Energy Consumption Optimizer

Distributed Systems

Salvatore Bennici
salvatore.bennici@studio.unibo.it

Rares Vasiliu
rares.vasiliu@studio.unibo.it

Academic Year 2025/2026

ECO is a distributed, microservices-based web platform for real-time monitoring and optimization of household utility consumption (electricity, gas, water). The system ingests telemetry from network-connected smart hookups, stores measurements in a time-series database, and exposes live dashboards, historical analytics, consumption forecasts, and configurable threshold alerts. Seven independently deployable services, derived from DDD bounded contexts, communicate through a Kafka-backed event-driven pipeline with transactional outbox, inbox deduplication, and dead-letter queues, achieving effectively-once processing and fault-tolerant data propagation. The Monitoring service maintains a Redis cache of measurement tags, hookup metadata, user existence, and zone mappings, kept consistent via Kafka domain events; on broker unavailability the system falls back to direct inter-service HTTP calls, with hookup ID misses always triggering a re-query to prevent measurement loss. The entire stack is containerized and deployed via Docker Compose on a single machine.

Contents

1 Concept

ECO is a distributed web-based platform designed for the monitoring and optimization of household consumption related to electricity, gas, and water. The platform is implemented through integration with intelligent power outlets (smart furniture hookups) deployed across different areas of the home. Thanks to the continuous data acquisition performed by these devices, the platform provides real-time monitoring of utility usage. The system also implements predictive functionalities for estimating future consumption. Finally, the platform allows the configuration of customized consumption thresholds which, supported by an alerting system, automatically send notifications to the user when the defined critical conditions are met.

Smart Furniture Hookups Smart furniture hookups are intelligent power outlets capable of monitoring their own energy consumption and transmitting it over the network via HTTP calls. Each outlet exposes several HTTP endpoints through which it is possible to:

- Retrieve information about the outlet, such as name, status, and type of consumed resource.
- Dynamically configure the destination endpoint to which the outlet must send consumption data.

From a functional standpoint, the outlets perform continuous consumption monitoring with a high sampling frequency (very short time intervals), transmitting the data to the configured remote endpoint via HTTP requests. In addition, in order to simplify the interpretation of state change events, the outlets transmit consumption values equal to zero during power-on and power-off phases.

Finally, these outlets can associate their consumption data with a username tag representing the household user currently using the device.

1.1 Usage scenarios

Users interact with the system through a web browser from personal computers or mobile devices within the household. Administrators configure the system periodically (defining zones, registering hookups, setting thresholds), while household users access consumption data and forecasts on a daily basis.

1. The administrator must be able to configure a house by defining the floor plan, zones, and smart furniture hookups.
2. The platform must display a map with real-time status and utility consumption of smart furniture hookups.
3. The platform must provide real-time data of current utilities consumption.

4. The platform must allow access to historical data of utilities consumption.
5. The platform must notify the administrator when forecasted consumption exceeds thresholds through an automatic alert system.
6. The platform must notify the administrator when defined consumption thresholds are exceeded, through an automatic alert system.
7. The platform must allow filtering of utility consumption by individual users and zones.

1.2 Why distribution is needed

The system is distributed in order to achieve:

- **Distributed Environment:** Smart furniture hookups are physically distributed across different rooms and zones of the house and by design communicate through REST-based HTTP interfaces. Consequently, the system must support distributed data collection and communication over the network. In addition, users and administrators can access the platform from different device/locations, making a distributed web-based system the most suitable solution.
- **Reliability and Fault Tolerance:** The system can be composed of several modules that perform different tasks, such as data collection, visualization, forecasting, and notifications. If a non-critical module fails, the core functionalities of the platform should continue to operate. For example, consumption data can still be collected and displayed even if the forecasting or alerting component is temporarily unavailable. This separation of responsibilities increases the overall reliability of the system by isolating failures to individual components.
- **Scalability:** By decomposing the system into several independent modules, each component can be developed, deployed, and upgraded independently. This modularity simplifies both vertical scaling, allowing core, high-traffic modules to be deployed on more powerful servers, and horizontal scaling, by seamlessly adding more server instances to distribute the load when needed.

2 Requirements Elicitation and Analysis

This section is dedicated to what the software being produced should do, without focusing on implementation details, in terms of functional, non-functional, and implementation requirements.

2.1 Functional Requirements and User Stories

The following functional requirements have been elicited through user stories, grouped by domain operation. Each requirement is numbered and includes an acceptance criterion that will guide the validation phase.

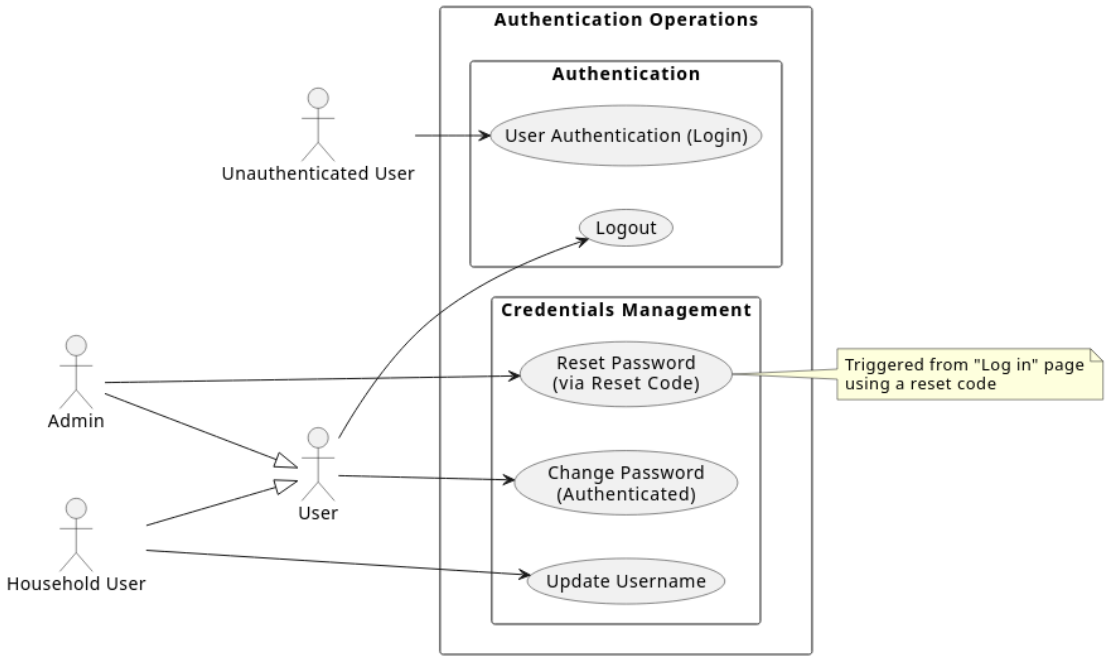

Figure 1: Authentication use case

2.1.1 FR1 – Account Management

The system shall include one default administrator account to manage the platform and allow the administrator to create, update, and delete household user accounts. Household users shall be able to edit their own account information.

- `FR1.1` – Create Household User Account: as an administrator, I want to create new household user accounts, so that they can access the platform and their utility consumption can be tracked. *Acceptance criterion:* a new user account appears in the user list after creation and can be used to log in.
- `FR1.2` – Edit Household User Account: as an administrator, I want to edit existing household user accounts, so that I can update their details when necessary. *Acceptance criterion:* modifications are reflected in the user list and the updated credentials work for login.
- `FR1.3` – Delete Household User Account: as an administrator, I want to delete household user accounts when needed, so that I can stop tracking their utility consumption. *Acceptance criterion:* the deleted user no longer appears in the user list and cannot log in.
- `FR1.4` – Edit Own Account Information: as a household user, I want to edit my account information, so that I can update my personal details. *Acceptance criterion:* changes are persisted and visible upon re-login.

