# Energy Consumption Optimizer (ECO)

Rares Vasiliu

Distributed Systems

Salvatore Bennici

# Project Overview

A web-based platform designed for the monitoring and optimization of household consumption related to electricity, gas, and water.

## Intelligent Hookups

HTTP-enabled hookups providing continuous high-frequency sampling and real-time status updates via remote endpoints.

## Predictive & Real-Time Monitoring

Enables interactive map views, historical data access, and predictive forecasting for future utility consumption.

## Alert System

Automated notifications triggered when usage exceeds custom thresholds, ensuring efficient household management.

# Smart Furniture Hookups

## HTTP Endpoints

Each hookup exposes REST endpoints to:

- Retrieve its information
- Configure the destination endpoint

## Consumption Monitoring

The hookups perform continuous consumption monitoring with a high sampling frequency, transmitting the data to the configured remote endpoint via HTTP requests.

## User Tagging

The hookups can associate their consumption data with a username tag representing the household user currently using the hookup

# Why Distribution?

## Distributed Environment

Smart furniture hookups are physically scattered across the house and natively communicate via REST HTTP.

## Reliability and Fault Tolerance

Modular separation ensures that core data collection remains active even if non-critical components like forecasting or alerts face temporary downtime.

## Scalability

Independent module decomposition allows for horizontal and vertical scaling, accommodating growing data traffic and user demands.

# Domain Analysis

## Requirements

- User login and management
- Hookup management
- Interactive map
- Consumptions forecasting
- Alert notification
- Consumption monitoring

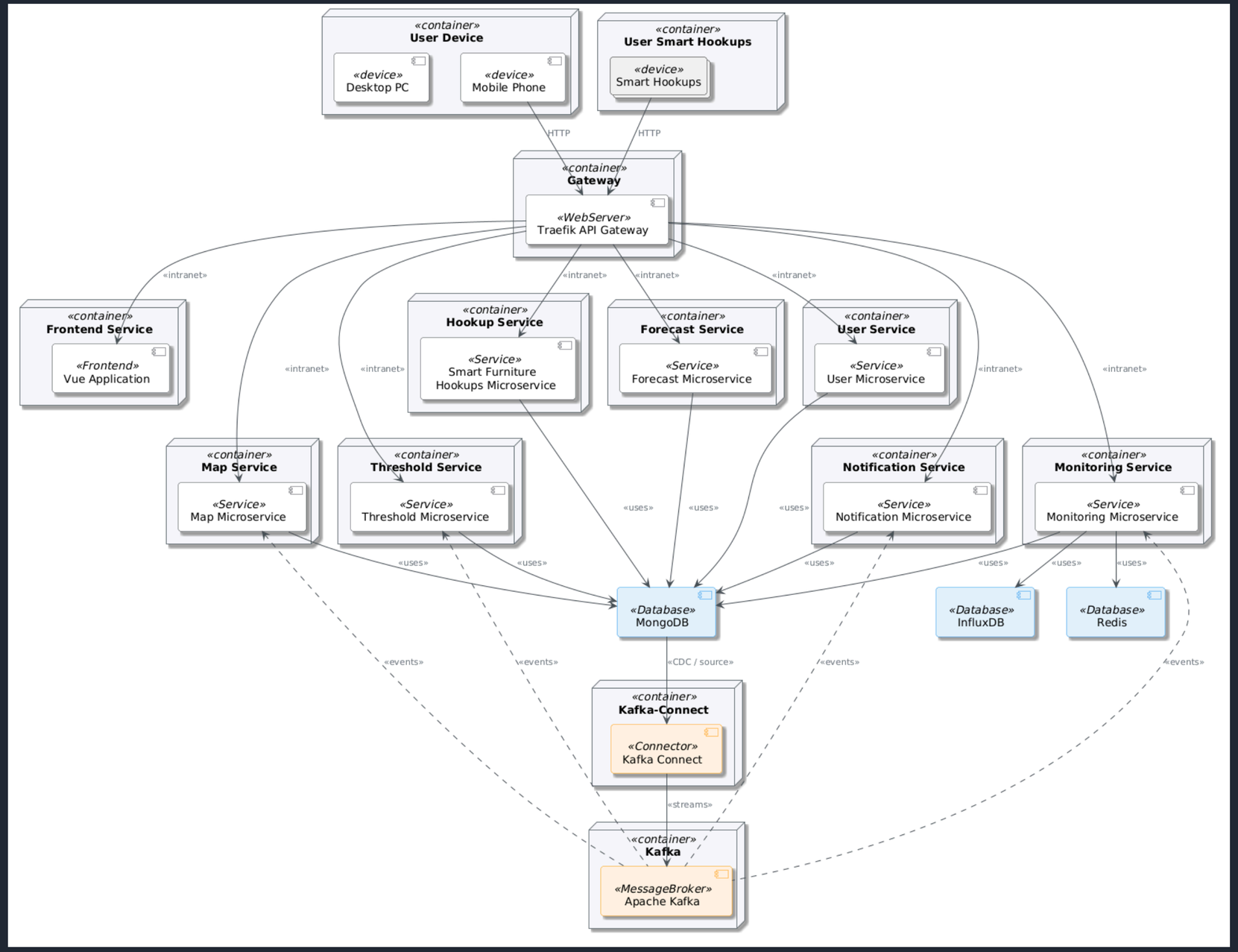
## Bounded contexts

For each context we defined:

- Domain entities
- Domain events
- Interactions with other contexts

# Main Deployment View

- API Gateway
- Microservices
- Message Broker
- Databases
- Frontend
- User’s Hookups



# Communication

## Synchronous Request/ Response

Implemented through HTTP RESTful API:

- Frontend → Backend: The client sends HTTP requests to the API gateway for all operations.
- Backend → Backend: Used for strict Request-Response interactions, such as read operations

