

16-6-2025

Version: 1.0

Software Analysis and Design

Module

ICT.GP.PRJCT.V22_2425

Coach

[Redacted]

Education

Windesheim Zwolle HBO-ICT Software Engineering

Students

Bark, Ivan (s1169347)

[Redacted]

Version management

Version	Date	Description	Remarks
0.1	22-04-2025	Document structure setup	N/A
0.2	06-05-2025	Document restructure	N/A
0.3	26-05-2025	Updated to current sprint	N/A
0.4	09-06-2025	Updated to current sprint	N/A
1.0	16-06-2025	First final version	Send for assesment

Distribution

Name	Role	Date	Version
	Coach	18-06-2025	1.0

Contents

Introduction.....	4
1. Project overview	5
2. System Context and Component Overview	6
2.1. Level 1	6
2.2. Level 2	7
2.3. Level 3	8
2.4. Level 4	11
3. Software Architecture	33
3.1. MQTT Module	33
3.2. Navigation Module.....	35
3.3. Maintenance Detection Module	37
3.4. Machine detection.....	39
3.5. Storage API.....	42
3.6. SSE & HTTP Client.....	44
3.7. Code Quality Practices	46
4. Functional Scope and User Interactions	48
4.1. Use cases	48
5. Data Structures and Relationships	53
5.1. Domain Model.....	53
5.2. Databases.....	54
6. User Flows and Feature Usage	56
6.1. Entire app.....	56
6.2. Maintenance detection Module.....	56
6.3. Navigation Module.....	57
6.4. Visualization Module	57
6.5. Storage API.....	58
7. API Interface Design.....	59
7.1. MQTT	59
7.2. Navigation Module.....	61
7.3. Maintenance detection module	62
7.4. Machine	62
8. Real-Time Data Handling.....	63
8.1. MQTT Topic	63
8.2. MQTT Flow	63

8.3.	Server Send Events	64
9.	Infrastructure and Deployment.....	65
9.1.	Hosting and Orchestration	65
9.2.	Domain and Access Configuration	65
9.3.	Image Registry and Deployment workflow	66
9.4.	Monitoring and Maintenance.....	66

Introduction

This technical design provides a detailed overview of the technical requirements, system architecture, and development approach for the product. It outlines what's needed to meet both the functional and non-functional goals of the project.

The aim is to translate the project's objectives into a solid technical foundation. This includes a breakdown of the software architecture, such as the AR application itself, the supporting API, and how everything connects to a central database. The document also highlights the chosen technologies, frameworks, and tools, and explains why they were selected and how they'll be used throughout the project.

We'll also cover the proposed system infrastructure, how the different components communicate, how data flows through the system, and what's needed to support that, whether it's cloud services, hosting platforms, or real-time data handling.

One of the standout features of this project is the use of fuzzy logic. This allows the AR Assistant to make smart decisions based on uncertain or gradual input, similar to how a real technician would approach a situation. This document explains how fuzzy logic is integrated into the system and how it supports the assistant's functionality.

1. Project overview

This project is proof of concept for an Augmented Reality (AR) application designed to assist technicians in navigating and learning their way around large and complex factory environments. As manufacturing facilities and technological sophistication grow, new technicians often face challenges in finding their way to the correct machines and understanding unfamiliar equipment. The goal of this app is to bridge that gap by providing an AR-guided interface that overlays navigation cues, contextual machine information, and real-time status data directly into the technician's field of view. This enables faster onboarding, better orientation, and increased situational awareness, particularly in high-density or high-risk environments.

The application is developed using Unity with mobile devices as the primary target for this prototype phase. However, the interface and infrastructure are designed with future support for dedicated AR headsets such as Magic Leap or HoloLens, which would allow for hands-free operation and more immersive spatial awareness. The backend, built with NestJS, handles all business logic including 3D pathfinding, fuzzy logic for interpreting machine conditions, and data orchestration. To keep the AR frontend lightweight and focused solely on visualization, the backend is responsible for computing and distributing all relevant data.

The architecture separates static and dynamic data into two database layers. A static data source stores persistent factory information such as 3D layouts and machine metadata, while real-time machine state data is streamed into a separate live database via an external API. This real-time data is then routed through a message broker, currently implemented using MQTT (Mosquitto in a local Docker setup). While MQTT is used in this prototype for its simplicity and publish-subscribe model, the system is designed to be modular, allowing for future substitution with other real-time communication protocols (such as WebSockets, AMQP, or gRPC) depending on scaling needs and integration requirements.