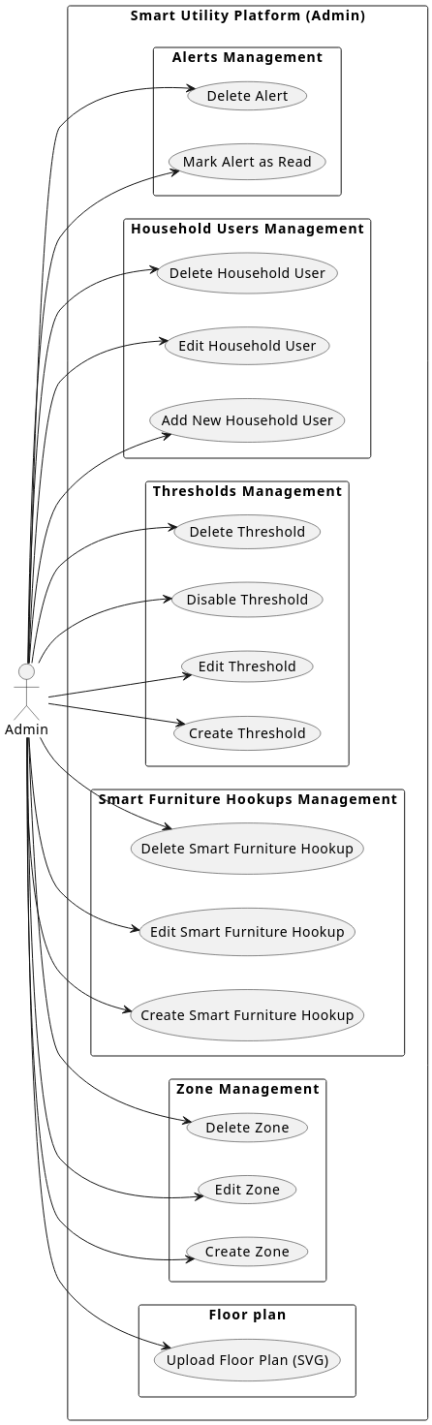

Figure 2: Admin use case

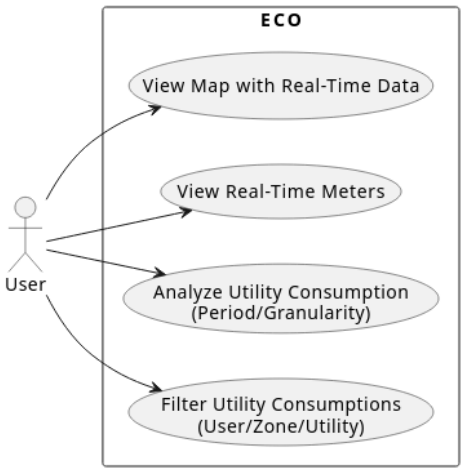

Figure 3: User use case

2.1.2 FR2 – Authentication

The system shall provide secure login and logout mechanisms, password reset capabilities for the administrator, and password update features for regular users.

- `FR2.1` – Administrator Login: as an administrator, I want to log in to the system using pre-configured credentials, so that I can access the administrative features. *Acceptance criterion:* successful authentication redirects to the dashboard; invalid credentials show an error message.
- `FR2.2` – User Login: as a household user, I want to log in to the system with my credentials, so that I can access the platform. *Acceptance criterion:* successful authentication shows the user dashboard with consumption data.
- `FR2.3` – Logout: as a user, I want to log out of the system when I am done, so that my session is securely terminated. *Acceptance criterion:* after logout, protected pages redirect to the login page.
- `FR2.4` – Administrator Password Reset: as an administrator, I want to reset my password if I forget it, so that I can regain access to my account. *Acceptance criterion:* after reset, the new password can be used to authenticate.
- `FR2.5` – User Password Update: as a household user, I want to edit my password, so that I can securely update my login credentials. *Acceptance criterion:* the new password works for subsequent logins; the old password no longer does.

2.1.3 FR3 – Smart Furniture Hookups

The system shall allow the administrator to register, update details, and remove smart furniture hookups.

- `FR3.1` – Register New Hookup: as an administrator, I want to register new smart furniture hookups to zones, so that utility consumption can be tracked. *Acceptance criterion:* the hookup appears on the interactive map in its assigned zone.
- `FR3.2` – Edit Hookup Details: as an administrator, I want to edit existing smart furniture hookups, so that they reflect the current setup. *Acceptance criterion:* edited details are reflected on the map and in hookup metadata.
- `FR3.3` – Delete Hookup: as an administrator, I want to delete smart furniture hookups, so that I can stop tracking them when they are outdated or unused. *Acceptance criterion:* the hookup is removed from the map and its consumption data is no longer ingested.

2.1.4 FR4 – Map and Zone Management

The system shall allow the administrator to upload a digital floor plan, define zones, and position hookups within those zones.

- `FR4.1` – Upload Floor Plan: as an administrator, I want to upload a file of the household floor plan, so that it can serve as the base for the map. *Acceptance criterion:* the uploaded image is displayed as the map background.
- `FR4.2` – Create Zones: as an administrator, I want to create zones on the floor plan, so that I can track consumption by area. *Acceptance criterion:* zones are visible and selectable on the map after creation.
- `FR4.3` – Edit Zones: as an administrator, I want to edit existing zones, so that I can make adjustments when needed. *Acceptance criterion:* zone modifications are immediately reflected on the map.
- `FR4.4` – Delete Zones: as an administrator, I want to delete zones, so that I can remove areas that are no longer relevant. *Acceptance criterion:* deleted zones disappear from the map.
- `FR4.5` – Position Hookup: as an administrator, I want to position a smart furniture hookup within a zone, so that I can accurately associate it with a designated household area for effective utility monitoring. *Acceptance criterion:* the hookup appears at the chosen coordinates within the selected zone on the map.

2.1.5 FR5 – Monitoring

The system shall display a map showing all components, live status, real-time meters, historical data, and computed forecasts.

- `FR5.1` – View Map Overview: as a user, I want to view a map with all zones and smart furniture hookups, so that I can get an overview of the household status. *Acceptance criterion:* the map loads with all configured zones and hookups visible.

- `FR5.2` – View Hookup Status and Consumption: as a user, I want to see the status and consumption of each hookup, so that I know which devices are active and their current supply. *Acceptance criterion:* active hookups display their current consumption and status.
- `FR5.3` – View Real-Time Consumption Metrics: as a user, I want to view real-time utility consumption metrics, so that I can monitor current usage. *Acceptance criterion:* consumption values update automatically as new data arrives.
- `FR5.4` – Filter Consumption: as a user, I want to filter utility consumption by household user and zone, so that I can analyse individual usage patterns. *Acceptance criterion:* only consumption data matching the selected filters is displayed.
- `FR5.5` – Access Historical Consumption Data: as a user, I want to access historical consumption data, so that I can analyse past trends. *Acceptance criterion:* historical data is retrievable for any past time window within the system's retention period.
- `FR5.6` – View Consumption Forecasts: as a user, I want to view forecasts, so that I can plan for future usage. *Acceptance criterion:* forecasts are displayed as charts per utility type, with support for daily, weekly, and monthly aggregation.

2.1.6 FR6 – Thresholds and Alerts

The system shall allow the definition of custom thresholds, notify users upon breaches (actual or forecasted), and maintain an alert history.

- `FR6.1` – Receive Consumption Alerts: as an administrator, I want to receive alerts when consumption exceeds a set threshold, so that I can take corrective action. *Acceptance criterion:* a notification appears in the alerts page within a bounded time window after a threshold breach.
- `FR6.2` – Configure Custom Thresholds: as an administrator, I want to set custom consumption thresholds per utility type, so that alerts match my personal needs. *Acceptance criterion:* thresholds are saved and enforced; breaches above the configured value trigger alerts.
- `FR6.3` – View Alert History: as an administrator, I want to view a history of past alerts, so that I can identify recurring issues. *Acceptance criterion:* past alerts are accessible and can be marked as read or deleted.