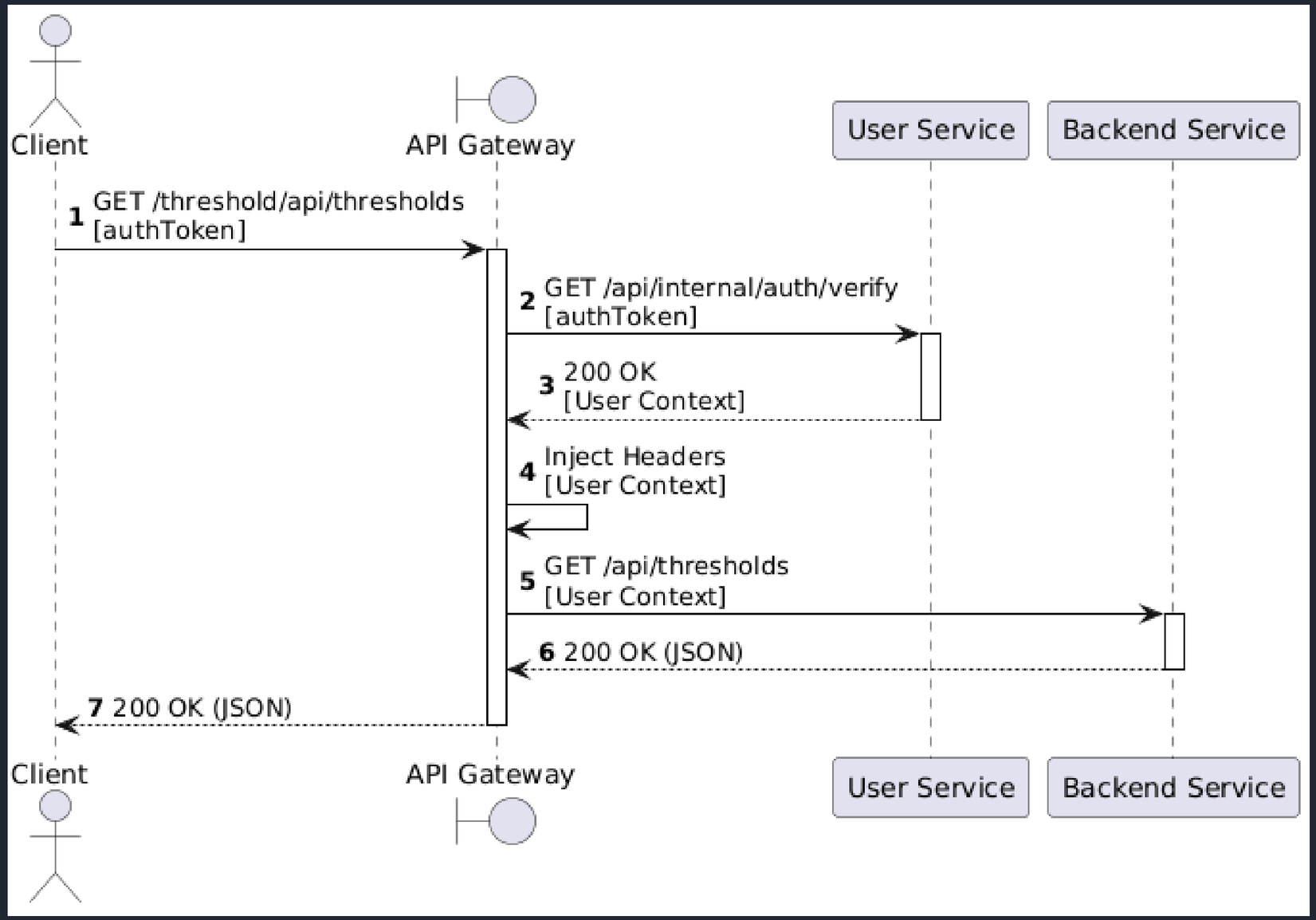
## Asynchronous Publish/ Subscribe

Asynchronous event-driven communication via Apache Kafka serves as the primary mechanism for Backend → Backend interactions. To ensure system reliability, we implement the Transactional Outbox pattern to solve the dual-write problem, and the Inbox pattern to handle message deduplication