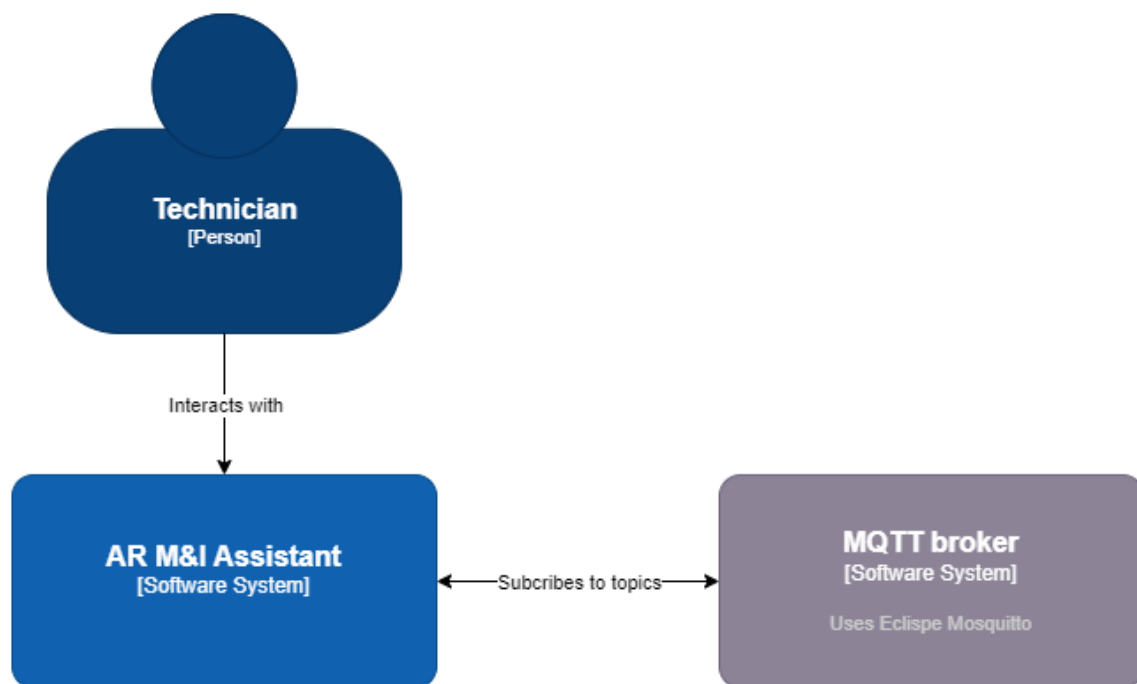
2. System Context and Component Overview

This chapter provides an overview of the system architecture using the C4 model. The C4 model helps visualize the structure of the product and shows how different parts of the system are connected and interact with each other. It also illustrates how the product fits into the broader environment it operates in.

The first three levels of the C4 model give a clear view of the system's overall structure, while the fourth level dives deeper into the internal code-level design. Each level highlights a specific layer of context related to the product:

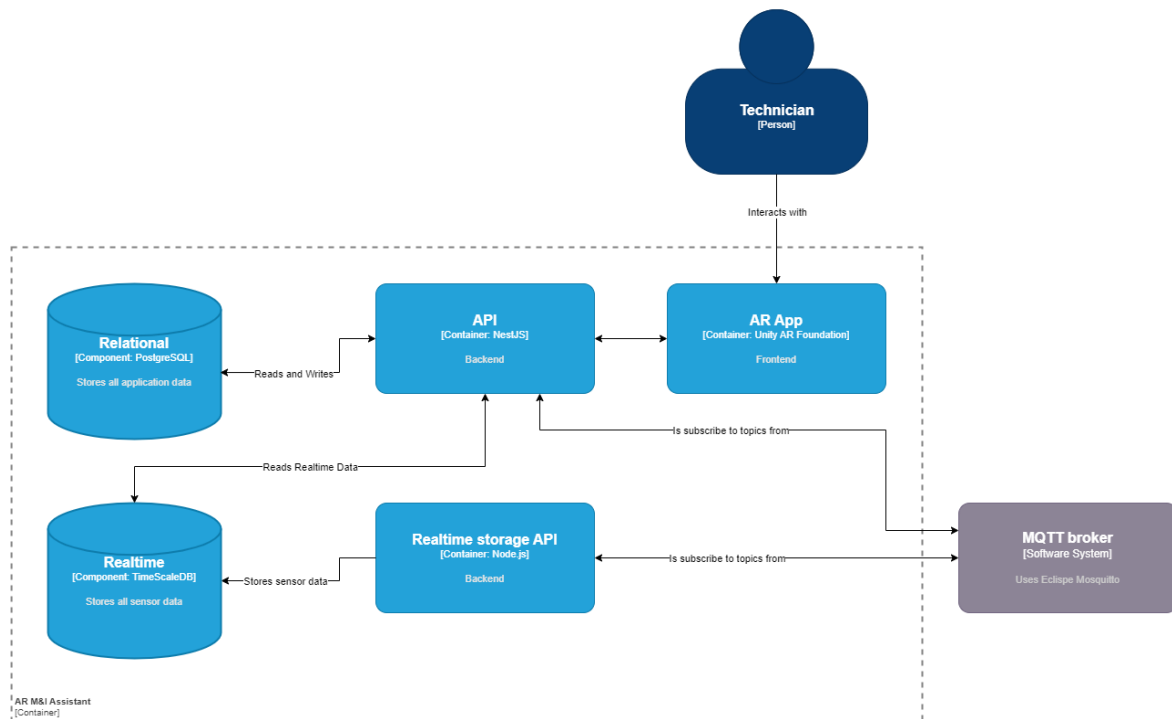
- **Level 1 System Context:** shows how the software fits into its external environment.
- **Level 2 System Containers:** outlines the main parts the system is made of, such as applications, APIs, and databases.
- **Level 3 System Components:** breaks down the internal structure of each container.
- **Level 4 System Code:** describes the most important parts of the actual implementation.