2.1.7 FR7 – Device Communication

The system shall interface reliably with physical devices.

- `FR7.1` – Ingest Telemetry Data: as a system, I want to receive and process data from smart furniture hookups, so that users can access accurate utility consumption

information. *Acceptance criterion:* data sent by a hookup is stored and reflected in the monitoring dashboard within the expected latency.

2.2 Non-Functional Requirements

The following constraints and quality attributes must be satisfied to ensure reliable operation within a distributed environment:

- `NFR1` – Scalability and Modularity: the system shall be highly modular to allow independent scaling of components, particularly for the high-throughput ingestion of telemetry data.
- `NFR2` – Fault Tolerance and Resilience: the failure of a single component shall not compromise the availability of critical data ingestion pipelines. The system shall recover automatically from component crashes.
- `NFR3` – Real-Time Responsiveness: the system shall push state changes and telemetry updates to the client with low latency.
- `NFR4` – Observability: given the distributed nature of the system, it shall provide centralised logging, metrics monitoring, and distributed tracing capabilities to facilitate debugging and health assessment.
- `NFR5` – Deployability: the system shall be fully containerized to ensure reproducibility across different environments (development, testing, production).

2.3 Implementation Requirements

As this project is developed for the courses "Applicazioni e Servizi Web" and "Software Process Engineering", the end product 'has the following implementation constraints:

- `IC1` – Tech Stack: Part of the application must be developed using the MEVN stack (MongoDB, Express.js, Vue.js, Node.js) .
- `IC2` – Real time communication: Use of the Socket.IO library to implement real-time communication.
- `IC3` – Multiplatform: The system shall technically involve two or more target platforms.
- `IC4` – Domain-Driven Design: The architecture and design shall strictly follow DDD principles.

2.4 Glossary

The following terminology is drawn from the project's ubiquitous language, established during the domain-driven design process.

Term	Definition
Administrator	A user with full access rights, responsible for configuring and managing the system.
Household user	An individual who resides in the household and uses the platform with read-only access to consumption data.
Utility	A household resource that is consumed (electricity, gas, or water).
Utility consumption	The measured quantity of a utility consumed over a period of time, represented as a time series.
Smart furniture hookup	A network-connected intelligent power outlet integrated into furniture, capable of monitoring its own resource consumption and transmitting readings to a remote service via HTTP.
Floor plan	A digital representation (SVG image) of the physical layout of a household.
Zone	A named section of the floor plan, delimited by polygonal coordinates.
Map	The graphical representation of the floor plan, showing zones and the position of smart furniture hookups.
Consumption threshold	A user-defined upper limit on consumption for a given utility type, used to trigger alerts. A threshold is characterised by a utility type, a numeric value, and a type (`ACTUAL`, `HISTORICAL`, or `FORECAST`).
Forecast	A prediction of future utility consumption computed from historical data, aggregable by period (daily, weekly, monthly).
Alert (or Notification)	A warning delivered to the administrator when a consumption threshold is exceeded, comprising a timestamp, a descriptive message, and a read/unread state.

Table 1: Domain-specific terms from the Ubiquitous language

2.5 Relevant Distributed System Features

We analyse below each distributed system feature from the course syllabus, motivating its relevance to ECO.

Transparency The API gateway provides location transparency via path-based routing, hiding the internal topology from clients.

With service discovery, handled through Docker's internal DNS, developers don't need to manage or track microservice locations and IP addresses manually.

Failure transparency is partially provided by the broker buffering events during consumer downtime.

Fault tolerance and dependability Uninterrupted service is not a strict requirement for this home monitoring system, allowing for moderate recovery times in case of a crash; however, preventing data loss is critical. The database-per-service pattern isolates failures, ensuring a crash in one service cannot corrupt another's data. The system uses an event broker to manage asynchronous communication, buffering events when consumers are down and delivering them upon recovery. Consumers retry processing up to 3 times with exponential backoff and route unprocessable events to dead-letter queues. However, the current deployment uses a single-node replica set and a broker replication factor of 1, meaning a disk failure would cause data loss. A Redis cache is introduced in the Monitoring service to reduce repeated cross-service queries during high-frequency measurement ingestion. The cache is treated as a non-authoritative layer: in case of cache misses or Redis unavailability, the Monitoring service falls back to querying source services, ensuring correctness at the cost of higher latency.

Scalability ECO is architected for horizontal scalability. While the user base is not expected to grow since the current deployment targets a single household, the system is designed to handle an increasing volume of consumption data as new hookup are added over time. Currently, the API gateway distributes HTTP requests across active service replicas. For asynchronous communication, all broker topics use a single partition; this trivially preserves event ordering but strictly serializes consumption through one consumer instance. The current setup sustains the household's traffic, though the architecture has not yet been subjected to stress testing under heavy load. As the number of devices increases, the Redis cache helps absorb read pressure and reduces latency by serving recently used data locally, while event-driven updates keep the cache aligned with CRUD changes from other services.

Security and trust The system processes sensitive household information, including credentials and behavioral consumption telemetry. Authentication uses cryptographically signed tokens (short-lived access, longer-lived refresh) validated centrally at the API gateway, with identity forwarded as headers. Two roles (administrator, household user) enforce access control via service-level middleware. Internal traffic is unencrypted (confined to a private bridge network, but an attacker on any container could eavesdrop).

Resource sharing Services share a single MongoDB cluster (per-service logical databases). The broker coordinates access through competing consumer groups with exclusive partition assignment. A Change Data Capture (CDC) pipeline centralises event publication from all outbox collections.

Openness, interoperability, and heterogeneity. Services span two runtimes (Node.js / TypeScript, Kotlin / JVM) yet interoperate through open standards: REST+JSON,

Kafka, and OpenTelemetry. Hookups communicate over HTTP with a well-defined JSON format.

Evolvability and maintainability Each microservice is independently versioned and deployed. New consumers can subscribe to existing broker topics without modifying producers. A development docker compose override provides hot reload, significantly streamlining the developer workflow.

Performance, concurrency, and communication efficiency. The system mixes synchronous and asynchronous communication by operation type: consumption ingestion and administrative operations use synchronous request-response over HTTP, while others propagates asynchronously through the broker. Furthermore, to minimize network overhead and avoid HTTP polling, real-time updates reach clients via bidirectional push (WebSocket) and unidirectional server-push (SSE) with exponential-backoff reconnection. The Redis cache is maintained using an event-driven invalidation/update strategy. This reduces latency and network overhead during high-frequency measurement ingestion.

3 Design

This chapter explains the strategies used to meet the requirements identified in section 2.

3.1 Architecture

The system relies on a combination of multiple architectural styles to address the requirements of modularity and resilience (NFR1, NFR2).

Microservices Architecture The system adopts a *microservices architecture* deriving them from the bounded context identified during the *Domain-Driven Design* (DDD). Bounded contexts are identified for each core domain—user management, smart hookup registration, map and zone configuration, real-time monitoring, threshold management, forecast computation, and alert notification—and each is implemented as an independently deployable service.

This architectural style is preferred given that the workloads are heterogeneous: high-frequency consumption ingestion demands throughput, while administrative CRUD operations are low-traffic but require strong consistency guarantees.

Clean Architecture Internally, each microservice adopts a *clean architecture*. This approach separates core business logic from technical details, improving maintainability and scalability. The architecture is composed of multiple layers and is based on the *dependency rule*, meaning that outer layers depend on innermost ones but vice versa is not true.

Event-Driven Architecture The system also adopts an *event-driven architecture* to decouple the microservices and ensure high fault tolerance. Instead of relying on synchronous request-response chains, services communicate asynchronously by publishing and consuming meaningful events through an event broker.

This architectural style directly enhances system resilience; if a specific microservice experiences downtime or a crash, the event broker ensures that messages are not lost and can be processed once the service recovers.