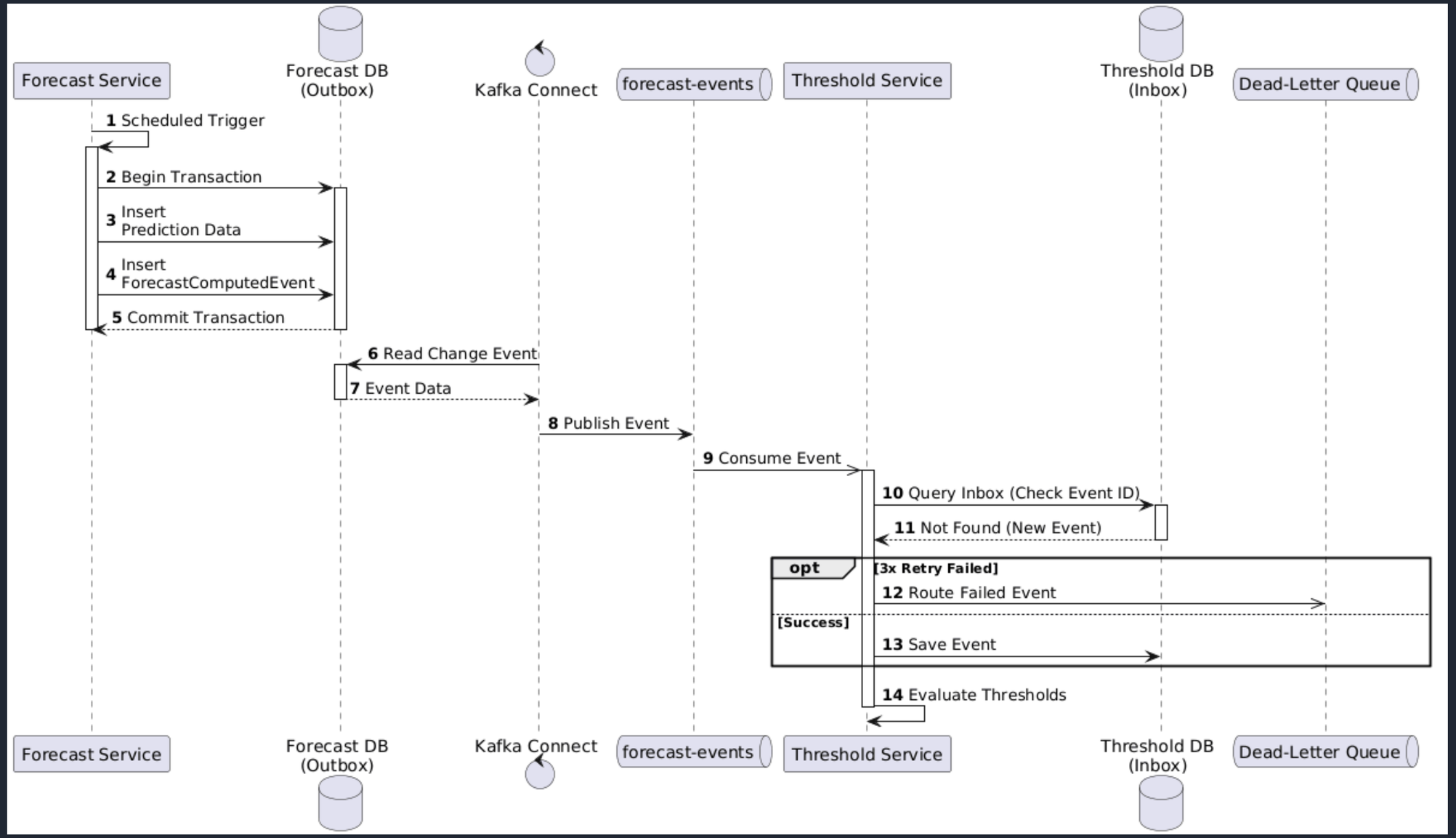
## Real Time Push

Only two components use this type of communication, specifically:
The monitoring service pushes live meter updates via Web-Socket.
The notification service pushes alerts via SSE. Both channels authenticate the client before opening the connection.

