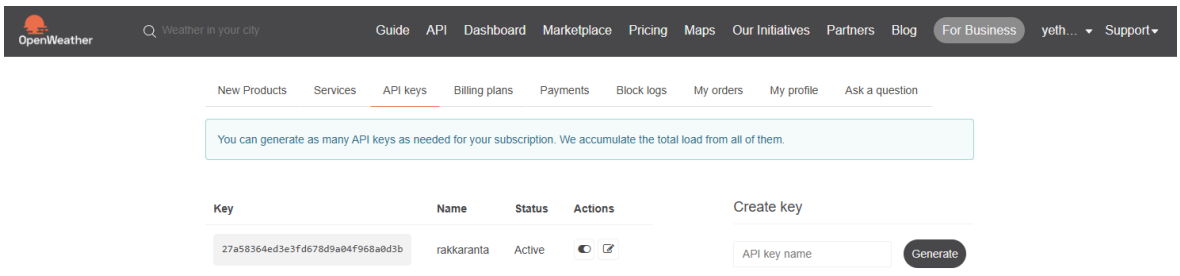


Figure 38. OpenWeather API token Generation.

The following asynchronous function fetches the real-time weather from the third-party service ‘OpenWeather’ with the additional parameters — geographical coordinates (latitude and longitude), and the metric unit of measurement. These parameters dynamically construct the URL, with a try-catch block to capture potential errors.

```
// Function to fetch weather data from OpenWeatherMap API
export async function fetchWeatherData(lat = DEFAULT_LAT, lon = DEFAULT_LON): Promise<WeatherData> {
    try {
        const response = await fetch(
        `https://api.openweathermap.org/data/3.0/onecall?lat=${lat}&lon=${lon}&units=metric&exclude=minutely&appid=${API_KEY}`,
        )
        if (!response.ok) {
            throw new Error(`Weather API error: ${response.status}`)
        }

        const data = await response.json()
        return data
    } catch (error) {
        console.error("Failed to fetch weather data:", error)
        throw error
    }
}
```

Figure 39. Fetching real-time weather data via OpenWeather API.

The ‘fetchWeatherData’ function is wrapped into the higher-order function with a caching layer. The caching configuration includes a static cache key ('weather-data') with revalidation duration in order to minimize the frequent API requests, saving the OpenWeather API rate limits and subscription bills.

```
"use server"

import { WEATHER_CACHE_DURATION , DEFAULT_LAT, DEFAULT_LON} from "@/config/weather";
import { fetchWeatherData } from "@/lib/open-weather";
import { unstable_cache } from "next/cache"
// Cache configuration
/**
 * Create a cached version of the API fetch function using unstable_cache
 */
export const fetchCachedWeatherData = unstable_cache(
    async (lat = DEFAULT_LAT, lon = DEFAULT_LON) => {
        console.log(`Fetching fresh weather data for ${lat},${lon}`);
        return fetchWeatherData(lat, lon);
    },
    // Corrected: Make the key static
    ['weather-data'],
    {
        revalidate: WEATHER_CACHE_DURATION, // Cache duration in seconds
        tags: ['weather-data'], // Tag for manual revalidation
    }
);
```

Figure 40. Optimizing OpenWeather API Usage with Caching.

The utility functions are created to visualize the data in more user-friendly way by mapping the OpenWeather weather Id with descriptive weather conditions. In such scenarios, using AI to decide weather conditions is unnecessary because of the data mapping information published by official documentation.

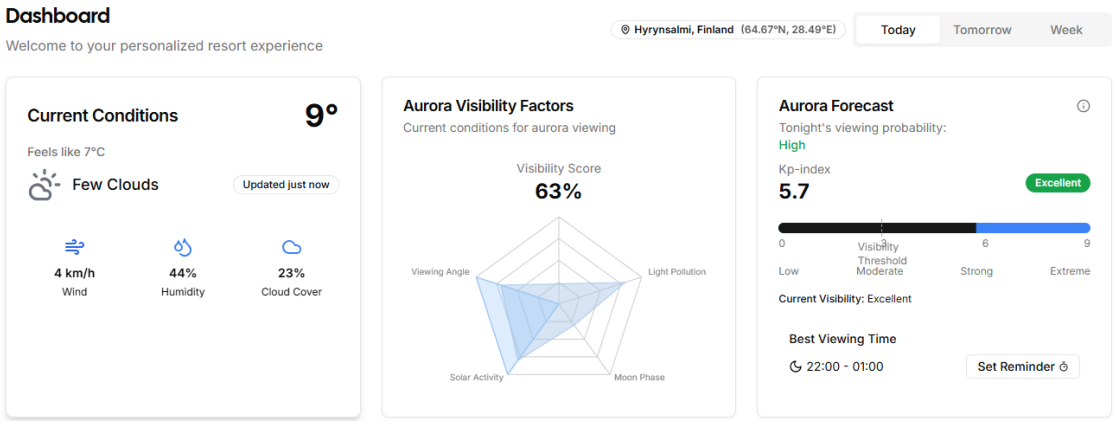


Figure 41. Personalized IoT dashboard for Rakkaranta Resort showing real-time weather data, aurora visibility factors, and aurora forecast.

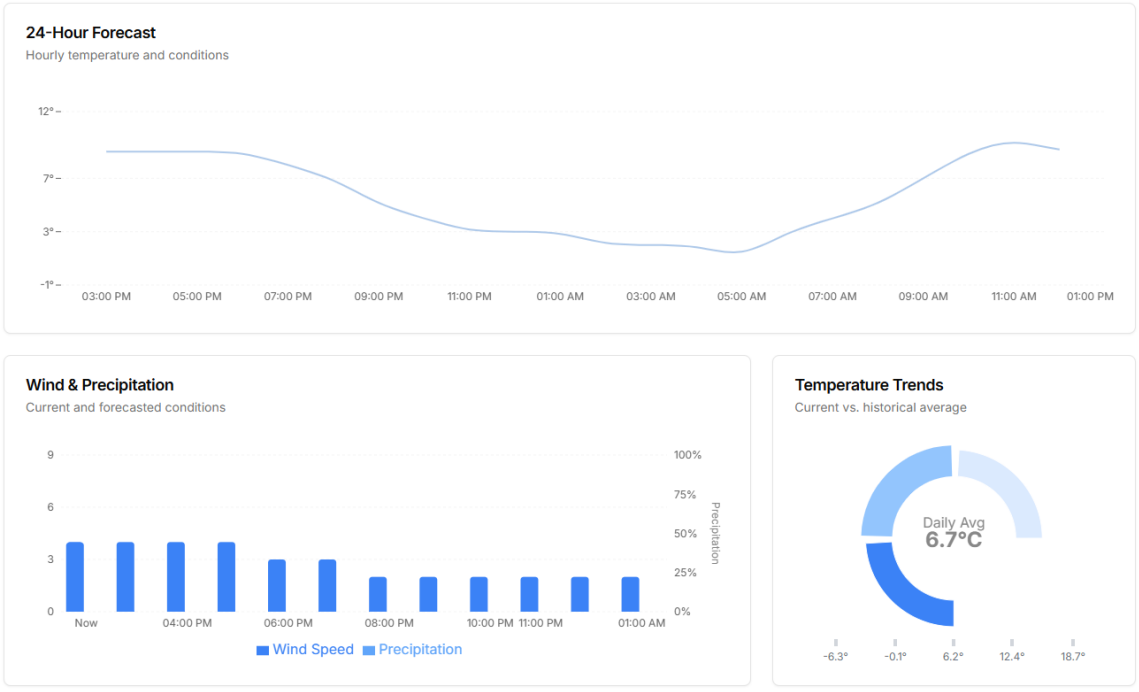


Figure 42. Personalized IoT dashboard for Rakkaranta Resort showing real-time temperature, wind & precipitation.

4.6.6 Activities Recommendation System

The Activities Recommendation System is implemented using a computational algorithmic approach instead of using reasoning-based AI model inference. Real time meteorological information is a critical determinant in this system. The system contains four primary components:

1. A comprehensive activity data type with environmental readings
2. An algorithm that evaluates the activities according to meteorological data
3. A natural language generation system for recommendation contextualization
4. A recommendation engine that employs all these elements to generate prioritized activity suggestions.