3.2 Infrastructure

The system is composed of the following infrastructural components:

- **API Gateway**: single entry point for client requests, performing path-based routing, prefix stripping, and centralised authentication.
- **Microservices**: seven independently deployable services, each exposing a REST API and, where applicable, consuming from and publishing to the message broker.
- **Message Broker**: durable publish-subscribe bus decoupling event producers from consumers.
- **Document Database**: shared, replica-set-enabled database with per-service logical databases for isolation.
- **Time-Series Database**: dedicated database for high-frequency consumption writes and temporal queries.
- **Change-Data-Capture Pipeline**: bridges database-level outbox events to the message broker.
- **Observability Stack**: centralised collection point with separate backends for metrics, logs, and traces, plus a unified dashboard.
- **Web Frontend**: single-page application delivered via the API gateway.

All components are deployed on a single machine within a virtualised network through the use of Docker. The API gateway is the only component reachable from outside the internal service network. Service discovery is achieved through DNS resolution provided by the orchestration layer. A graphical overview of the system can be observed in the Deployment View Diagram, Figure 4.

3.3 Modelling

The application modeling was carried out following the Domain-Driven Design (DDD) domain analysis.

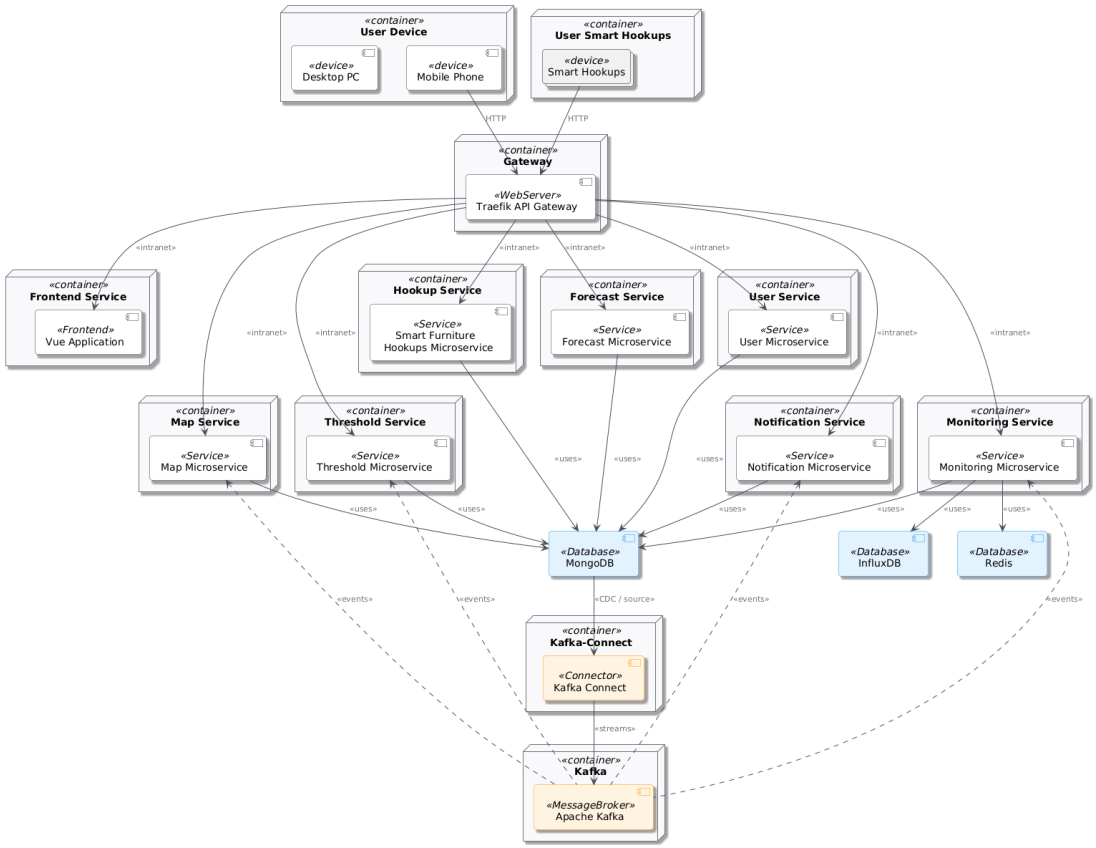

Figure 4: Deployment View Diagram

3.3.1 Domain entities

The domain model comprises the following core entities, each owned by a specific microservice:

- **User** (user service): account with credentials, role, and personal details.
- **Smart Furniture Hookup** (hookup service): hookup with network capability, characterized by name, utility type (electricity, gas, water), operational status, and username tag.
- **Zone** (map service): named area on the floor plan defined by polygon coordinates.
- **Floor Plan** (map service): digital image of the household layout.
- **Consumption Measurement** (monitoring service): time-stamped data point recording consumption of a utility of an hookup, with additional information such as utility type and optional tags such as user id and zone id.
- **Threshold** (threshold service): configurable upper bound on consumption for a utility type, with an enabled/disabled state.
- **Forecast** (forecast service): predicted time series for a utility type over a time window.
- **Notification** (notification service): alert generated on threshold breach, with a read/unread state.

Entities referenced across service boundaries (e.g., a notification references a threshold) are linked by unique identifiers, not by associate foreign keys.

3.3.2 Domain Events

The main domain events exchanged across the system are:

- **User service:** `UserCreated`, `UserDeleted`.
- **Hookup service:** `HookupCreated`, `HookupDeleted`.
- **Map service:** `ZoneDeleted`, `HookupZoneUpdated`.
- **Threshold service:** `ThresholdBreached`.
- **Forecast service:** `ForecastComputed`.

3.4 Interaction

Communication follows three directions, each using specific protocols and patterns.

3.4.1 Communication Mechanisms

- *Frontend → Backend.* The client sends HTTP requests to the API gateway for all the operations. For protected endpoints, the gateway delegates token verification to the user service via `GET /api/internal/auth/verify`. If valid, the gateway injects the caller's identity as headers (`X-User-Id`, `X-User-Role`, `X-User-Username`) and forwards the request to the target backend, as shown in Figure 5. Backend services extract the identity through middleware. Public endpoints bypass authentication.

- *Backend → Backend.* Services communicate mainly asynchronously via Kafka using the Transactional Outbox pattern and Inbox pattern, as shown in Figure 6. Each service writes domain events to a MongoDB outbox; a Kafka Connect CDC connector tails these collections and publishes the events to Kafka topics. Consumers ensure exactly-once processing using an inbox collection for deduplication. Failures trigger a retry policy before being routed to a Dead-Letter Queue (DLQ). Synchronous HTTP communication is also used.

- *Backend → Frontend.* The monitoring service pushes live meter updates via WebSocket. The notification service pushes alerts via SSE. Both channels authenticate the client before opening the connection.

3.4.2 Interaction Patterns

- **Request-Response** Synchronous HTTP. The caller blocks until a response arrives. Used for CRUD operations, gateway-to-user-service authentication, hookup submissions, and forecast-to-monitoring data fetch.

- **Publish-Subscribe** We use Kafka for robust, backend-to-backend event streaming, and WebSockets to extend this pub/sub model to the frontend, allowing clients to subscribe to bidirectional live consumption updates.

- **Push** SSE (Server Side Event) for unidirectional alert notifications to the frontend.

3.5 Behaviour

Each service manages its own local database. Services are **stateful** regarding their databases, but **stateless** regarding incoming HTTP requests, meaning they do not keep memory between requests. This approach makes it easy to replicate instances horizontally.

Each component is the only entity allowed to update its own state. They update the state in response to different triggers:

- **Request-Driven:** The User, Hookup, and Map services update their database strictly when they receive standard HTTP requests from the API gateway.