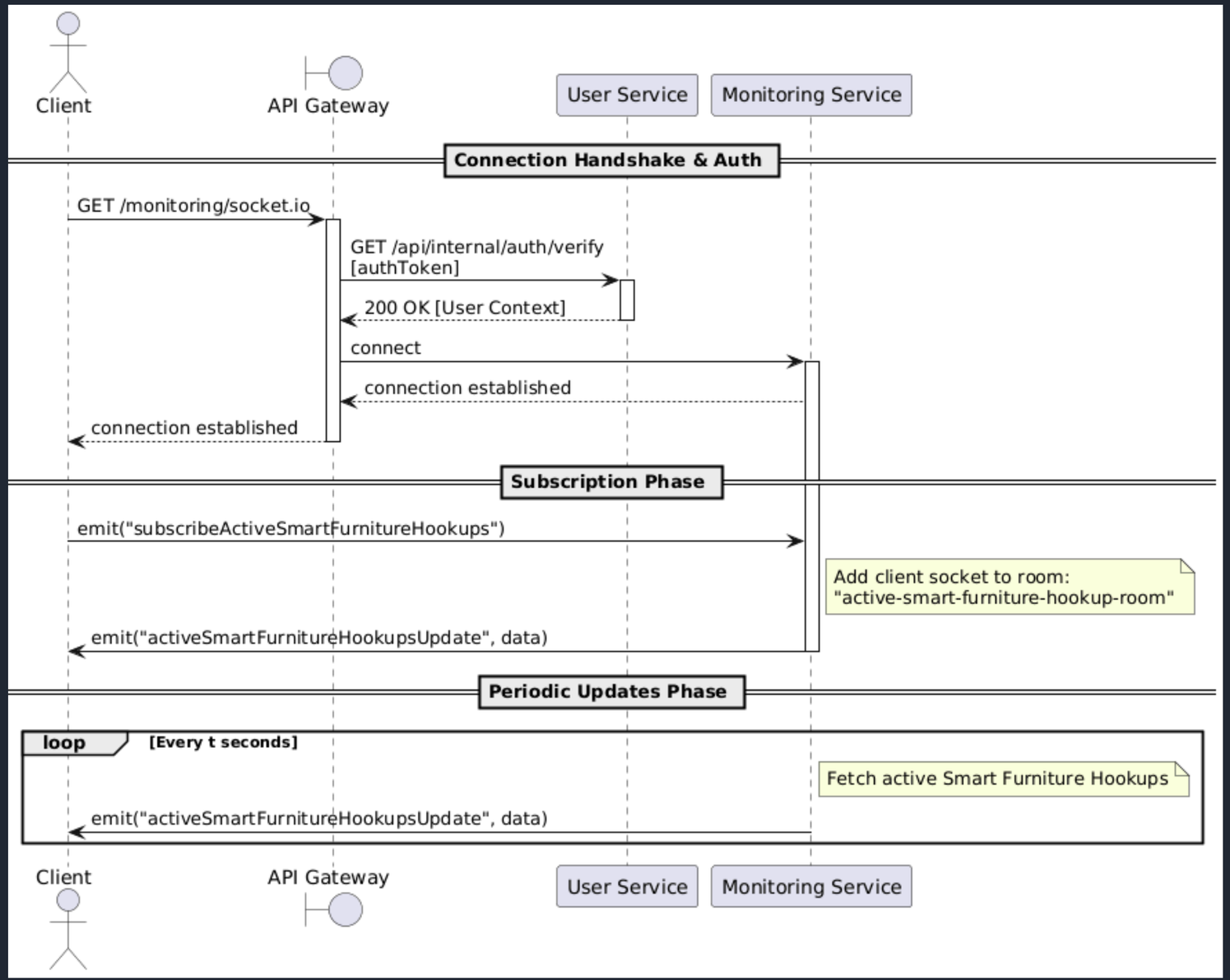
# Frontend HTTP REST Interaction



# Events Interaction



# WebSocket Interaction



# Monitoring Service

## Ingest measurements

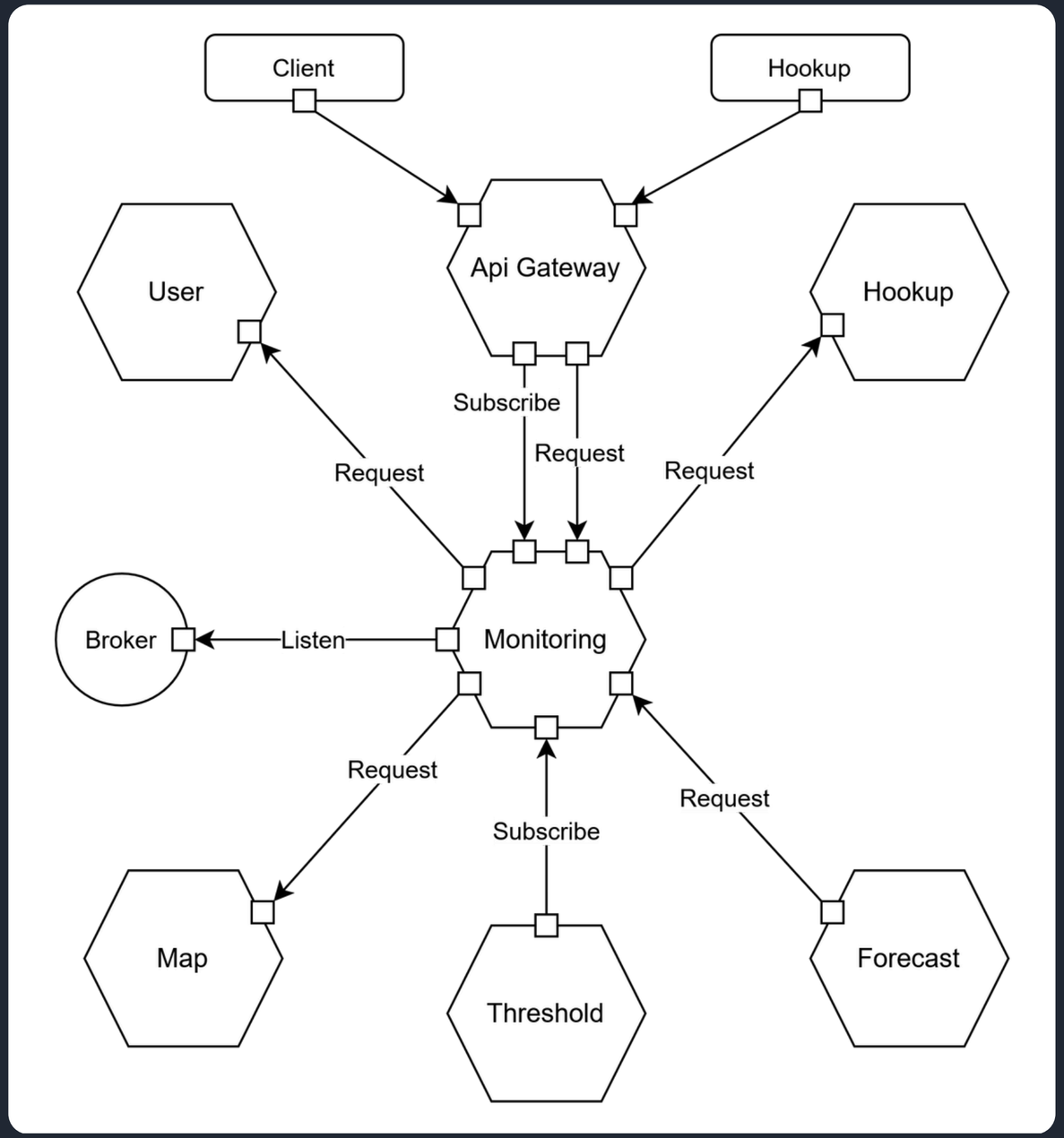
Ingest measurements from smart furniture hookups, validating the hookup ID, zone, and (if provided) the username.

## Update measurements

Listen for the delete events to remove the associated user and zone tags from measurements.

## Real Time Consumptions

Send consumptions in real time using web sockets.