2.1. Level 1



At the highest level, the application interacts with two key stakeholders: technicians who use the AR app on mobile devices and factory systems that expose machine data via external APIs. From a system perspective, the AR frontend communicates exclusively with a backend API, which orchestrates data retrieval, transformation, and logic execution.

2.2. Level 2

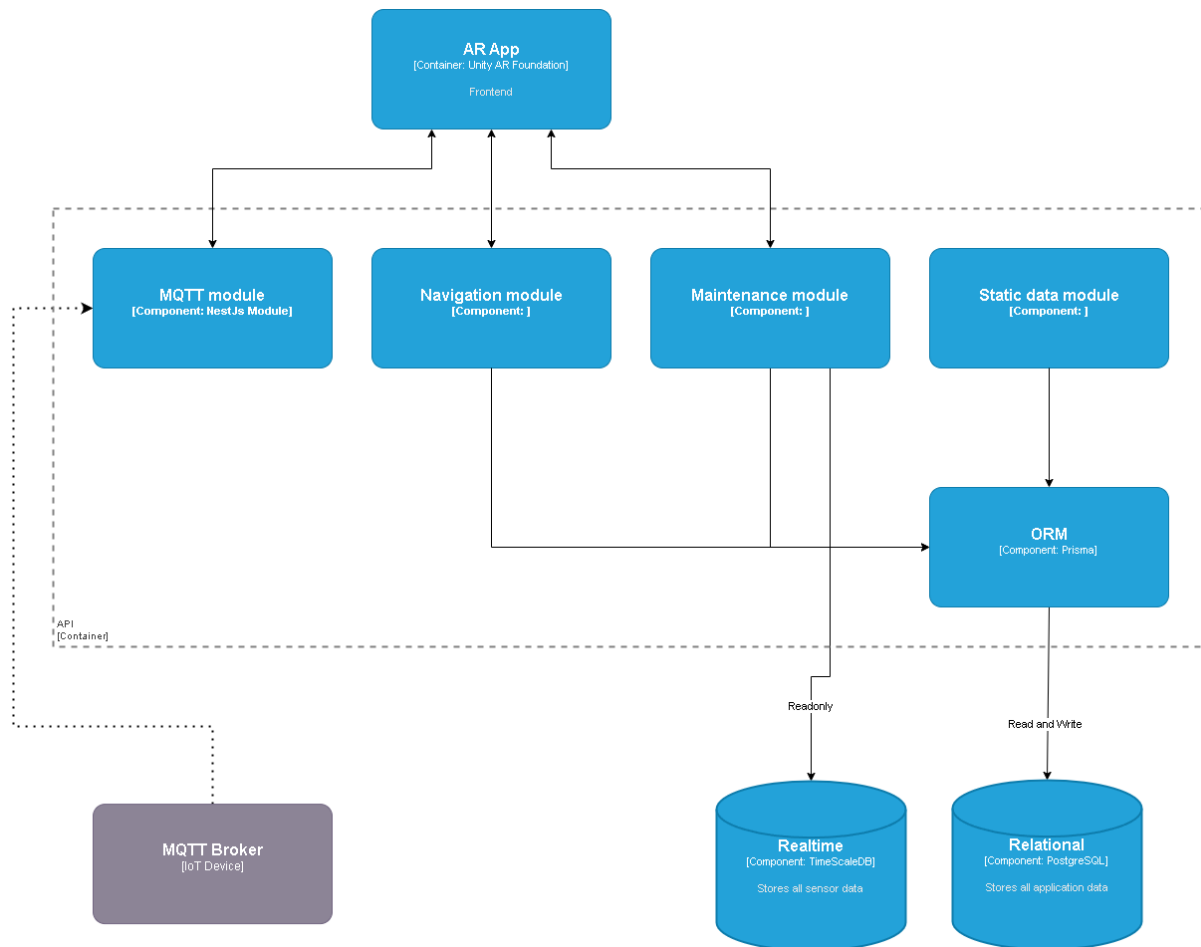


Internally, the backend comprises distinct modules that separate concerns such as pathfinding, fuzzy logic evaluation, sensor data ingestion, and database access. Static information like machine locations, factory geometry, and metadata is stored in a traditional relational database, while incoming real-time sensor data is stored in a dedicated real-time datastore and managed through a broker-based messaging system. The current implementation uses MQTT for real-time message distribution, allowing backend components and frontend clients to subscribe to relevant machine or environment updates.

The AR frontend, developed in Unity, functions purely as a visualization layer. It fetches processed data from the API and renders it contextually into the user's environment using AR Foundation. Responsibilities such as navigation path computation, machine state interpretation, and fuzzy logic decisions remain fully centralized in the backend. This design allows the client to remain lightweight and hardware-agnostic, easing future transitions to more advanced AR platforms such as Magic Leap or HoloLens.

2.3. Level 3

2.3.1. API



This model diagram illustrates the internal component structure of the API backend container within the system architecture. The backend handles real-time IoT data, route computation, maintenance logic, and data storage for the AR application frontend through a modular design that enables separation of concerns and scalable functionality.