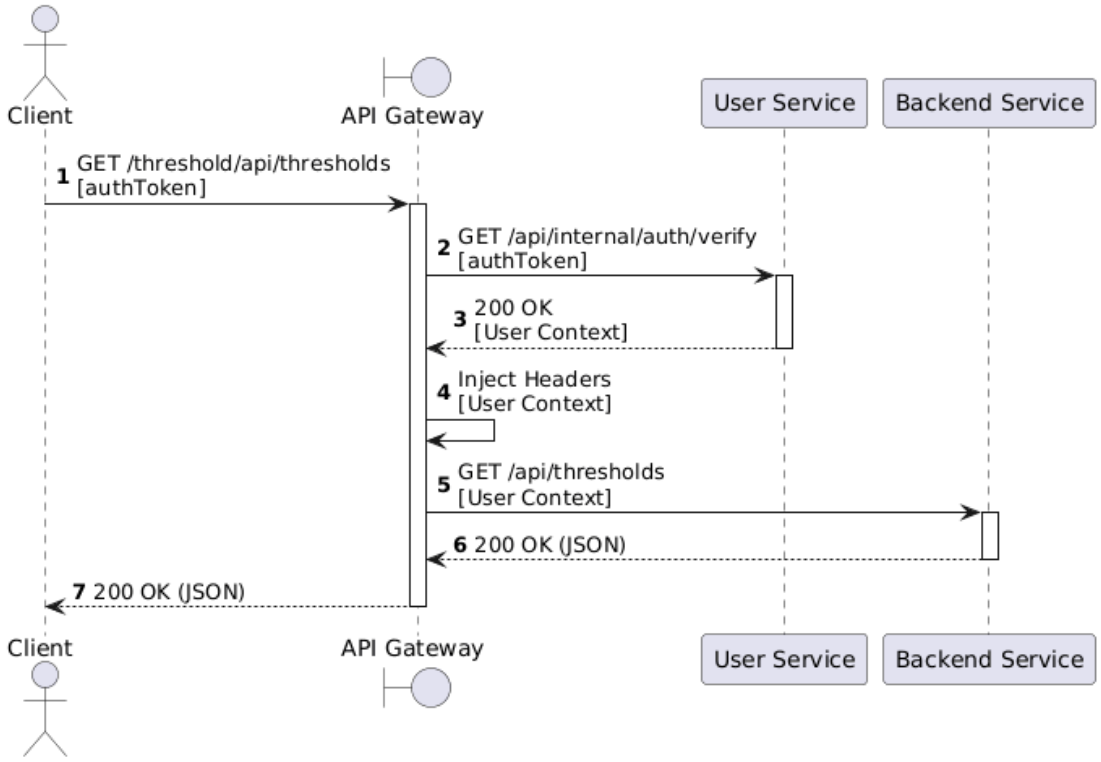

Figure 5: Authentication Sequence Diagram

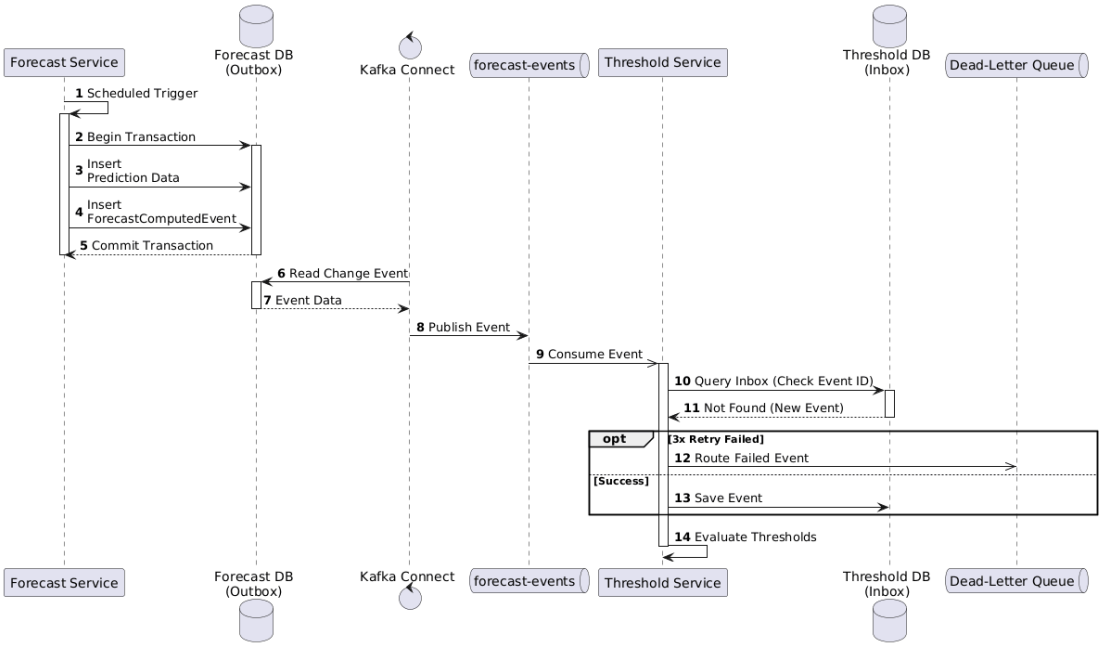

Figure 6: Event Sequence Diagram

- **Event-Driven:** The Threshold and Notification services react to WebSockets and Kafka messages.
- **Time-Driven:** The Forecast service is triggered by a timer. It wakes up on a schedule, fetches historical data, calculates new predictions, saves them, and goes back to being idle.

3.5.1 How the State is Updated

The system mainly relies on an asynchronous event-driven pipeline.

State modifications follow a strict pattern to ensure reliability:

- **Outbox Pattern:** When a service updates its state, it saves the new data and writes a domain event to a local Outbox collection. Both actions happen inside a single transaction, ensuring that they either succeed or fail together.
- **Event Propagation:** A Change Data Capture (CDC) process reads the Outbox and pushes the events to the event broker. This decouples the services, so that the sender does not need to wait for the receiver.
- **Event Deduplication:** Since messages in distributed systems can sometimes be delivered more than once (e.g,. due to retries), consumer services use an Inbox collection. Before processing an event, the service saves its ID in the Inbox. A unique database index rejects duplicate IDs, guaranteeing that the state is updated exactly once per event.