# Event driven cache

## Update cache using events

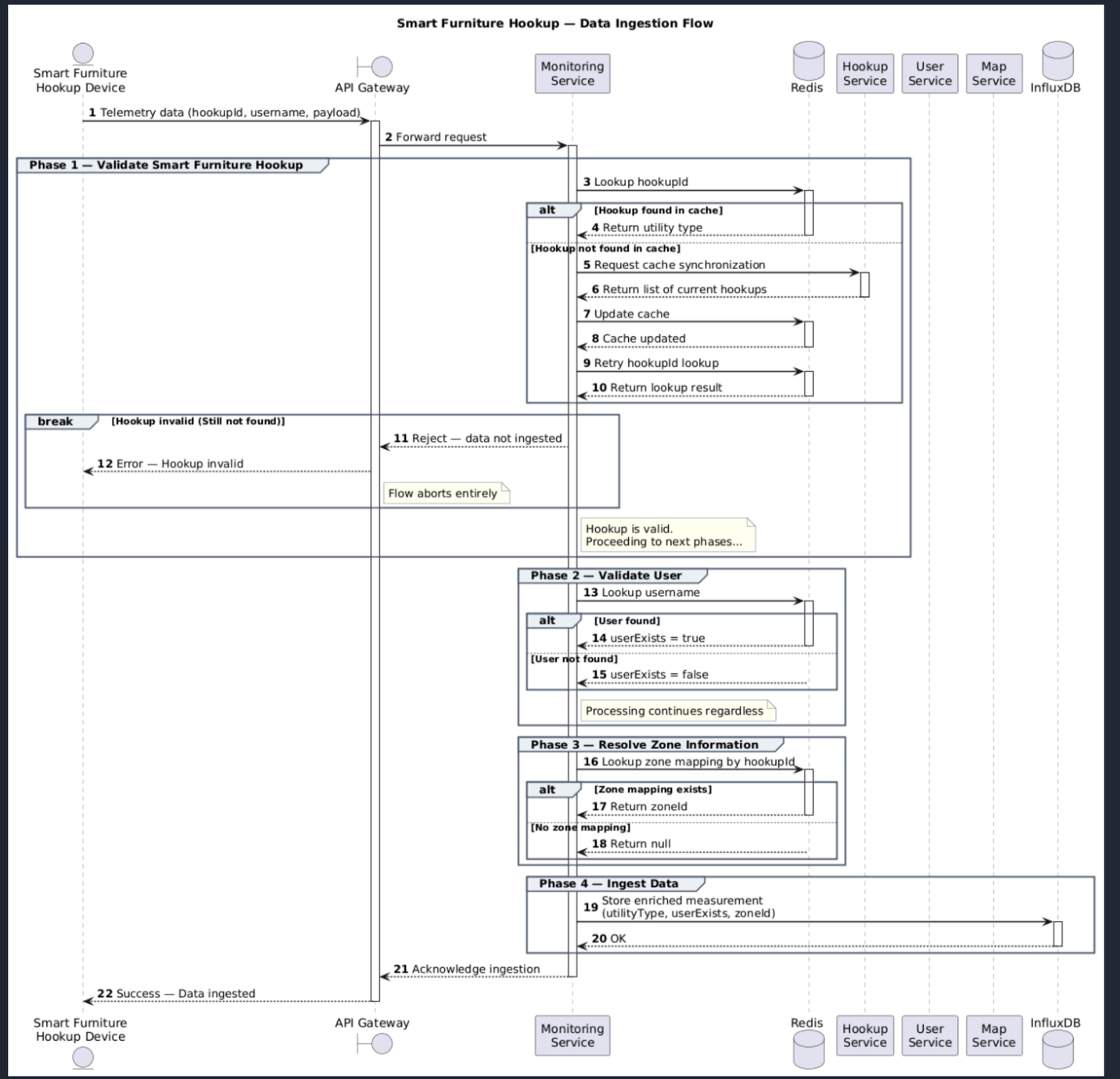
Listen for create, update and delete events and update the cache accordingly

## Use cache for resolving tags

Rely on cache to resolve Hookup, Zone and User tags instead of direct service requests

## Fallback policy

If the broker is unreachable, rely on direct HTTP request until the broker is up again



# Data Consistency

## Storage Strategy

Database-per-service pattern. MongoDB for storing business data. InfluxDB for storing consumption data.

## Query Strategy

- Monitoring service pushes real-time updates via WebSockets.
- Configuration and management services interact with MongoDB using standard RESTful APIs.

## Shared Data

Data crosses service boundaries via four specific mechanisms:

- Events
- API Gateway Headers
- WebSocket
- HTTP Rest API

# Availability

## Redis Cache

The cache enables tag resolution even when services are unreachable, as long as the broker is working

## Prioritize measurement ingestion

In case of a network partition, the system prioritizes consistency for consumption data, while favoring availability over consistency for tag-related information

# Fault-Tolerance

## Async Decoupling & Outbox/Inbox Patterns

Prevention of cascading failures and data corruption. Kafka buffers messages to isolate consumer downtime. The Outbox pattern ensures the system survives sudden application crashes mid-transaction with zero data loss. The Inbox pattern tolerates network problems by safely discarding duplicate retries.

## Error Containment via Dead Letter Queues

Event processing failures are absorbed by an explicit retry policy with exponential backoff (max 3 attempts). Events that persistently fail are routed to dedicated Dead-Letter Queues.