The MQTT Module, implemented as a NestJS module, acts as the bridge between IoT devices and the backend by subscribing to the MQTT Broker to receive real-time sensor data such as temperature and vibration, which is then ingested for storage and processing. The Navigation Module is responsible for computing the shortest or most efficient paths based on static graph data and the current user position. It interfaces with the Static Data Module to retrieve graph information and returns path results to the AR frontend. The Maintenance Module encapsulates predictive maintenance and decision support logic, potentially using fuzzy logic or rule-based systems to interpret sensor values and recommend maintenance actions.

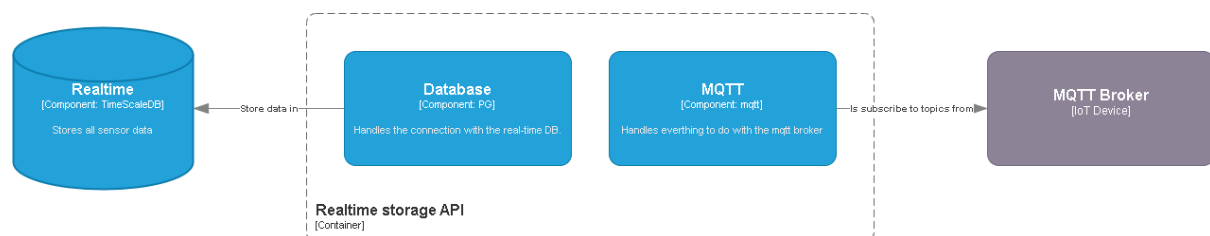
The Static Data Module provides access to static building and graph data, including node positions, floorplans, and topology, fetching this data from the relational database via the Prisma ORM. Prisma acts as the interface to both the real-time and relational databases, abstracting data model access and offering methods for reading and writing persistent data while enforcing schema constraints and ensuring system-wide consistency.

The backend relies on two datastores: TimescaleDB, optimized for handling high-frequency time-series data from IoT devices and storing real-time sensor values accessed primarily by the Maintenance Module. And PostgreSQL, which stores static and persistent application data such as user information, graph data, metadata, and maintenance rules, with access provided through Prisma.

Important to note is that this API is not responsible for writing to the time-series database. To remain performant, a separate Storage API is hosted to handle this writing part. Therefore this API only has read capabilities and can use the timeseries data for calculations within the fuzzy logic.

External systems include the MQTT Broker, a publish-subscribe broker that delivers sensor data from IoT hardware to the MQTT Module in real time, and the Unity AR Foundation-based AR App, which visualizes paths, sensor data, and maintenance instructions for users. The AR App communicates with the backend via HTTP or WebSocket to retrieve computed paths, updates, and trigger maintenance workflows.