As shown in Figure 43, the activity data type consists of a unique Id, activity names, timeframe, location and environmental conditions (temperature limitations, wind speed tolerances, precipitation, cloudiness, and daylight tendency).

```
// Available activities at the resort
const availableActivities: Activity[] = [
  {
    id: 1,
    name: "Cross-Country Skiing",
    icon: "Snowflake",
    suitability: 0,
    reason: "",
    timeframe: "9:00 - 16:00",
    location: "Northern Trails",
    conditions: {
      minTemp: -20,
      maxTemp: 5,
      maxWindSpeed: 20,
      maxPrecipitation: 50,
      requiresDaylight: true,
    },
  },
  {
    id: 2,
    name: "Sauna Experience",
    icon: "Flame",
    suitability: 0,
    reason: "",
    timeframe: "All day",
    location: "Main Lodge",
    conditions: {
      // Sauna is always good in Finland!
    },
  },
  {
    id: 3,
    name: "Aurora Photography",
    icon: "Camera",
    suitability: 0,
    reason: "",
    timeframe: "22:00 - 01:00",
    location: "Lakeside Viewpoint",
    conditions: {
      maxCloudCover: 30,
      requiresDarkness: true,
    },
  },
  {
    id: 4,
    name: "Nature Hiking",
    icon: "Footprints",
    suitability: 0,
    reason: "",
    timeframe: "10:00 - 15:00",
    location: "Forest Trails",
    conditions: {
      minTemp: -10,
      maxTemp: 25,
      maxWindSpeed: 25,
      maxPrecipitation: 30,
      requiresDaylight: true,
    },
  },
  {
    id: 5,
    name: "Night Snowshoeing",
    icon: "Moon",
    suitability: 0,
    reason: "",
    timeframe: "20:00 - 22:00",
    location: "Moonlight Trail",
    conditions: {
      minTemp: -15,
      maxTemp: 0,
      maxWindSpeed: 15,
      maxPrecipitation: 20,
      requiresDarkness: true,
    },
  },
  {
    id: 6,
    name: "Ice Fishing",
    icon: "Fish",
    suitability: 0,
    reason: "",
    timeframe: "8:00 - 16:00",
    location: "Frozen Lake",
    conditions: {
      minTemp: -20,
      maxTemp: 0,
      maxWindSpeed: 15,
      requiresDaylight: true,
    },
  },
  {
    id: 7,
    name: "Snowmobile Safari",
    icon: "Snowflake",
    suitability: 0,
    reason: "",
    timeframe: "10:00 - 15:00",
    location: "Adventure Center",
    conditions: {
      minTemp: -25,
      maxTemp: 0,
      maxWindSpeed: 20,
      maxPrecipitation: 30,
      requiresDaylight: true,
    },
  },
  {
    id: 8,
    name: "Indoor Crafts Workshop",
    icon: "Scissors",
    suitability: 0,
    reason: "",
    timeframe: "14:00 - 16:00",
    location: "Activity Center",
    conditions: {
      // Indoor activities are good in bad weather
    },
  },
]
```

Figure 43. Available Activity Data at the Resort.

First and foremost, the suitability calculation algorithm is created based on scoring with an initial optimal score of 5 to systematically evaluate multiple environmental readings against the conditions of existing activity programs. Each environmental metric has its own scoring

system. The scaled penalizing technique is utilized in temperature analysis between current weather conditions and activity-specific threshold. Wind speed analysis similarly applies proportional score deduction where it is penalized if it exceeds the maximum speed limit. However, the precipitation evaluation uses a graduated penalty system that deducts points based on how much precipitation surpasses the activity's tolerance. For cloudiness evaluation, the scoring algorithm focuses on visibility-dependent activities like Aurora Photography, penalizing how cloud cover could hinder visibility. The temporal context analysis evaluates how current daylight conditions are correlated to the activity's daylight threshold. Regarding indoor activities (Indoor Crafts Workshop and Sauna Experiences), the inverse relationship with external weather conditions is implemented. The ultimate score is calculated by normalizing within a defined range (1-5).

```
// Function to calculate activity suitability based on weather conditions
function calculateSuitability(activity: Activity, weather: WeatherData): number {
  const current = weather.current
  let suitability = 5 // Start with maximum suitability

  // Check temperature conditions
  if (activity.conditions.minTemp !== undefined && current.temp < activity.conditions.minTemp) {
    suitability -= Math.min(3, Math.ceil((activity.conditions.minTemp - current.temp) / 2))
  }

  if (activity.conditions.maxTemp !== undefined && current.temp > activity.conditions.maxTemp) {
    suitability -= Math.min(3, Math.ceil((current.temp - activity.conditions.maxTemp) / 5))
  }

  // Check wind conditions
  if (activity.conditions.maxWindSpeed !== undefined && current.wind_speed > activity.conditions.maxWindSpeed) {
    suitability -= Math.min(3, Math.ceil((current.wind_speed - activity.conditions.maxWindSpeed) / 5))
  }

  // Check precipitation conditions
  const precipitation = weather.hourly[0].pop * 100 // Convert to percentage
  if (activity.conditions.maxPrecipitation !== undefined && precipitation > activity.conditions.maxPrecipitation) {
    suitability -= Math.min(3, Math.ceil((precipitation - activity.conditions.maxPrecipitation) / 20))
  }

  // Check cloud cover conditions
  if (activity.conditions.maxCloudCover !== undefined && current.clouds > activity.conditions.maxCloudCover) {
    suitability -= Math.min(3, Math.ceil((current.clouds - activity.conditions.maxCloudCover) / 20))
  }

  // Check daylight/darkness requirements
  if (activity.conditions.requiresDaylight && !isDaytime(current.dt)) {
    suitability -= 3
  }

  if (activity.conditions.requiresDarkness && !isNighttime(current.dt)) {
    suitability -= 3
  }

  // Special case for indoor activities - they're better when weather is bad
  if (activity.name === "Indoor Crafts Workshop" || activity.name === "Sauna Experience") {
    if (current.weather[0].main === "Rain" || current.weather[0].main === "Snow" || current.temp < -15) {
      suitability = Math.min(5, suitability + 2)
    }
  }

  // Ensure suitability is between 1 and 5
  return Math.max(1, Math.min(5, suitability))
}
```

Figure 44. Calculating the Suitability of Activities based on Weather Information.

The reasoning generator function adopts the advanced NLP practises by leveraging the structural logic with contextual awareness. Having both positive and negative reasoning

templates facilitate to generate relevant explanation in accordance with activity-specific conditions. For instance, snow conditions are accessed for winter-activity skiing, cloudiness is evaluated for skywatching, and temperature metric is used for thermal comfort programs.

```
// Function to generate reasons based on weather conditions
function generateReason(activity: Activity, weather: WeatherData): string {
  const current = weather.current

  // Default reasons
  const goodReasons = ["Perfect conditions right now", "Ideal weather for this activity", "Highly recommended today"]

  const badReasons = [
    "Weather conditions not ideal",
    "Limited visibility may affect experience",
    "Consider alternative activities",
  ]

  // Activity-specific reasons
  if (activity.name === "Cross-Country Skiing") {
    if (current.weather[0].main === "Snow") return "Fresh snow provides excellent skiing conditions"
    if (current.temp < -5) return "Cold temperatures keeping snow in great condition"
    if (current.temp > 0) return "Snow may be soft due to warmer temperatures"
  }

  if (activity.name === "Aurora Photography") {
    if (current.clouds < 20) return "Clear skies perfect for aurora viewing"
    if (current.clouds > 70) return "Cloud cover may obstruct aurora visibility"
  }

  if (activity.name === "Sauna Experience") {
    if (current.temp < -10) return "Perfect day to warm up in the sauna"
    return "Traditional Finnish relaxation experience"
  }

  // Generic good/bad reasons based on suitability
  return activity.suitability >= 4
    ? goodReasons[Math.floor(Math.random() * goodReasons.length)]
    : badReasons[Math.floor(Math.random() * badReasons.length)]
}
```

Figure 45. Calculating the Suitability of Activities based on Weather Information.