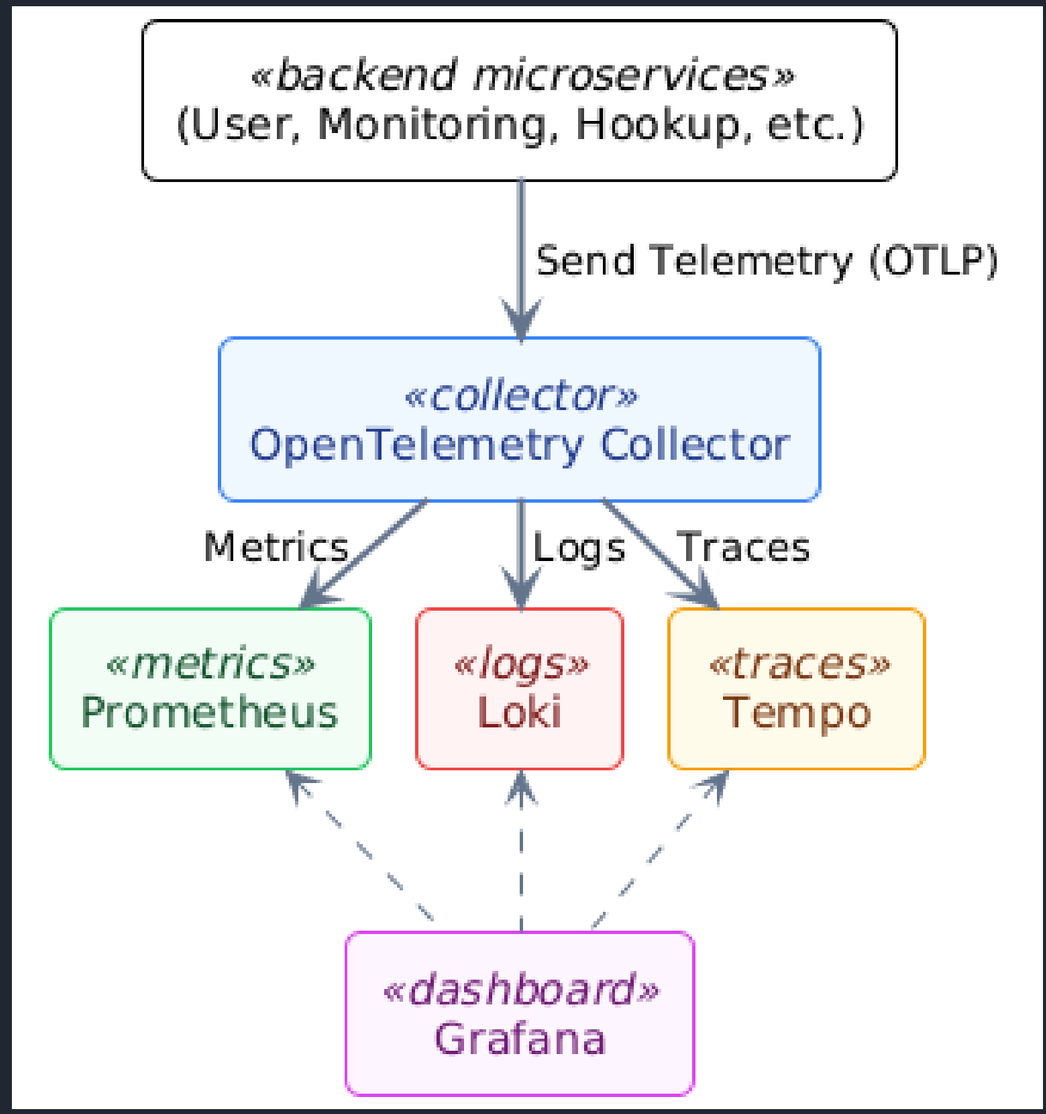
## Stateful Crash Recovery

Microservices and the CDC connector (Kafka Connect) do not rely on volatile memory. Upon recovering from a crash, they utilize strictly committed internal offsets to resume tailing databases and event streams exactly from their last checkpoint.

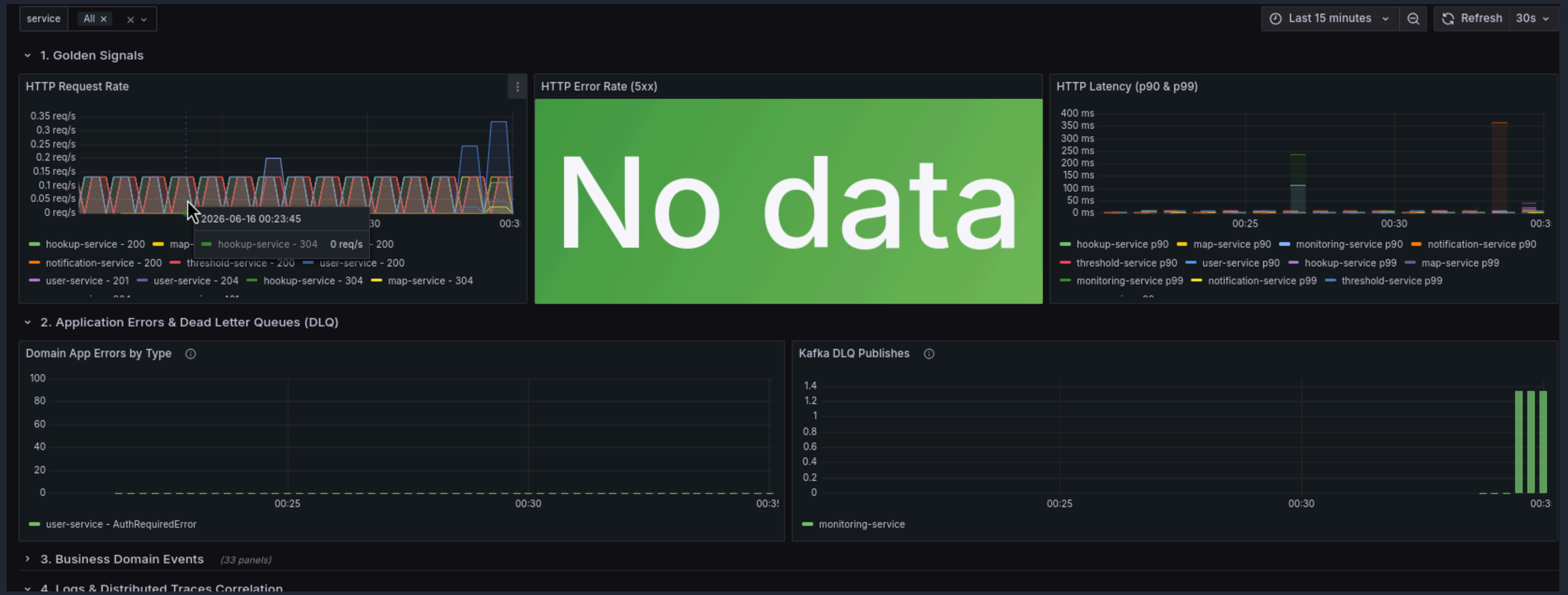
# Observability Stack

We have introduced a dedicated stack:

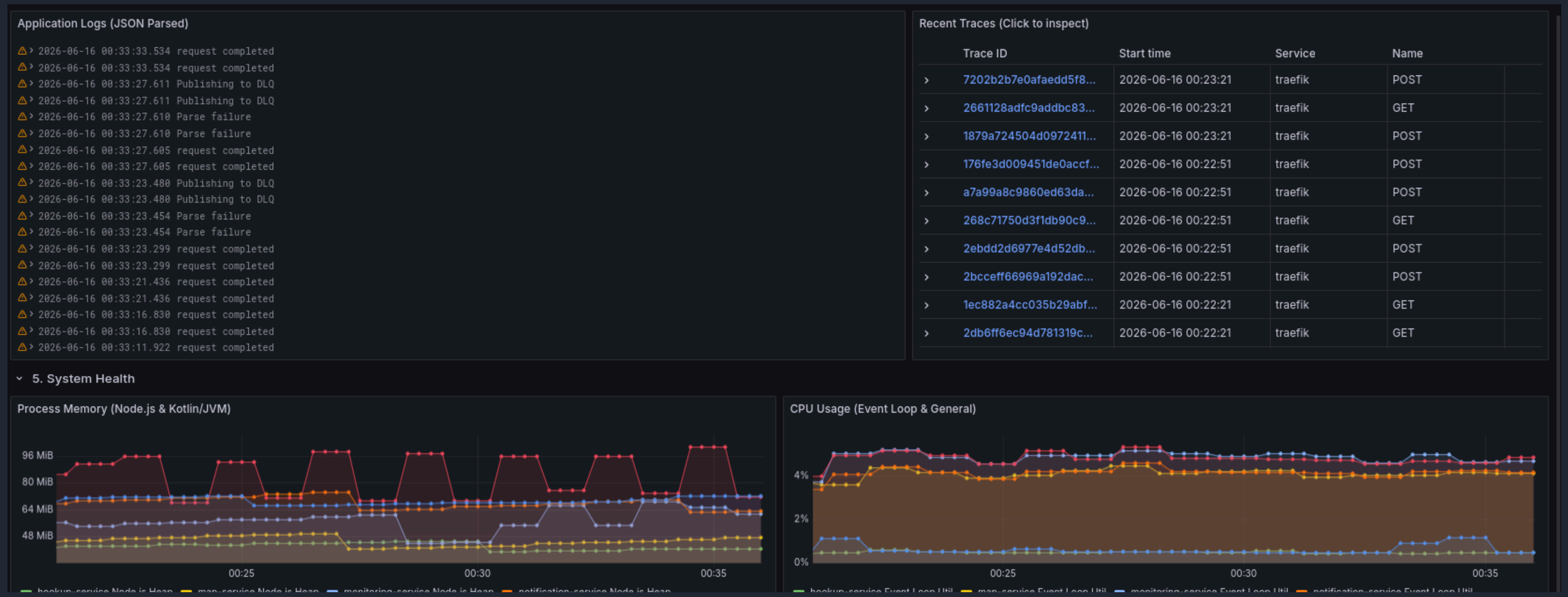
- OpenTelemetry is used to send and route (via a collector) telemetry traffic.
- We track Logs (Loki), Metrics (Prometheus), and Traces (Tempo).
- Visualization: Grafana queries the data to display it in dashboards.