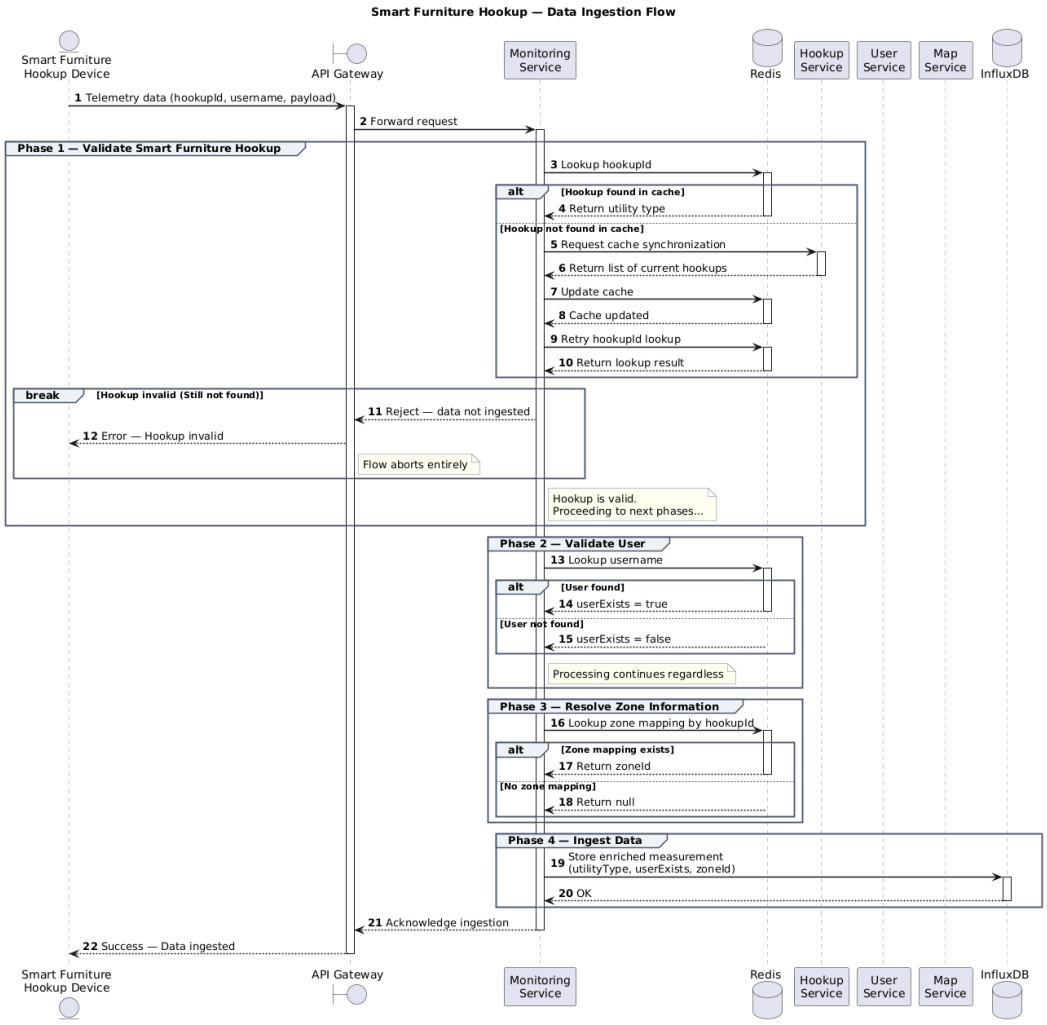

Figure 7: Sequence diagram for ingesting measurements using Redis to resolve tags

3.5.2 How the data are ingested

A Smart Furniture Hookup device sends telemetry data (hookup ID, username, and payload) to an API Gateway, which forwards it to a Monitoring service. The Monitoring service then runs through four validation steps, confirming the hookup exists, checking whether the user is known, resolving any zone the hookup belongs to, and finally enriching the data with those results before storing it in InfluxDB. How those steps are performed depends on the health of the Broker: if it's healthy (Fig.7), all lookups go through Redis as a cache, with a fallback to the Hookup service to resync if needed; if it's unhealthy (Fig.8), each step hits the relevant backend service directly, treating timeouts or failures as graceful fallbacks rather than hard stops. The only exception is for an unrecognized hookup ID, which always causes a rejection in both scenarios.

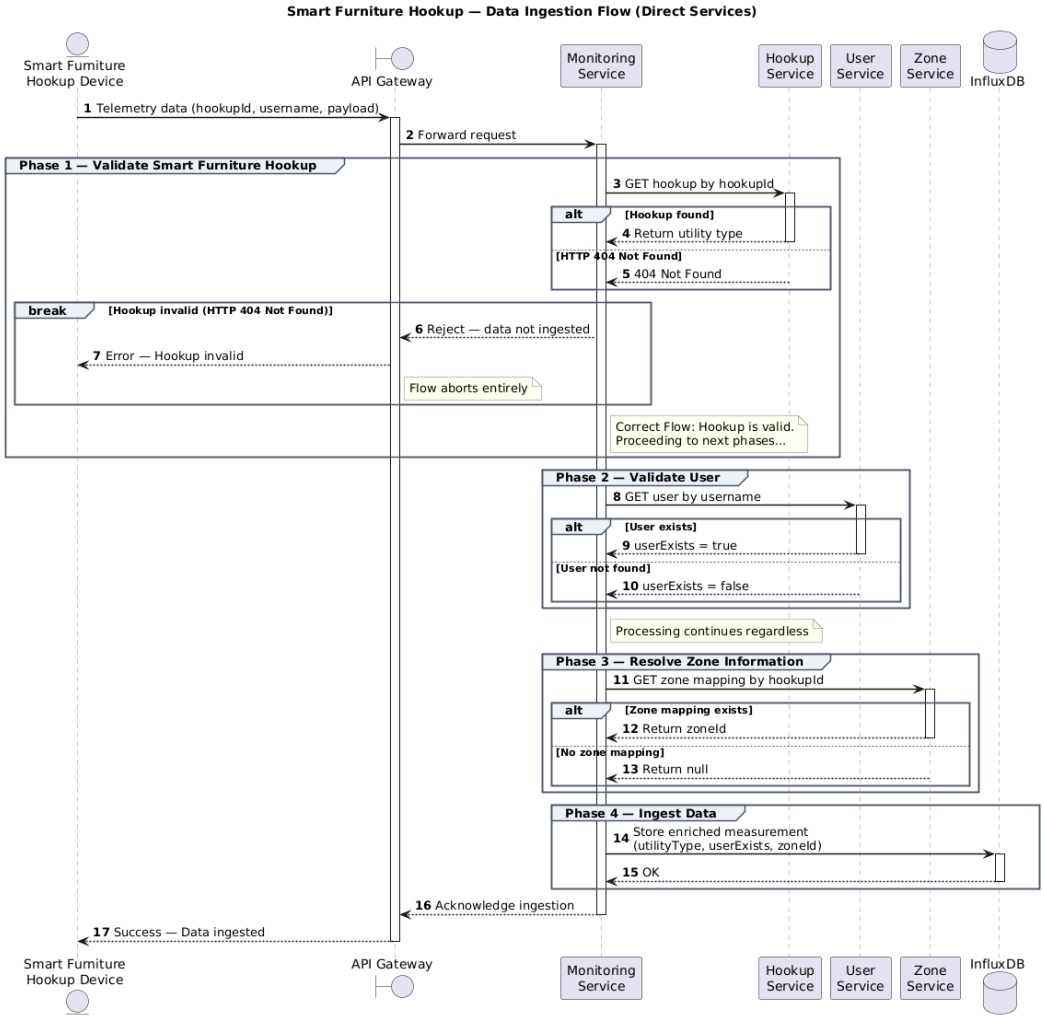

Figure 8: Sequence diagram for ingesting measurements using HTTP requests to resolve tags

3.6 Data and Consistency Issues

The architecture enforces a database-per-service pattern, utilizing two distinct storage technologies tailored to specific data needs:

- **MongoDB (Document-oriented):** Used by most microservices. Bounded contexts are modeled as self-contained documents (e.g., a User embeds all its profile fields), making each aggregate a natural unit of consistency and eliminating the need for complex joins.
- **InfluxDB (Time-series):** Dedicated strictly to the Monitoring service for storing telemetry measurements. It is heavily optimized for the system's high-frequency, append-only write patterns and time-windowed aggregation queries.

Queries and Concurrency Services query exclusively their local databases to serve frontend requests or execute domain logic. To manage concurrent operations and maintain consistency without distributed locks:

- **Write Concurrency:** Domain mutations and their corresponding Outbox events are written using MongoDB sessions within a single, atomic local transaction.
- **Read/Write Idempotency:** Concurrent operations triggered by Kafka message deliveries are managed via an Inbox collection. A unique index on the event ID silently rejects duplicate messages, guaranteeing exactly-once state updates.

Data Sharing Across Components Directly sharing database records is strictly prohibited. Instead, data crosses service boundaries via four specific mechanisms:

- **Kafka Events:** Domain events carry minimal payloads (e.g., aggregate ID, status). Downstream consumers reconstruct their own local views from these payloads without ever querying the producer's database.
- **API Gateway Headers:** The gateway validates JWTs and injects identity headers (e.g., `X-User-Id`) into routed HTTP requests, removing the need for backend services to query the User service for authorization.
- **WebSocket**: The Threshold service receives real-time meter data from the Monitoring service via WebSocket connections, enabling low-latency, continuous data streaming without polling.
- **Synchronous REST:** The Forecast service directly fetches historical data from the Monitoring service via HTTP, as large dataset retrieval is strictly necessary for model training. When the cache system is not available, the Monitoring service directly fetches hookup, user and zone information from the Hookup service, User service, and Map service, respectively.