The main recommendation system integrates the above-mentioned functions by applying the suitability calculation method to the activities template and generates relevant reasons for each activity in descending order.

```
// Main function to get recommended activities based on weather data
export function getRecommendedActivities(weather: WeatherData): Activity[] {
  // Calculate suitability for each activity
  const activitiesWithSuitability = availableActivities.map((activity) => {
    const suitability = calculateSuitability(activity, weather)
    const reason = generateReason({ ...activity, suitability }, weather)

    return {
      ...activity,
      suitability,
      reason,
    }
  })

  // Sort by suitability (highest first)
  return activitiesWithSuitability.sort((a, b) => b.suitability - a.suitability)
}
```

Figure 46. Activity Recommendation Algorithm.

The dashboard page serves as the hub for delivering weather activities generated by the recommendation function, which accepts real-time weather data. React's useEffect hook automatically activates the 'fetchCachedWeatherData' function on mount, and the state data stores the results to display on the UI. The 'setInterval' refreshes every 30 minutes, ensuring the most recent information is updated without manual intervention. This interval is removed on component unmount to prevent memory leaks.

```
export default function DashboardPage() {
  const [selectedDate, setSelectedDate] = useState("today")
  const [weatherData, setWeatherData] = useState<WeatherData | undefined>(undefined)
  const [recommendedActivities, setRecommendedActivities] = useState<Activity[]>([])
  const [loading, setLoading] = useState(true)

  // Fetch weather data on component mount
  useEffect(() => {
    async function loadWeatherData() {
      try {
        setLoading(true)
        const data = await fetchCachedWeatherData()
        console.log("Open Weather : ", data)
        setWeatherData(data)

        // Generate activity recommendations based on weather
        const activities = getRecommendedActivities(data)
        setRecommendedActivities(activities)
      } catch (err) {
        console.error("Error loading weather data:", err)
        toast.error("Failed to load weather data. Please try again later.")
      } finally {
        setLoading(false)
      }
    }

    loadWeatherData()

    // Refresh data every 30 minutes
    const intervalId = setInterval(loadWeatherData, 30 * 60 * 1000)

    return () => clearInterval(intervalId)
  }, [])
```

Figure 47. Rakkaranta Dashboard Page.

Following the implementation, the result UI is demonstrated as show in figure below.

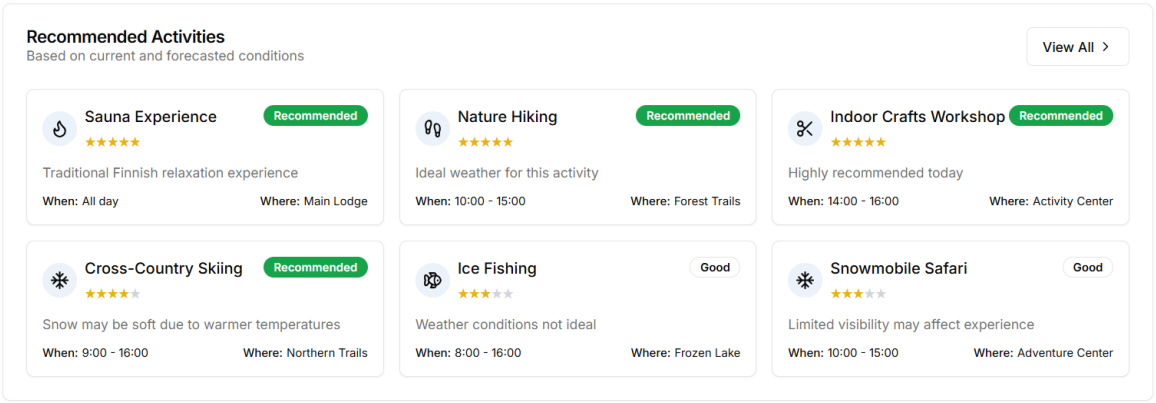


Figure 48. Recommended Activities in Rakkaranta Dashboard Page.

4.6.7 WebSocket Connection Hook

Before implementing the UI components, the TypeScript type is primarily defined as all the categories of collected sensor readings. The system captures air quality metrics (CO, CO2, NO2, PM10, ozone level), some specific gas concentration (methane, propane, hydrogen, ammonia), physical measurements (vibration and light intensity), and resort locations (sauna, lakeside, restaurant, office, woodshed, cottage, firepit). The structure also includes metadata such as timestamp as Date Object, numerical sensor readings, measurement type and sensor category.

Afterwards, a custom React hook named 'useWebSocketData' is implemented to fetch real-time sensor data from the backend via WebSocket connections. Technically, this React hook serves as a bridge between frontend UI rendering and event-driven communication. Multiple state variables are used to manage the return results from the backend service. Moreover, React refs are utilized to store values regardless of re-rendering: one for the WebSocket instance itself, another for queuing messages in case the network is unstable, and a third for caching data.

```
interface DataCache {
    [sensorType: string]: {
        historicalData: SensorData[];
        lastTimestamp: string | null;
    };
}

interface UseWebSocketDataResult {
    sensorData: { [sensorType: string]: SensorData[] | null };
    connectionStatus: 'connecting' | 'connected' | 'disconnected';
    error: Error | null;
    reconnect: () => void;
    sendMessage: (message: string) => void;
    subscribe: (sensorTypes: string[], timeRanges?: { [sensorType: string]: string }) => void;
    updateTimeRange: (sensorType: string, timeRange: string) => void;
}

export function useWebSocketData(): UseWebSocketDataResult {

    const url = process.env.NEXT_PUBLIC_WEBSOCKET_URL as string;
    const [sensorData, setSensorData] = useState<{ [sensorType: string]: SensorData[] | null }>({});
    const [connectionStatus, setConnectionStatus] = useState<'connecting' | 'connected' | 'disconnected'>('connecting');
    const [error, setError] = useState<Error | null>(null);
    const wsRef = useRef<WebSocket | null>(null);
    const messageQueueRef = useRef<string[]>([]);
    const dataCacheRef = useRef<DataCache>({});
```

Figure 49. WebSocket React Hook.

The 'mergeData' method is encapsulated within React's 'useCallback' to preserve integrity between renders. It primarily takes charge of merging the newly received sensor data with the previously cached data in the 'useRef' object. This function deduplicates by filtering out any cached data having the same timestamp as new incoming data so that both types of data can be concatenated and sorted based on last updated time. This design pattern is significantly efficient especially for continuous and time-sensitive data streaming.

```
const mergeData = useCallback((sensorType: string, newData: SensorData[]) => {
    if (!newData || newData.length === 0) return;

    if (!dataCacheRef.current[sensorType]) {
        dataCacheRef.current[sensorType] = {
            historicalData: [],
            lastTimestamp: null
        };
    }

    dataCacheRef.current[sensorType].historicalData = [
        ...dataCacheRef.current[sensorType].historicalData.filter(existing =>
            !newData.some(update => update._time === existing._time)
        ),
        ...newData
    ].sort((a, b) => new Date(a._time).getTime() - new Date(b._time).getTime());

    setSensorData(prevData => ({
        ...prevData,
        [sensorType]: dataCacheRef.current[sensorType].historicalData
    }));
}, []);
```

Figure 50. Merging Data in WebSocket React Hook.

Another method inside this custom webhook is ‘connect’ function using the Observer pattern where the WebSocket broadcasts events to which the React Component subscribes. Once the incoming messages are received, they are parsed and dispatched according to their ‘type’ property to route specific paths. This approach separates the main business logic from the communication layer. The queuing mechanism also solves the decoupling problems in distributed system by queueing the messages. Having messages buffered during unstable connections provides the system more resilient against disconnections.