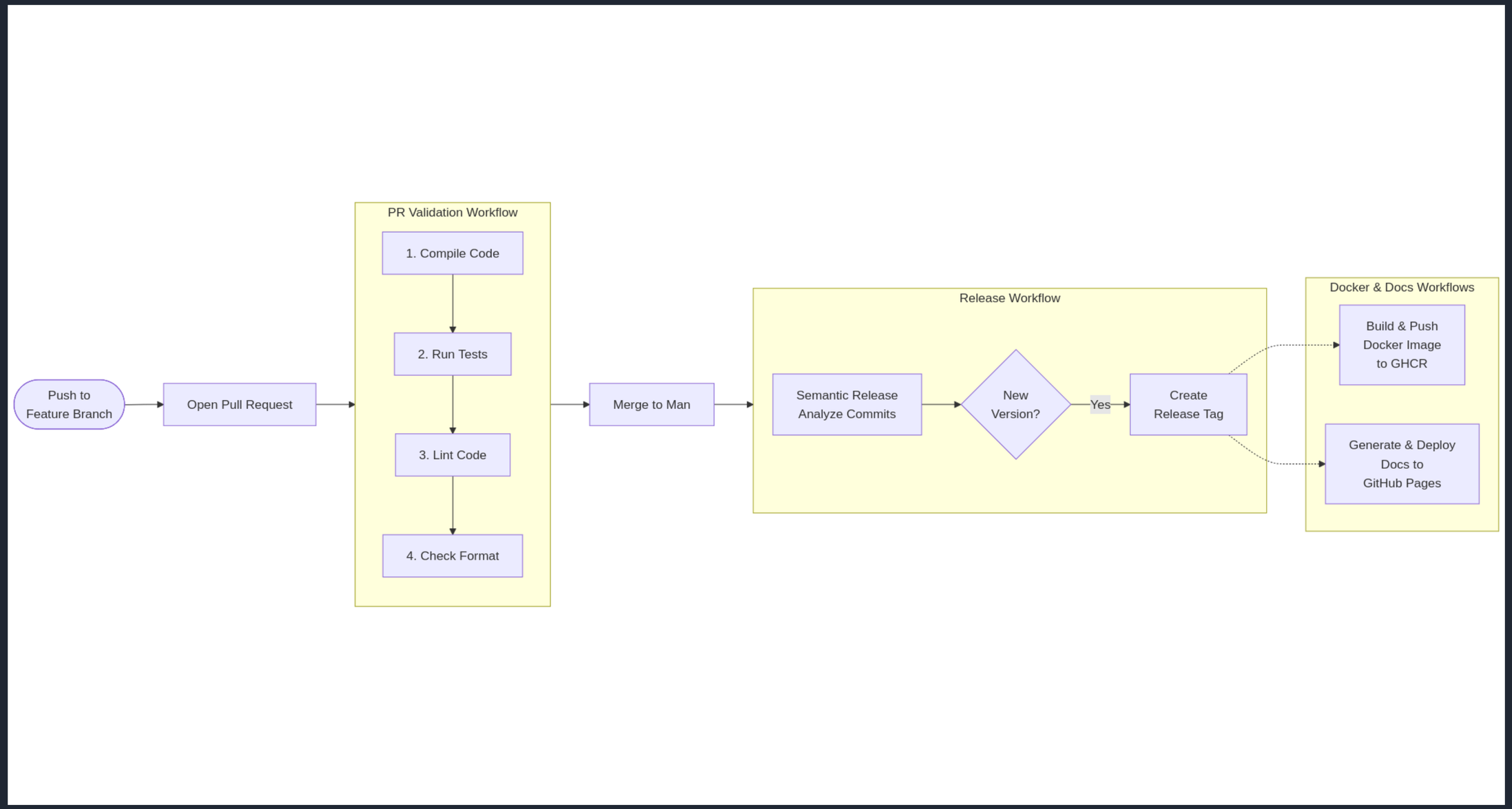
# Grafana Dashboard



# Grafana Dashboard



# CI/CD Pipeline



# Tests

## E2E

Tests the entire application through user journey tests. We used both automated (Playwright) and manual tests to verify the system through user stories.

## Component

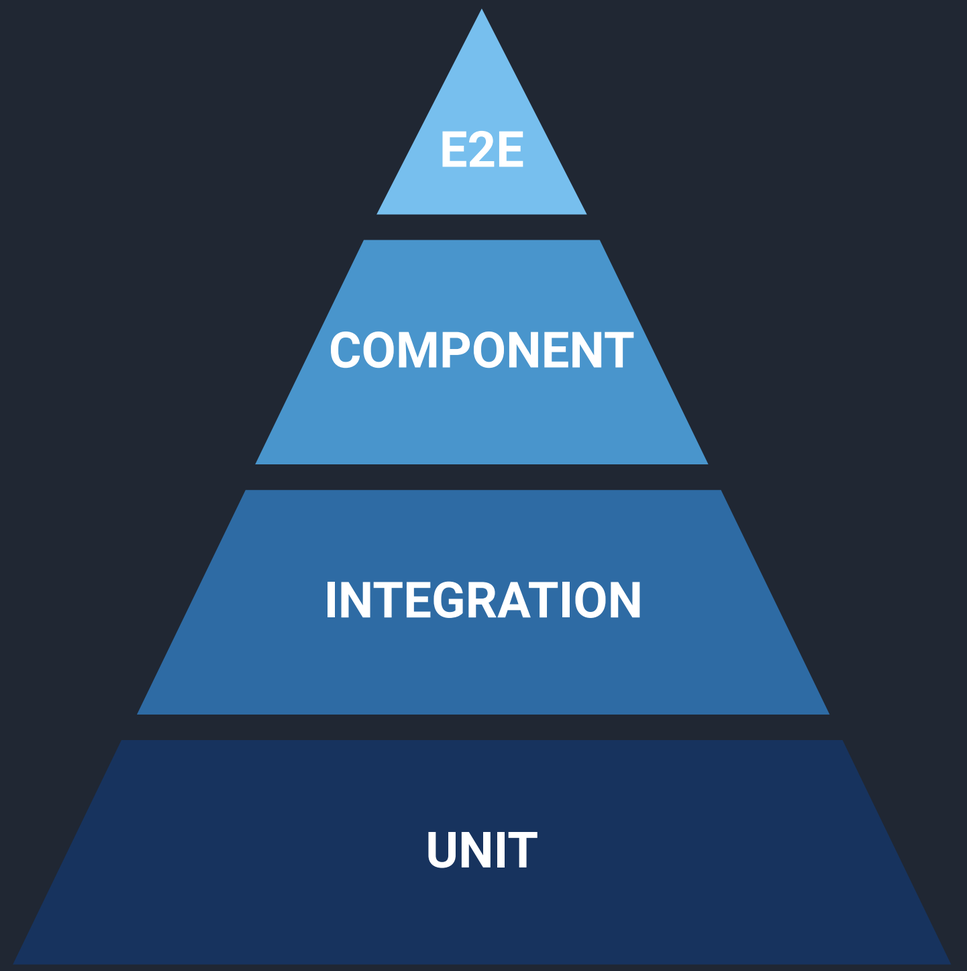
Verifies the behavior of a service in isolation. It treats the service as a black box by replacing its external dependencies with stubs.

## Integration

Verifies that a service can properly interact with infrastructure and other application services. For example, to test database access logic.

## Unit

Verifies that a small part of a service works correctly. It includes both solitary unit tests (e.g., testing REST Controllers or event handlers using mocks) and sociable unit tests (e.g., testing Domain entities).



# Deployment

## GitHub Container Registry

Each Docker image for the respective microservices is pushed to GHCR.

## Docker Compose

We use Docker Compose for both local development (leveraging docker override) and production.

## Configurazione

Proper configuration of certain environment variables is required, as described step-by-step in the documentation.

# Thanks for your attention!