2.3.2. Storage API



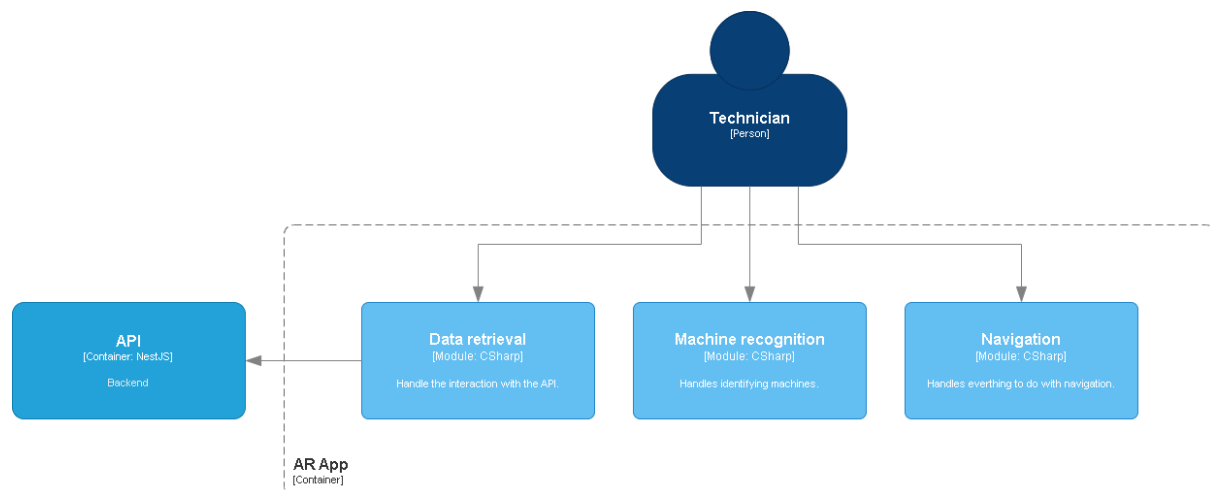
This component establishes and manages persistent connections to both an MQTT broker and a TimescaleDB (PostgreSQL) database, using the Singleton pattern to ensure efficient resource usage. Upon startup, it connects to the MQTT broker and subscribes to all incoming messages using a wildcard topic (#). It listens continuously for sensor data, assuming each MQTT message contains a valid JSON payload structured according to a predefined schema, including an ISO 8601 timestamp.

Each message is parsed and validated as JSON. The data is then inserted into the sensor data hypertable in TimescaleDB. Any errors during this process, either from MQTT subscription or database insertion, are logged for diagnostics, but no retry or alerting mechanisms are implemented yet.

The service is designed to shut down gracefully on termination signals (e.g., SIGINT), disconnecting cleanly from the MQTT broker and the database. It relies on environment variables (supplied via Docker or .env files) for configuration, including database URI credentials and broker address. No authentication is required for the MQTT connection, and message validation is limited to JSON parsing.

This component assumes the infrastructure, TimescaleDB instance, hypertable creation, and MQTT broker, is already provisioned and accessible at runtime.

2.3.3. Frontend



The AR app is composed of three core modules, all developed in C#, which work together to assist technicians in the field.

1. **Data Retrieval Module:** This module is responsible for communicating with the backend API to fetch relevant data needed for the application. It ensures that the AR app always has up-to-date information about machines, procedures, and environments.
2. **Machine Recognition Module:** This module enables the app to identify machines in the technician's surroundings. It uses image tracking to recognise machine location and overlay relevant static and real-time data.
3. **Navigation Module:** This module visualises the path the technician should follow and is responsible for displaying the nodes to assist the technician with following the path. It maps the coordinates from model space to world space.

2.4. Level 4

2.4.1. MQTT Module

The MQTT module facilitates real-time communication between the API and an MQTT broker. It enables the system to subscribe to real-time sensor data, published by a mocking script, to specific MQTT topics and stream this data to clients via Server-Sent Events (SSE). This allows the Unity-based AR frontend to receive continuous updates about machine states directly within the technician's field of view. The module provides dynamic topic-based subscriptions and several API endpoints to manage broker connections, subscribe or unsubscribe from sensor data, and retrieve both live and latest sensor readings.

More information about this module found in MQTT Module

Structure

The MQTT module is organized into a clear, modular structure that separates responsibilities across services, controllers, and validation logic. This structure makes the module easy to extend and maintain and ensures that each part of the MQTT functionality is encapsulated within its own component.

The main file, `mqtt.module.ts`, defines the module by registering the necessary controllers and the `MqttService` provider. This file serves as the central configuration point, bringing together all MQTT-related logic. The following image shows the structure of the MQTT directory, along with a code fragment from the actual module definition:

- `mqtt.module.ts`: Declares the module, including its service and controllers. It encapsulates all MQTT logic, making it reusable and easy to manage within the broader NestJS architecture.
- `mqtt.service.ts`: Implements a singleton service that handles all interactions with the MQTT broker. It manages connection state, topic subscriptions, incoming message routing, and broadcasting data via Server-Sent Events (SSE).
- `MqttService`: Central class that contains all the business logic for MQTT communication. This includes subscribing to topics, maintaining live message streams, handling reconnection events, and keeping track of the latest message per topic.
- `MqttClientController`: Provides endpoints to connect or disconnect from the broker and to retrieve the current connection status. This controller is mostly used for testing and diagnostics.
- `MqttSubscriptionsController`: Defines the main public-facing endpoints for subscribing to or unsubscribing from machine topics. It also handles SSE-based streaming of live sensor data to the Unity frontend.
- `MqttDebugController`: Offers debugging utilities such as retrieving the latest message from a topic or testing the live stream in a basic HTML view. These endpoints are used during development to verify sensor data flow.
- `parse-sensor.pipe.ts`: Implements validation logic to ensure that only allowed sensor types (such as Temperature, Pressure_in_PSI, and the wildcard '#') can be passed into the API. This adds an extra layer of type safety and control.

This modular layout supports a clean separation of concerns and makes it straightforward to integrate this module into other parts of the application or to expand it with new MQTT-related features.

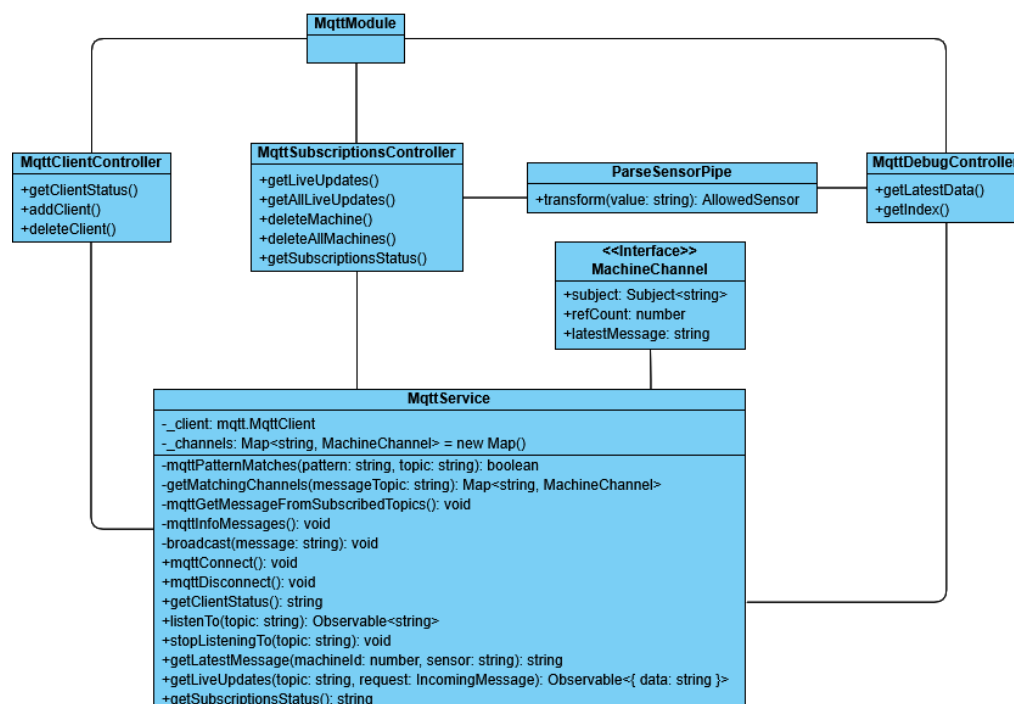
```
mqtt.module.ts

/**
 * mqtt/
 * ├── mqtt.module.ts           # Main module definition
 * ├── mqtt.service.ts         # MQTT logic: subscription, SSE, connection handling
 * ├── mqtt-client.controller.ts # MQTT client management (connect/disconnect/status)
 * ├── mqtt-debug.controller.ts # Endpoints for debugging and snapshot testing
 * ├── mqtt-subscriptions.controller.ts # Subscriptions + SSE streaming per topic
 * └── parse-sensor.pipe.ts     # Validation pipe for sensor types
 */

@Module({
  imports: [],
  controllers: [
    MqttClientController,
    MqttSubscriptionsController,
    MqttDebugController,
  ],
  providers: [MqttService],
})
export class MqttModule {}
```

Code

The core logic of the MQTT module is implemented in the MqttService class. This singleton service handles all MQTT-specific tasks, including connecting to the broker, managing topic subscriptions, processing incoming messages and broadcasting data through Server-Sent Events (SSE). Below is a class diagram of the MQTT module for reference.