4.6.8 Real-time Monitoring Dashboard

This UI component creates a complete data visualization dashboard with various chart types to monitor environmental metrics as following Figure 45.

```
export default function ChartsPage() {
    return (
        <>
            <DashboardHeader
                heading="Comprehensive Monitoring Dashboard"
                text="Visualize trends, compare performance, and stay informed with intuitive charts and graphs designed for actionable insights"
            />
            <div className="flex flex-col gap-5">
                <LineChartTemperature />

                <div className="grid grid-cols-1 gap-4 sm:grid-cols-1 2xl:grid-cols-2">
                    <BarChartNoise />
                    <StepChartWaterFlow />
                </div>
                <BarChartWaterLevel />
                <div className="grid grid-cols-1 gap-4 sm:grid-cols-1 2xl:grid-cols-3">
                    <GuageLightIntensity />
                    <GuageVibration />
                    <GuageEmergency />
                </div>
                <div className="grid grid-cols-1 gap-4 sm:grid-cols-1 2xl:grid-cols-2">
                    <BarChartHumidity />
                    <RadialGridGas />
                </div>
                <AreaChartStacked />
                <RadarChartPopulation />
            </div>
        </>
    );
}
```

Figure 51. Rakkaranta Real-time Monitoring Page.

In 'LineChartTemperature', 'BarChartNoise', 'AreaChartStacked', 'StepChartWaterflow', and 'BarChartWaterLevel', the same pattern is used for retrieving sensor data from the backend via the WebSocket hook. Since these data are time-range type sensors, the duration time is included when each component subscribes to its own category.

```
export function LineChartTemperature() {
    const { sensorData, connectionStatus, subscribe, updateTimeRange } = useWebSocketData();
    // Subscribe to multiple sensors
    useEffect(() => {
        subscribe(['temperature'], {
            temperatureSensor: '5m',
        });
    }, []);
```

Figure 52. Temperature Data Subscription via WebSocket Hook.

The Gauge UIs such as 'GaugeLight Intensity', 'GaugeVibration', and 'GaugeEmergency' utilize the 'last' keyword instead of time-range values. This returns an array of the most recent values, and then the latest value among them is selected for the UI.

```
export function GuageLightIntensity() {
    const { sensorData, connectionStatus, subscribe, updateTimeRange } = useWebSocketData();

    useEffect(() => {
        subscribe(['light-intensity'], {
            "light-intensity": 'last',
        });
    }, []);
    const latestReading = sensorData?.['light-intensity']?.[sensorData['light-intensity']?.length - 1] as SensorData;
    const formatIntensityValue = (value: any): string => {
        if (value !== undefined && value !== null) {
            return value.toFixed(1) + ' lux';
        }
        return '0'; // Default to an empty string or another fallback string
    };
```

Figure 53. Light Intensity Data Subscription via WebSocket Hook.

As shown in figure 47, the Guage UIs show latest sensor readings with the optimal values.

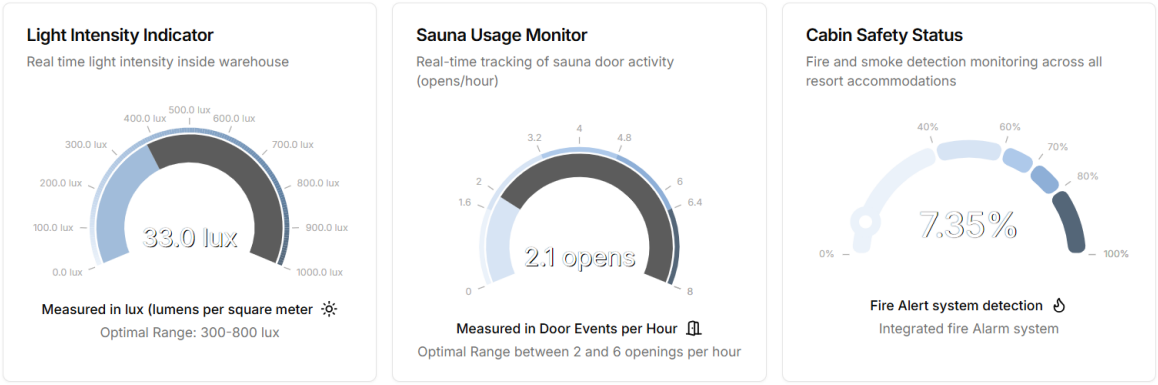


Figure 54. Guage UI in Real-time Monitoring Dashboard.

4.7 DevSecOps

DevOps culture evolved as a practice to address the disparity between the development and operation of software to develop products in iterations and remove bottlenecks from the development process. Security testing is usually done after the fact in the software development lifecycle (SDLC) – in quality assurance or post-deployment phases. This makes security vulnerable with potential security breaches if vulnerabilities are detected too late in production environments. Accordingly, DevSecOps emerged by directly integrating automated security tools such as SCA, SAST, DSAT, and dependency scanners into the development stage based on the "Shifting security left" principle (Eze, 2025).

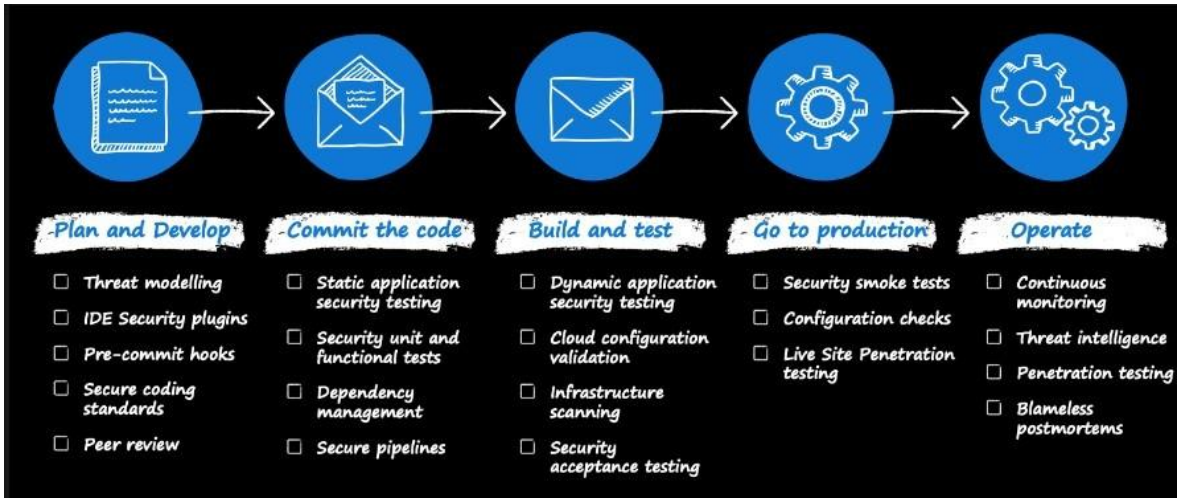


Figure 55. Key Components of DevSecOps Pipeline (Paranjpe, 2023).

Addressing vulnerabilities at the development stage reduces the overload on operation teams and minimizes major code rewrites that could potentially delay the project timeline. However, the 'Shift security left' process should not substitute later-stage testing. Instead, combining early security testing that continues throughout the cycle, known as "Shift security right," provides complete defense (Osnat, 2024).

Before implementing the DevSecOps pipeline for the Rakkaranta project, first create an account on SonarCloud.com and set up the organization and project. SonarCloud is a cloud-based code quality and security analysis service that automatically diagnoses errors, vulnerabilities, and code smells in source code. After successful account creation, a token is generated, copied, and stored securely in the GitHub repository secrets. To prevent simultaneous SonarCloud code analysis, the automatic analysis mode should be disabled in the “Administration” setting.