3.7 Fault-Tolerance

Data replication and sharing Services share state changes by mainly emitting domain events to Kafka rather than relying on synchronous cross-service database queries. Downstream components consume these events to build and persist only the specific data payloads required for their local views, loosely coupling the architecture.

Error handling and transaction safety When a component fails during a state update, the transactional outbox pattern guarantees structural consistency. Both the local domain mutation and the outbox event emission are executed within a single, atomic MongoDB transaction. If a service crashes mid-operation, the database automatically rolls back the entire session, preventing partial states or orphaned events. At runtime, Docker handles service crashes by automatically restarting containers via an `unless-stopped` policy. During these temporary service outages, the Traefik API gateway intercepts incoming HTTP traffic and returns 502 errors to prompt client-side retries, while Kafka safely buffers asynchronous messages. Once connectivity is restored, the Kafka Connect CDC connector relies on internal offsets to resume tailing the database logs from its last checkpoint without data loss.

Heartbeats, retries, and deduplication To mitigate transient network issues and consumer unresponsiveness, the event pipeline implements continuous liveness monitoring and resilient retry mechanisms. Kafka consumers maintain constant heartbeats with the broker; if a consumer hangs, the broker times out the connection and triggers a partition rebalance to reassign workloads among available instances. Processing failures are handled via an explicit consumer retry policy with exponential backoff across three attempts. Events that persistently fail and exhaust all retries are routed to dedicated Dead-Letter Queue (DLQ) topics-such as `threshold-dlq` and `notification-dlq`-preserving them for inspection without blocking the pipeline. Finally, to safely handle duplicate message deliveries caused by network retries, consumer services implement an inbox pattern: a unique database index on the event ID silently rejects duplicate inserts, successfully turning at-least-once delivery into exactly-once processing.

3.8 Availability

3.8.1 Cache

The Monitoring service maintains a cache containing hookup's ID, household user's username, and the zone ID for each hookup on the map. Its purpose is to store these measurement tags and make them quickly available when new consumption data is ingested, thereby reducing network traffic and avoiding repeated cross-service lookups. The cache is kept up to date through the microservices's event-driven architecture by listening for create, update, and delete events. If the broker is unavailable, the system no longer updates the cache in real time. For user and zone tags, the monitoring service trusts the cache on a cache miss. For hookup IDs, however, after a cache miss it queries

the hookup service directly to avoid losing measurements, since measurement ingestion has priority over the accuracy of position and user-related tags.

3.8.2 Network Partitioning

In case of a network partition, the system prioritizes consistency for consumption data, while favoring availability over consistency for tag-related information: if the hookup service is unavailable, measurements cannot be persisted. However, if the map or user services are unreachable, consumption data is still ingested. Other than that, the system remains operational within each reachable partition. Additionally, through the inbox pattern, events are durably stored in the database and processed once the affected services become available again.

3.9 Security

Authentication is centralised at the API gateway: every request to a protected route is intercepted and verified against the user service before reaching any backend. JWT Tokens are stored in HTTP-only cookies, preventing client-side scripts from accessing them. On success the gateway injects identity headers (`X-User-Id`, `X-User-Role`, `X-User-Username`); on failure it returns 401 directly. This keeps raw tokens out of backend services entirely. Public endpoints - login, registration, password reset - bypass the check, and the frontend refreshes expired tokens transparently.

Once identity is established, authorization is enforced at each service by middleware that reads the gateway-injected headers. The model is a simple two-role RBAC: ADMIN has full privileges across all services, while the household user is limited to read-only consumption views and self-profile editing, without access to alerts, thresholds, or administrative functions. No per-resource isolation is implemented - a household user sees all the other household consumption data.

Underpinning both layers, passwords are hashed with bcrypt (Blowfish-based key derivation, 10 salt rounds); access tokens are signed with HMAC-SHA256; and the administrator password reset is gated by a pre-shared secret. Internal traffic on the Docker bridge is not encrypted, which is acceptable in the current single-machine deployment. For inter-service communication and encryption are not implemented and are discussed in section 9.

4 Implementation

Microservices Six backend services are written in TypeScript on Node.js with Express.js (user, threshold, notification, monitoring, hookup, and map); one runs on the JVM in Kotlin with Ktor (forecast). The frontend is a Vue.js application bundled with Vite.

Communication Four distinct protocols handle all network traffic, utilizing JSON as the standard data payload format. HTTP facilitates synchronous REST calls routed

through the API gateway; WebSockets (Socket.io) provide bidirectional transport for live meter updates; Server-Sent Events (SSE) stream alert notifications to the admin frontend; and the Apache Kafka wire protocol manages asynchronous event distribution between backend services.

Authentication JSON Web Tokens signed with HMAC-SHA256 are stored in HTTP-only cookies. Access tokens expire after 1 h, refresh tokens after 7 d. The API gateway verifies every token via Traefik's ForwardAuth middleware, injecting identity and role headers downstream so backend services never handle raw tokens. A two-role RBAC model is enforced at each service; the administrator password reset is gated by a pre-shared secret.

Deployment The entire stack runs in Docker, orchestrated by Docker Compose: a single declarative file defines every service, network, volume, health check, and environment variable. Service discovery uses Docker's internal DNS; only the API gateway is exposed externally.

Messaging Apache Kafka, running in KRaft mode, serves as the message broker. The transactional outbox pattern uses a MongoDB Kafka Connect source connector: each service writes domain events to its outbox collection, and the connector captures them through CDC. Dead-letter queues capture events that exhaust all retries.

Observability OpenTelemetry collects traces, metrics, and logs from all services via a central Collector. These signals are exported to purpose-built backends: Prometheus for metrics, Loki for log aggregation, and Tempo for distributed tracing. Grafana then serves as the unified user interface to visualize and correlate this telemetry data. At the application layer, Node.js services rely on auto-instrumentation, while Kotlin services incorporate manual spans for specific business logic. To maintain visibility across asynchronous boundaries, the W3C Trace Context is propagated through Kafka headers, ensuring unbroken end-to-end tracing.

4.1 Implementation Challenges

Automating the Kafka Connect setup in Docker Compose presented a unique challenge: Compose is great at orchestrating container boot sequences, but it cannot make the REST API calls needed to actually configure the connectors. To solve this, we implemented a "one-shot" initialization container. This lightweight container acts as a bridge-it polls the Kafka Connect endpoint until the service is fully booted, injects our connector configurations via a POST request, and then exits once the setup is complete.

5 Validation

5.1 Automatic Testing

All backend services follow the testing pyramid: unit, integration, and component tests, supplemented by end-to-end tests at the system level. Unit tests isolate business logic with mocked dependencies, covering entities, services, RESTful controllers, and so on, using Vitest for TypeScript services and Kotest for Kotlin ones. Integration tests are used for example to verify persistence against a real MongoDB instance, confirming the transactional outbox (atomicity of domain write and event emission) and inbox deduplication (duplicate events silently rejected). Component tests boot each service with its own HTTP server and database, exercising complete request-to-response flows - authentication, notification lifecycle, forecast computation, and hookup CRUD - without external dependencies. The point is to fully test the microservice by treating it like a black-box.

A Playwright end-to-end suite runs against a fully deployed Docker Compose instance, covering the critical paths. The key cross-service scenario verifies eventual consistency: for example, after a forecast triggers a threshold breach, the test polls the notification API until the alert appears, confirming the end-to-end event chain completes within a bounded window.

5.2 Acceptance Testing

Beyond automated testing, every system use case was manually validated on the deployed instance to confirm the overall end-to-end results. A key focus of this manual testing was real-time data propagation via WebSockets, where a scripted hookup emitted consumption data to verify live chart updates on the frontend. Additionally, the complete map and zone configuration workflow-spanning floor plan uploads, polygon drawing, hookup placement, and point-in-polygon assignments-was manually executed to ensure perfect coordination between the underlying microservices.