Channels

The service uses a private map called `_channels`, which associates each MQTT topic with a reactive stream and metadata. This structure allows multiple clients to subscribe to the same topic without duplicating broker-level subscriptions. The reference count ensures that topics are only unsubscribed when all consumers have disconnected.

```
mqtt.service.ts

private _channels: Map<string, {
  subject: Subject<string>,
  refCount: number,
  latestMessage: string
}> = new Map();
```

Each channel entry contains:

- a `Subject<string>` stream used to push live messages to subscribers
- a `refCount` to manage how many clients are consuming the stream
- the `latestMessage` which stores the last received value for quick retrieval

Method Overview

mqttConnect

Initializes a persistent connection to the MQTT broker using configuration values defined in the `.env` file. Upon successful connection, it automatically re-subscribes to all previously active topics to ensure continuity.

mqttDisconnect

Gracefully closes the active MQTT connection and stops all active topic listeners. This method is typically used during shutdown or diagnostic resets.

getClientStatus

Returns a human-readable status string indicating whether the MQTT client is currently connected to the broker. This is mainly used for diagnostics.

mqttInfoMessages

Registers internal listeners for client lifecycle events like error, offline, and reconnect. When such an event occurs, this method broadcasts a formatted warning or info message to all currently active channels, ensuring that frontend consumers receive relevant connection updates.

mqttGetMessageFromSubscribedTopics

Handles the global message event from the MQTT client. When a new message is received for any subscribed topic, it:

- Finds all matching topics using wildcard-aware pattern matching
- For each match, stores the message in `latestMessage`

- Pushes the message to the corresponding Subject stream so it can be sent to any connected clients

mqttPatternMatches(pattern: string, topic: string)

Private utility function that checks whether a topic matches a given MQTT subscription pattern. It supports both + (single-level) and # (multi-level) wildcards.

- Splits both the pattern and the topic into segments
- Matches each segment based on MQTT rules
- Returns true if the topic structure aligns with the pattern

This method is used internally to support dynamic and flexible topic subscriptions, such as /1/Temperature or 123/#.

getMatchingChannels(messageTopic: string)

Loops through all subscribed topics in `_channels` and applies `mqttPatternMatches` to find which subscriptions match the incoming message topic. Returns a filtered map of only the matching topic entries.

This method allows the system to handle multi-topic matches and ensures that all appropriate consumers receive the message.

listenTo(topic: string)

Subscribes to a new MQTT topic or reuses an existing subscription if one already exists. This method:

- Checks if the topic is already in `_channels`
- If not, subscribes to the topic via the MQTT client and creates a new channel entry
- If already subscribed, increments the `refCount` and returns the existing stream
- Returns an `Observable<string>` stream, which is used in the controllers to power SSE endpoints

stopListeningTo(topic: string)

Decrements the `refCount` of a topic. If no clients remain, it unsubscribes from the topic at the broker level and removes the channel entry from memory.

getLiveUpdates(topic: string, request: IncomingMessage)

This method is the bridge between the backend and the Unity frontend. It creates an SSE-compatible stream of messages for a specific topic. When the HTTP connection to the frontend is closed, the request is cleaned up and the subscription is automatically removed if no one else is using it.

```
mqtt-subscriptions.controller.ts

getLiveUpdates(topic: string, request: IncomingMessage): Observable<{ data: string }> {
  const stream$ = this.listenTo(topic).pipe(map((data) => ({ data })));

  request.on('close', () => {
    this.stopListeningTo(topic);
  });

  return stream$;
}
```

getLatestMessage(machineld: number, sensor: string)

Returns the last known message for the topic composed of {machineld}/{sensor}. If no message is available or the topic is not subscribed, a default info message is returned instead.

getSubscriptionsStatus

Compiles a summary of all currently subscribed topics and their count. This is useful for monitoring the system state and debugging.