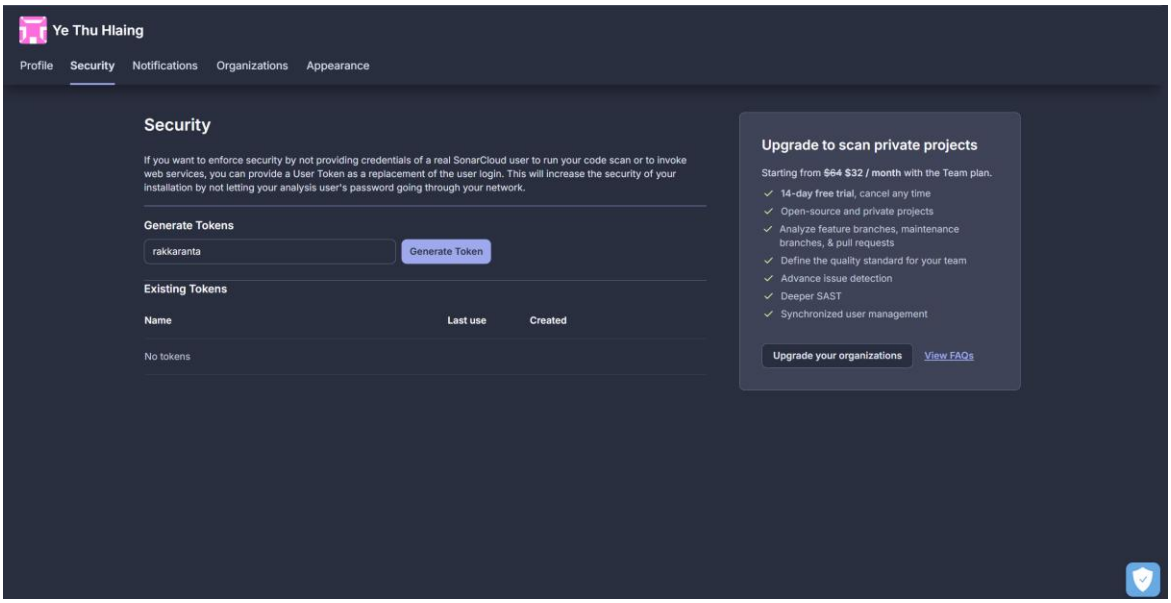


Figure 56. SonarCloud Dashboard.

The Rakkaranta project adopt a comprehensive security scanning strategy in multiple stages from the initial code analysis to post-deployment testing. This workflow is automatically triggered by either pull requests to the main branch or manual dispatch ensuring that the source code undergoes every security measures. The rigorous security testing starts from the codebase using Trivy, OWASP Dependency Vulnerability Checker and SonarCloud for code quality, diagnose bugs and security hotspots.

```yaml
jobs:
  security-scan:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Run Trivy vulnerability scanner for repository
        uses: aquasecurity/trivy-action@master
        with:
          scan-type: 'fs'
          scan-ref: '.'
          format: 'table'
          exit-code: '1'
          ignore-unfixed: true
          severity: 'CRITICAL,HIGH'

      - name: Run OWASP Dependency Check
        uses: dependency-check/Dependency-Check_Action@main
        with:
          project: 'rakkaranta'
          path: '.'
          format: 'HTML'
          out: 'reports'
          args: >
            --failOnCVSS 7
            --enableRetired

      - name: Upload dependency check report
        uses: actions/upload-artifact@v4
        with:
          name: dependency-check-report
          path: reports/

      - name: Run Code Quality Check with SonarCloud
        uses: SonarSource/sonarqube-scan-action@v5.1.0
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
          SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
        with:
          args: >
            -Dsonar.projectKey=${{ secrets.SONAR_PROJECT_KEY }}
            -Dsonar.organization=${{ secrets.SONAR_ORGANIZATION }}
```

Figure 57. Security Scanning in DevSecOps Configuration.

Afterwards, the pipeline undergoes container security analysis before the built image is pushed to the rakkaranta.azurecr.io registry. The infrastructure security validation using Checkov and Gitleaks detects secrets exposure and misconfigurations. The workflow deploys containers to the Azure App Service environment only after all security gates have been passed, with final DAST in post-deployment using OWASP ZAP.

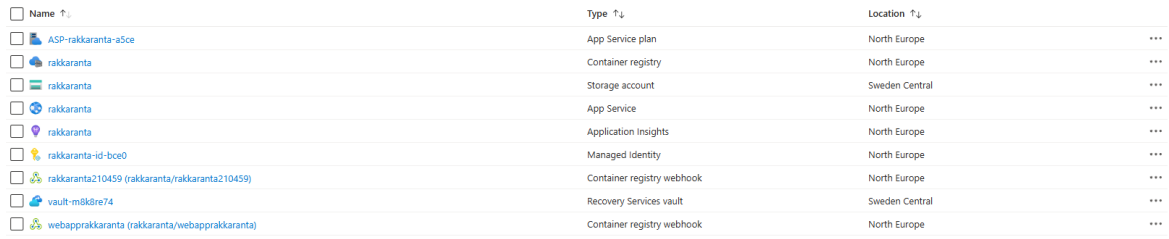
```yaml
security-compliance:
  runs-on: ubuntu-latest
  needs: build-and-push
  steps:
    - uses: actions/checkout@v4

    - name: Setup Checkov
      uses: bridgecrewio/checkov-action@master
      with:
        directory: .
        quiet: true
        soft_fail: false
        framework: dockerfile,kubernetes,secrets
        output_format: cli
        download_external_modules: true

    - name: Scan Infrastructure for Misconfigurations
      uses: bridgecrewio/checkov-action@master
      with:
        directory: ./docker-compose.prod.yml
        quiet: true
        soft_fail: false
        framework: dockerfile,kubernetes,secrets,yaml
        output_format: sarif
        output_file_path: infrastructure-scan.sarif

    - name: Upload Infrastructure Scan Results
      uses: github/codeql-action/upload-sarif@v2
      with:
        sarif_file: infrastructure-scan.sarif

    - name: Secret Scanning
      uses: gitleaks/gitleaks-action@v2
      env:
        GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
        GITLEAKS_LICENSE: ${{ secrets.GITLEAKS_LICENSE }}
```

Figure 58. Security Compliance in DevSecOps Configuration.

4.8 Azure Deployment

The Azure architecture contains - Azure App Service Plan, Azure Container Registry, Azure App Service, and Application Insights. The setup process starts with creating a "rakkaranta" resource group where all the services are grouped. Following this, an Azure Container Registry is created in the North Europe region to store all the Docker images as the centralized repository, promoting versioning and vulnerability scanning.

Before creating the Azure App Service, an App Service Plan named "rakkaranta" in the North Europe region is established to determine the underlying virtual machine infrastructure. The Premium pricing tier (at least P1v2) is recommended to meet the CPU and memory requirements for production level deployment. The core App Service "rakkaranta" is created as a Linux Web app which becomes the main runtime engine handling all the traffic and applying business workflows within containers.



Name	Type	Location
ASP-rakkaranta-a5ce	App Service plan	North Europe
rakkaranta	Container registry	North Europe
rakkaranta	Storage account	Sweden Central
rakkaranta	App Service	North Europe
rakkaranta	Application Insights	North Europe
rakkaranta-id-bce0	Managed Identity	North Europe
rakkaranta210459 (rakkaranta/rakkaranta210459)	Container registry webhook	North Europe
vault-m8k8re74	Recovery Services vault	Sweden Central
webapprakkaranta (rakkaranta/webapprakkaranta)	Container registry webhook	North Europe

Figure 59. Azure App Service Resources.

The following Figure 53 shows the main app service dashboard where all analytics and configuration management can be executed. For health monitoring, there is a specific route implemented on the frontend side, checking memory usage at the production stage.

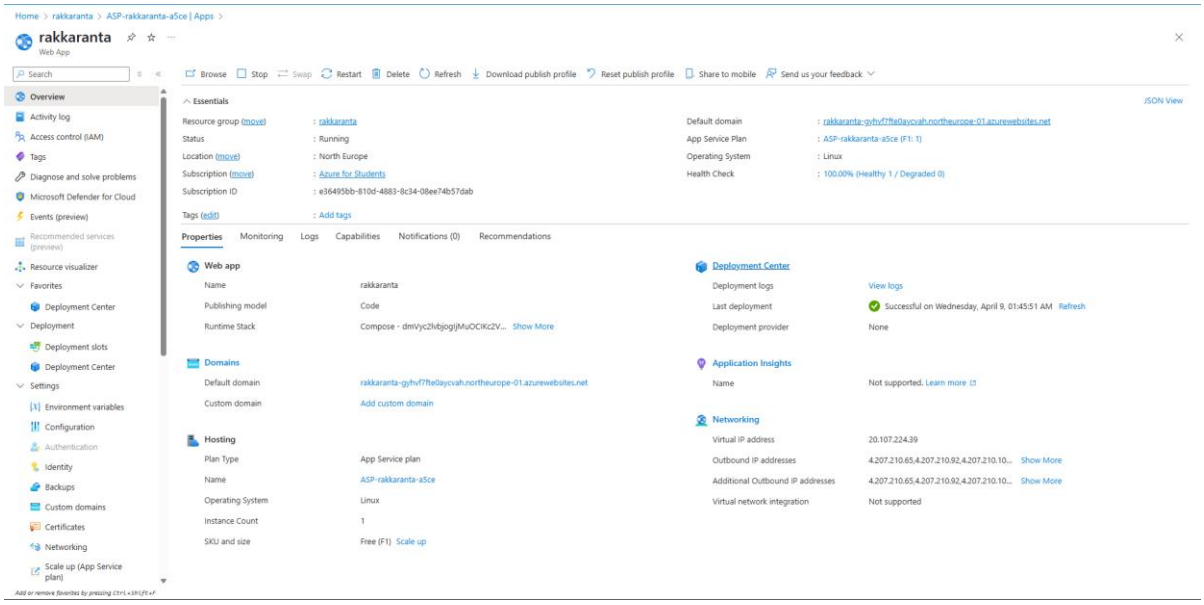


Figure 60. Azure App Service Resources.