6 Deployment

The entire system is deployed via Docker Compose. The only prerequisites are Docker Engine (version 24+) and Docker Compose (version 2+). No additional requirements are necessary.

Starting from scratch requires three steps:

1. **Clone the main repository** to your local machine.
2. **Configure the environment.** Create a `.env` file in the project root using the provided `.env-example` template. The following mandatory secrets have no defaults and require explicit values:
   - `JWT_SECRET_KEY` - Cryptographic key for signing and verifying JWTs.

- `RESET_CODE` - Pre-shared secret for administrator password resets.
- `DOCKER_INFLUXDB_INIT_USERNAME` - InfluxDB administrator username.
- `DOCKER_INFLUXDB_INIT_PASSWORD` - InfluxDB administrator password.
- `DOCKER_INFLUXDB_INIT_ADMIN_TOKEN` - InfluxDB API access token.

There are also some optional variables with preconfigured defaults to customize token expiration, the forecast schedule, and other aspects of the application.

3. **Launch the stack** by running `docker compose up`.

The expected outcome is 14 running containers, all reporting as healthy, with the frontend application accessible at `http://localhost`.

6.1 Development Cycle

For local development with hot reload, the project includes a compose override that builds images from local source trees (services repo must be placed under the same parent folder) and enables Compose Watch for automatic synchronisation of code changes. Development ports for databases and the message broker are exposed to the host machine for direct inspection. A Makefile provides targets for common operations: full stack, infrastructure-only, development mode, log following, and teardown with optional volume removal.

7 User Guide

The platform is accessed via browser at `http://localhost` with a pre-configured administrator account (username: `admin`, password: `admin`).

On first access, the administrator follows the guided onboarding: upload the floor plan, draw zones on it, register smart furniture hookups with their endpoint URLs and positions, and optionally create household user accounts and configure consumption thresholds. This setup is performed once; everything can be adjusted afterwards.

The dashboard displays the interactive map with live hookup statuses, real-time consumption meters, and historical charts filterable by user, zone, and utility type.

The forecasts page shows consumption predictions per utility type, aggregated daily, weekly, or monthly, computed automatically every day.

Dedicated pages let the administrator create, edit, enable, disable, and delete thresholds per utility type, and create, edit, or delete household user accounts.

When a threshold is breached a notification appears in real-time. Notification can be marked as read and can be deleted; duplicate alerts for the same real-time threshold within one hour are suppressed.

The household user, at first logs in with credentials provided by the administrator. They can update their credentials later from the account settings page. Their experience is read-only: they can view the interactive map, real-time consumption, historical and forecasts consumption charts, applying the same filters by user, zone, and utility type. They have no access to alerts, thresholds, hookup registration, or user management.

8 Self-evaluation

8.1 Salvatore Bennici

I developed the threshold, forecast, and notification services, and contributed to the user service. My primary responsibility was designing the system's event-driven backbone. This included implementing a transactional outbox pattern, a Kafka CDC pipeline, inbox deduplication with Dead-Letter Queue (DLQ) routing, and a comprehensive observability stack. Additionally, I authored the Docker Compose configurations for both production and development environments.

What I am happiest with is the reliability of the event pipeline through the adoption of both the outbox and the inbox pattern. The observability stack - Prometheus, Loki, Tempo, Grafana, all fed by OpenTelemetry - could use a more polished UI, but it trace everything across all the services.

If I were to redo the project, I would invest more time in documentation from the beginning, especially around service contracts. I would also work harder to reduce code duplication. Many services share the same patterns - outbox, inbox, retry logic, shared utility code, and so on - copy-pasted across the codebase, creating multiple sources of truth. Extracting a shared library early on would have kept the codebase leaner and made changes safer. I would also work on further improving the app's security, as there is still room for enhancement.

8.2 Rares Vasiliu

I developed the monitoring, map, and hookup services, and contributed to the user service. My main responsibility was designing the ingestion and query system for measurements. This included both information retrieval through the event-driven cache and the Socket.IO connection between the threshold service and the frontend.

I am especially happy with how the query system works and communicates with external systems. It uses clean logic for filters and WebSocket events, along with real-time synchronization on the dashboard, which allows the service to run queries that do not depend on filters and execute only once, while tracking active clients and performing queries only when at least one client is connected.

The cache works well, but it could be improved by adding TTL-based data fallback in case of broker failure, so the current logic can still be reused.

The main weakness is how hookups are handled. If I had more time, I would build logic to track active hookups using the last consumption message received, in order to detect possible crashes. Since I also developed the smart furniture hookup simulator (*Waves Lab*), I assigned responsibility for idle and stop-consuming conditions to the physical smart hookup instead of implementing that monitoring logic from the beginning.

Another weakness I would like to improve concerns data consistency when optional tags cannot be retrieved after a network partition. A more interesting approach could be to take event ordering into account and update measurement tags by adding or removing them once the connection is re-established, in order to achieve eventual consistency.

Unfortunately, the core monitoring system was very complex, since it uses three types of communication (REST, sockets, and events) and multiple persistence databases, including InfluxDB, MongoDB, and Redis. Developing it and the related services consumed almost all of the available development time. Nevertheless, the overall development experience was extremely rewarding and enjoyable.

9 Future works

- **Introduce a fallback cache layer for measurement tag resolution**: implement a caching mechanism that can be used when the broker is unavailable. This would reduce the number of requests required to resolve measurement tags and improve system resilience during broker downtime.
- **Improve the connection management of physical hookups**: In the current implementation, if the Hookup service fails during the creation of a hookup after already providing the hookup ID to the physical device, the device may start sending data to the Monitoring service even though the hookup was not successfully configured in the Hookup service. This results in unnecessary network traffic and processing overhead. To address this, we could:
  - Check for pending hookups at Hookup service startup and retry registration.
  - Implement a blacklist on the Monitoring service to reject data from invalid or non-existent hookups.

  It should be noted that the Monitoring service cannot directly disconnect devices, as doing so could introduce race conditions with the Hookup Service during subsequent connection attempts.
- **Develop a hookup management daemon**: Create a daemon responsible for tracking hookup lifecycle events and operational states, including:
  - Detection of newly connected consumers.
  - Monitoring of the latest consumption activity.
  - Crash detection to stop tracking consumptions.
- **Introduce event-driven eventual consistency for measurement tags**: Currently, measurements require all related services to be available in order to resolve metadata such as Hookup IDs, Zone IDs, and user information. When one of these services is unavailable, measurements may be delayed or discarded.

  A possible improvement is to decouple measurement ingestion from tag resolution. Measurements could be temporarily stored with unresolved metadata and processed later by a background mechanism. The system could leverage existing domain events, such as hookup creation, zone assignments, or user lifecycle events, to reconstruct the metadata associated with a measurement. By correlating the

measurement timestamp with the historical sequence of events, the correct metadata could be determined based on the system state at the time the measurement was generated. This approach would increase resilience to service outages, reduce data loss, and enable accurate metadata reconstruction even when dependent services are temporarily unavailable.

- **Monitoring service replication**: The Monitoring service could be replicated to support a larger number of connected devices and increase overall system availability. Multiple instances of the service could be deployed behind a load balancer, allowing incoming traffic to be distributed across replicas and preventing a single instance from becoming a bottleneck.

  A key challenge of this approach is the management of WebSocket connections used to deliver real-time data to frontend applications. Since clients may be connected to different Monitoring Service instances, broadcast messages must be propagated across all replicas to ensure consistent real-time updates. This issue can be addressed by leveraging the Socket.IO Redis adapter, which enables the distribution of events between instances through Redis. By sharing WebSocket events across all replicas, frontend clients can receive real-time updates regardless of the instance to which they are connected.

- **Internal endpoint security:** Internal endpoints currently lack authentication and authorization checks, making them implicitly trusted. A future improvement is to introduce service-to-service security mechanisms and enforce network-level restrictions to ensure only trusted components can access internal APIs.