In the “Deployment Center” tab, the CI/CD pipeline is configured with Container Registry option using the YAML configuration with Administrator Credential for authentication. The ‘restart:always’ policy makes the service 24/7 available after failures.

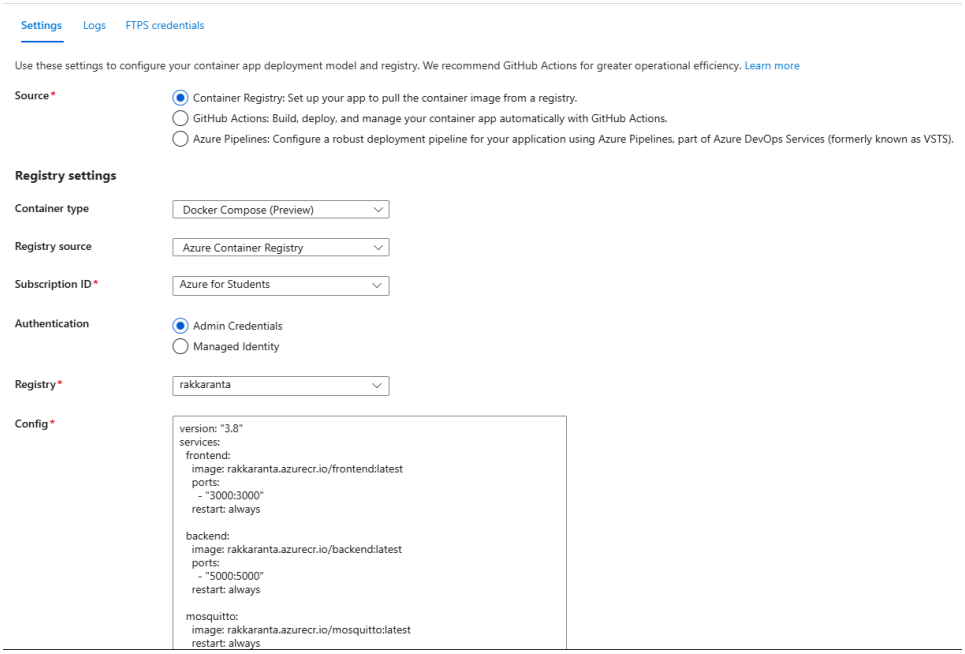


Figure 61. Azure App Service Deployment.

After successful CI/CD deployment, all the docker images can be found on the registry. Moreover, there are container registry webhooks configured to automatically pull the image and orchestrate the deployment process without user manual.

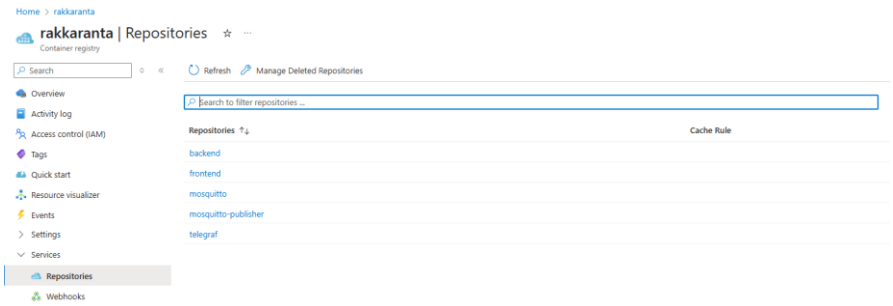


Figure 62. Azure Container Registry.

The production application should be kept continuously running on the platform rather than removing it from memory when no requests are coming.

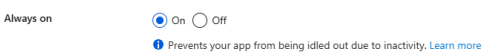


Figure 63. Azure Web App Configuration Setting.

Instead of automatic scaling, custom autoscaling is suitable for the project scope featuring two distinct scaling policies. The main default policy utilizes metric-based autoscaling where the system increases one instance only after CPU usage rises over 80% and memory usage increases over 75%. If CPU percentage drops below 25%, it scales back to the original

default instances. The second policy, “Scale down on Weekends” is a schedule-based scaling to reduce the costs on weekends. On Saturday and Sunday, it is supposed to have low traffic unlike normal weekdays, thus the lower capacity during these off-peak periods is sufficient to maintain performance without the server downtime. This dual policy adopts a predictive scaling approach for dynamic performance cost-efficiently to meet the actual requirements of the platform.

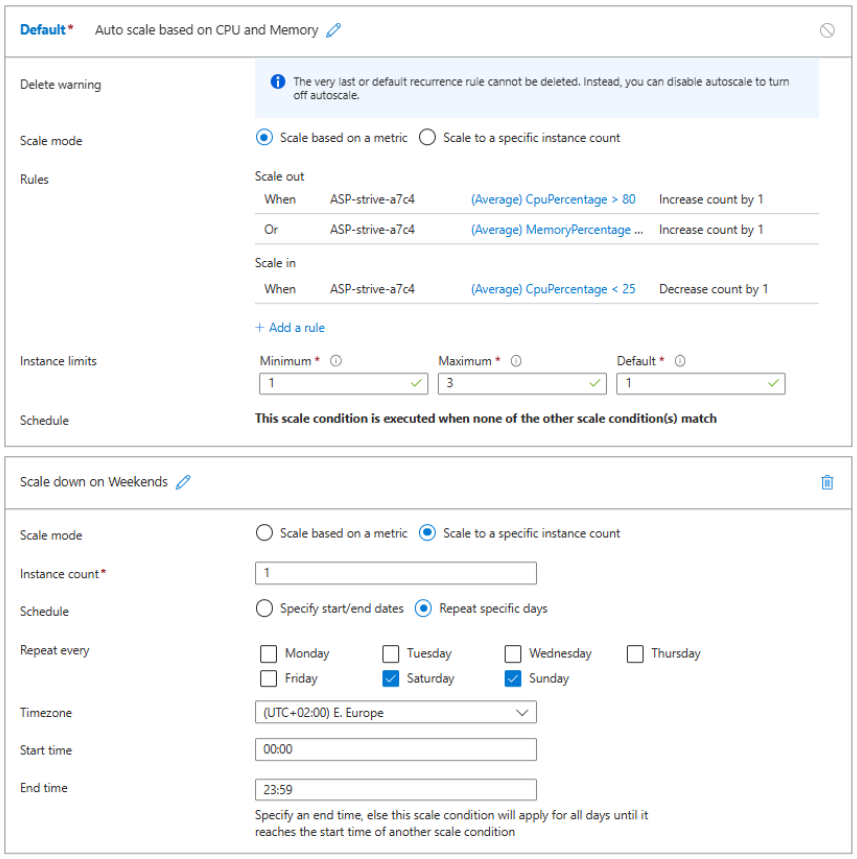


Figure 64. Rule-based Autoscaling for Deployment.

Afterwards, all environmental variables and third-party service secret credentials are stored inside the Azure App Service application. Subsequently, the application becomes operational on the internet.

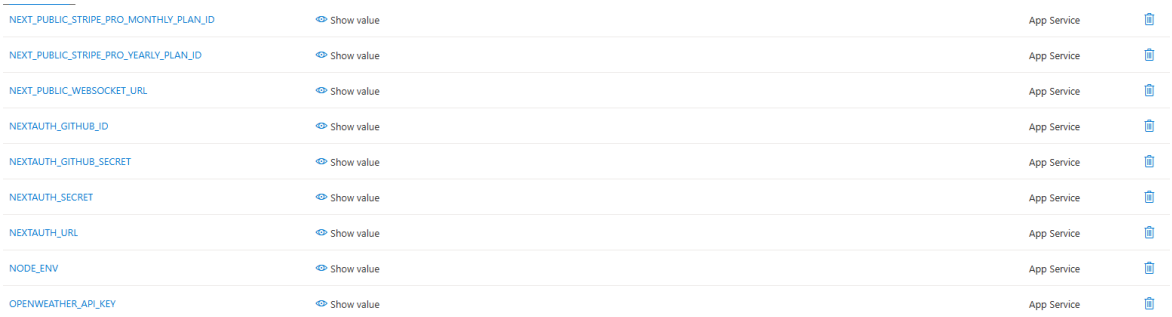


Figure 65. Environmental Variables in Azure.

5 Summary

This practical project-based thesis highlighted the integration of an IoT system aiming for data-driven decision making designed for the Finnish Rakkaranta Resort. It addressed the necessity of a smart management system in hospitality industry through a custom-built platform tailored to the Finnish resort operations. Moreover, it is a well-rounded solution that addresses many priorities: intuitive dashboard for non-technical staff, a robust system during unstable network connections, and scalable architecture for the future development. The adoption of a containerized microservice architecture enforces the modular design and independent deployment of individual components.

The environmental metrics are sent through the centralized MQTT server, and then pulled by Telegraf, mitigating the system overhead compared to the traditional push-based methodology. The time-series InfluxDB takes charge of storing the high-throughput sensor readings while the operational data are stored in Supabase PostgreSQL. The smoother UI/UX is implemented by Next.js features leveraging the client-side and server-side rendering. A separate Node.js server handles the real-time WebSocket architecture for bidirectional communication to the frontend.

The thesis also acknowledges some limitations that needs consideration in a production-level deployment. For the sake of simplicity, the role-based UI implementation is intentionally excluded for different user types such as staff, customers and maintenance personnels. The Google TV integration is deemed redundant for the early development. Although the system utilized a dummy service to generate simulated sensor data without hardware configuration, the architecture has been designed for a straightforward transition to actual sensor implementation at the resort. Moreover, the transition to production deployment with Azure architecture requires detailed cost analysis, comprehensive documentation and operational sustainability perspective.

Despite its limitations and areas for further improvement, this comprehensive IoT system highlights the potential to transform modern technology integration in the hospitality field. The system enhances the value proposition of the resort through its advanced unique features which are adaptable to both operational efficiency and customer experiences in the tourism industry. This implementation builds the robust foundation for digital transformation, providing the energy savings and customer satisfaction through analytic monitoring and real-time responsiveness that position Rakkaranta Resort at the forefront of smart hospitality innovation.

References

Astanakulov, O., BALBA, M.E., Khushvakt, K. and Muslimakhon, S., 2025. IoT Innovations for Transforming the Future of Tourism Industry: Towards Smart Tourism Systems. *Journal of Intelligent Systems and Internet of Things*, *2*, pp.153-53.

Azzedin, F. and Alhazmi, T., 2023. *Secure data distribution architecture in IoT using MQTT*. *Applied Sciences*, 13(4), p.2515. Available at: https://doi.org/10.3390/app13042515 [Accessed 10 Apr. 2025].

EMQ Technologies (2025) 'Why and How MQTT is Used in AI/LLM Applications: Architecture and Use Cases,' *Medium*, 25 March. https://emqx.medium.com/why-and-how-mqtt-is-used-in-ai-llm-applications-architecture-and-use-cases-c8a72cea6cd8 .

Endo, T. (2025) 'Why Supabase became the Go-To Open-Source alternative to Firebase | Medium,' *Medium*, 26 February. https://medium.com/@takafumi.endo/why-supabase-became-the-go-to-open-source-alternative-to-firebase-2d3cd59e7094 .

Eze, E. (2025) *DevOps vs DevSecOps: Key Differences and Best Fit*. https://roadmap.sh/devops/vs-devsecops (Accessed: April 26, 2025).

Garg, S. (2022) 'Azure Hosting Options - FAUN — Developer Community 🐾,' *Medium*, 19 September. https://faun.pub/azure-hosting-options-d4b1f031f37c .

Microsoft, 2024. *Azure App Service Documentation* [online] Microsoft Learn. Available at: https://learn.microsoft.com/en-us/azure/app-service/overview [Accessed 13 Apr. 2025].

Naqvi, S.N.Z., Yfantidou, S. and Zimányi, E., 2017. Time series databases and influxdb. *Studienarbeit, Université Libre de Bruxelles*, *12*, pp.1-44.

Nkenyereye, L. and Jang, J.W., 2016. Performance Evaluation of Node. js for Web Service Gateway in IoT Remote Monitoring Applications. *International Journal of Advanced Culture Technology*, *4*(3), pp.13-19.

Norman, D. (2020) *The Jamstack with Next.js and Prisma*. https://www.prisma.io/blog/jamstack-with-nextjs-prisma-jamstackN3XT (Accessed: May 2, 2025).

Oliveira, G.H.S., 2021. DEVELOPMENT OF A MESSAGE BROKER.

Osnat, R. (2024) *Shift-Left security: what it means, why it matters, and best practices*. https://www.aquasec.com/cloud-native-academy/devsecops/shift-left-devops/ .

Patel, V., 2023. Analyzing the Impact of Next. JS on Site Performance and SEO. *International Journal of Computer Applications Technology and Research*, *12*(10), pp.10-7753

Paranjpe, S.S. (2023) *Introduction to DevSecOps*. https://www.linkedin.com/pulse/introduction-devsecops-sankalp-sandeep-paranjpe (Accessed: May 2, 2025).

Sood, K. (2024) 'What is NoDEJs used for? | NoDEJs use cases,' *Tekki Web Solutions Inc.*, 20 November. https://www.tekkiwebsolutions.com/blog/what-is-nodejs-used-for/amp/ .

Ulili, S. (2024) *Node.Js vs Deno vs Bun: Comparing JavaScript RunTimes | Better Stack Community*. https://betterstack.com/community/guides/scaling-nodejs/nodejs-vs-deno-vs-bun/ (Accessed: May 5, 2025).

Appendix 1: Supplementary Materials

A.1 Source Code Repository

The complete source code for this project is available at:

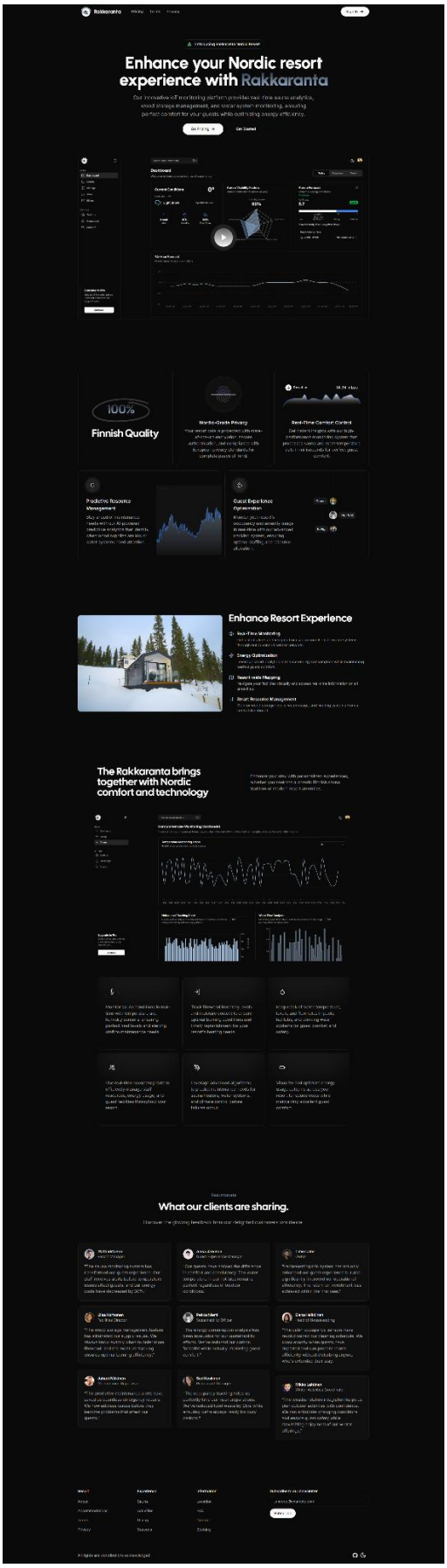
https://github.com/yethuhlaing/Rakkaranta

A.2 Product Demonstration

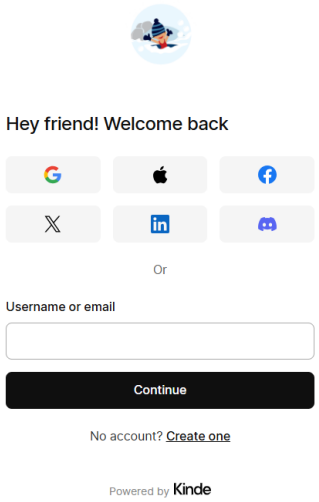
A working demo is accessible online in:

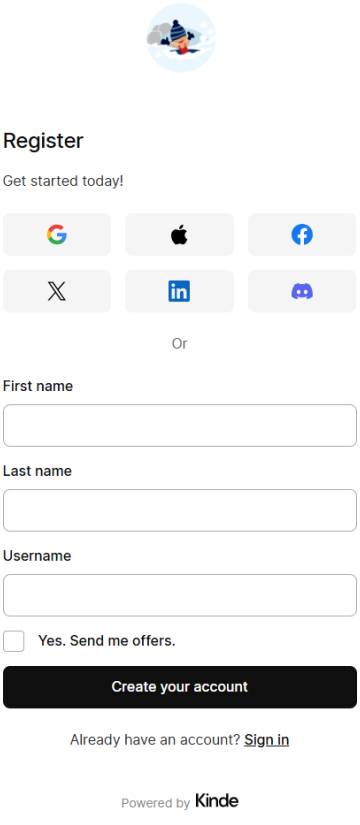
https://www.youtube.com/watch?v=o8TD4SVxuME

Appendix 2. Landing Page

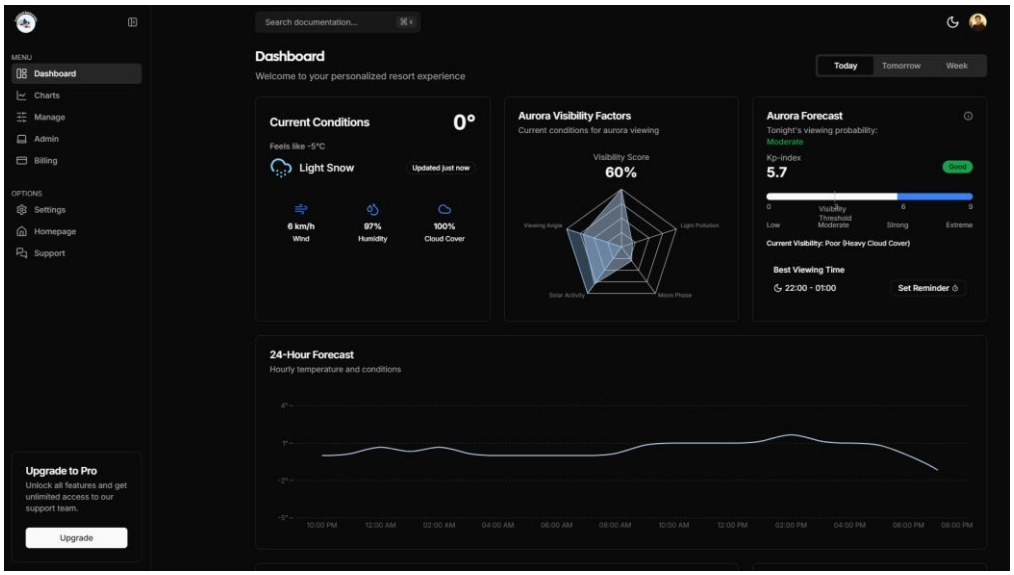


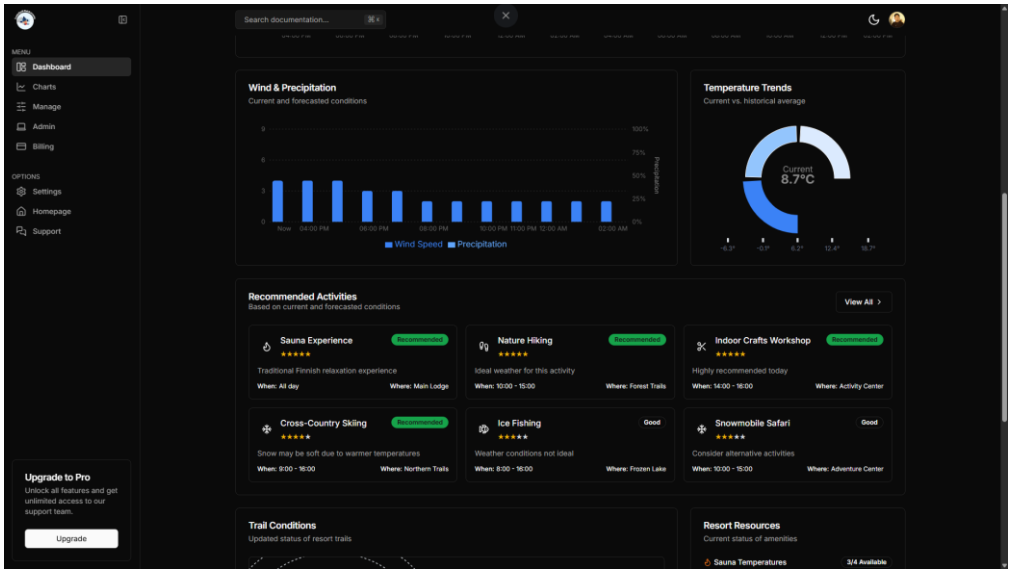
Appendix 3. Authentication Pages

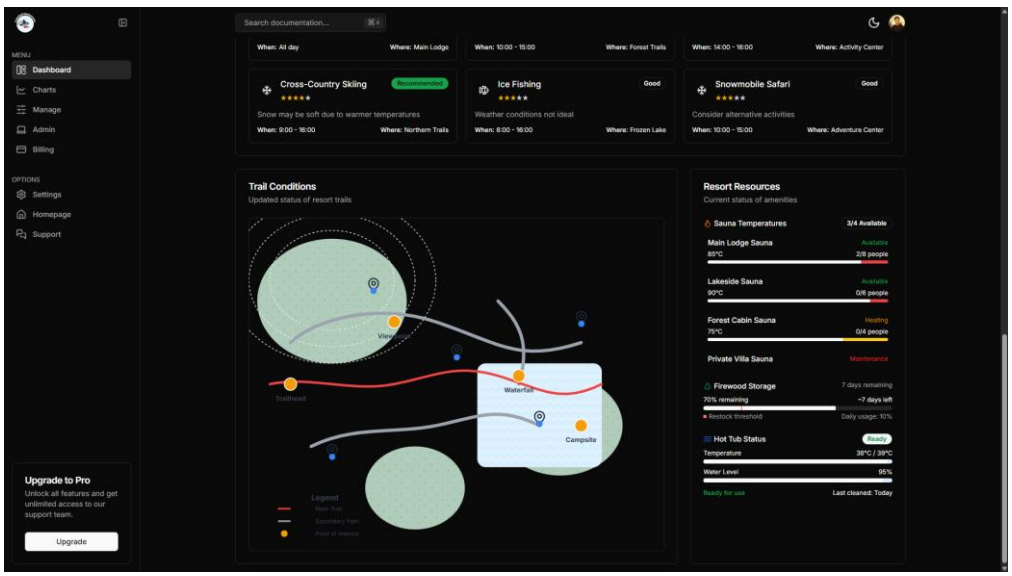




Appendix 4. Real-time Monitoring Dashboard Page







Appendix 5. Analytic Dashboard Page