Validation Logic

The module uses a custom ParseSensorPipe to validate the requested sensor name before it is passed to the controllers or service. This ensures that only known sensor types (such as Temperature, Pressure_in_PSI, or the wildcard #) can be accessed through the API. This adds robustness and prevents errors caused by typos or unsupported sensor requests.

```
parse-sensor.pipe.ts

export const ALLOWED_SENSORS: string[] = [
  'Temperature',
  'Pressure_in_PSI',
  '#',
];

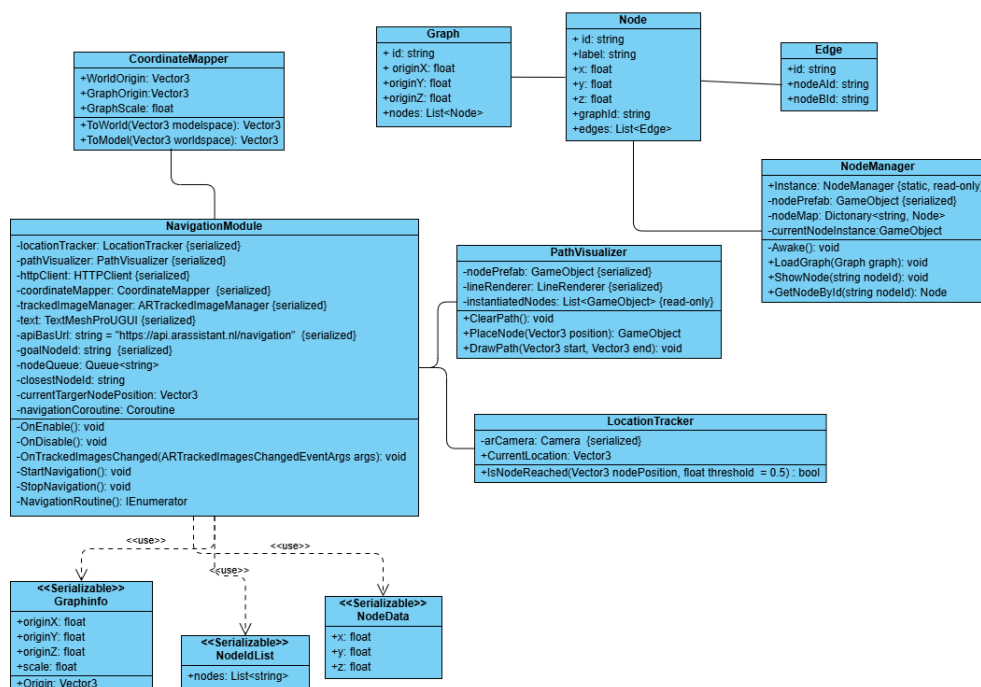
export type AllowedSensor = (typeof ALLOWED_SENSORS)[number];
```


2.4.2. Navigation Module

The NavigationModule is a core component present on both the backend and frontend of the application, responsible for enabling seamless route planning and guidance. On the backend, the module handles all computational logic: it retrieves the static graph and node data from the database and calculates the optimal path between two points using the A* algorithm. The frontend implementation of the NavigationModule focuses solely on visualization. It receives the computed path and node information from the backend, displays navigation points in the AR environment, and draws lines between them to guide the user along the route. This separation of responsibilities ensures efficient performance and a clear division between computation and presentation.

More information about this module found in Navigation Module

Frontend



Structure

The Unity NavigationModule is a prefab containing most of the following key components:

- LocationTracker**
 Tracks the AR device's position in world space. It determines if the user has reached the current target node by measuring distance between the camera and node position. It also provides the current position in the scene via the AR camera transform.
- CoordinateMapper**
 Maps model space coordinates (used by the backend and stored graphs) to Unity world space. It does this based on a spatial origin (e.g., from an AprilTag marker) and a scale factor retrieved from the backend. This ensures the virtual path aligns accurately with the real-world environment.
- PathVisualizer**
 Renders the navigation path in the AR scene. It places visible node indicators at each target position and draws lines connecting the user's current position to the next

navigation node. It supports dynamic clearing and redrawing of the path as the user progresses.

- **HTTPClient**
Handles asynchronous API communication for fetching graph info, nearest nodes, and full path sequences. The responses are parsed into structured node data and passed along to the other components.
- **ARTrackedImageManager**
Detects AR markers in the physical environment. When a marker with ID "0" is detected, the navigation session initializes by aligning the Unity world origin and requesting graph details and path data.

Code

The `NavigationModule.cs` manages all core navigation logic. It relies on Unity's `ARTrackedImageManager` to detect the starting image (marker "0"), which then triggers backend communication and visualization logic.

The most important methods and their responsibilities are explained:

NavigationModule.OnTrackedImagesChanged

This method is triggered when an `AprilTag` image is detected or updated. It uses the image's position to set the world-space origin and fetches the graph's origin and scale data from the backend. This aligns model space to the physical world. Once alignment is complete, `StartNavigation()` is invoked to begin the routing process.

NavigationModule.StartNavigation

This method starts the coroutine that manages the navigation sequence. It is called once a valid marker is detected and graph info is fetched. If needed, the process can be cancelled gracefully using `StopNavigation()`.

NavigationRoutine

This coroutine implements the full navigation logic:

1. Determine the nearest node by transforming the user's current location to model space and querying the backend.
2. Request a path from the nearest node to the goal node using `/navigation/path`.
3. Render and track the path step-by-step:
 - Fetch node positions using `/navigation/node/:nodeId`
 - Convert positions to world space and visualize them.
 - Wait until the node is reached.
 - Proceed to the next node.

The process concludes when the final node is reached and the path is cleared.

CoordinateMapper.ToWorld / ToModel

These methods are used to convert between backend model coordinates and Unity world coordinates.

- ToWorld: Used for placing nodes and drawing the path in Unity space.
- ToModel: Used to determine the user's model-space position for proximity queries.

LocationTracker.IsNodeReached

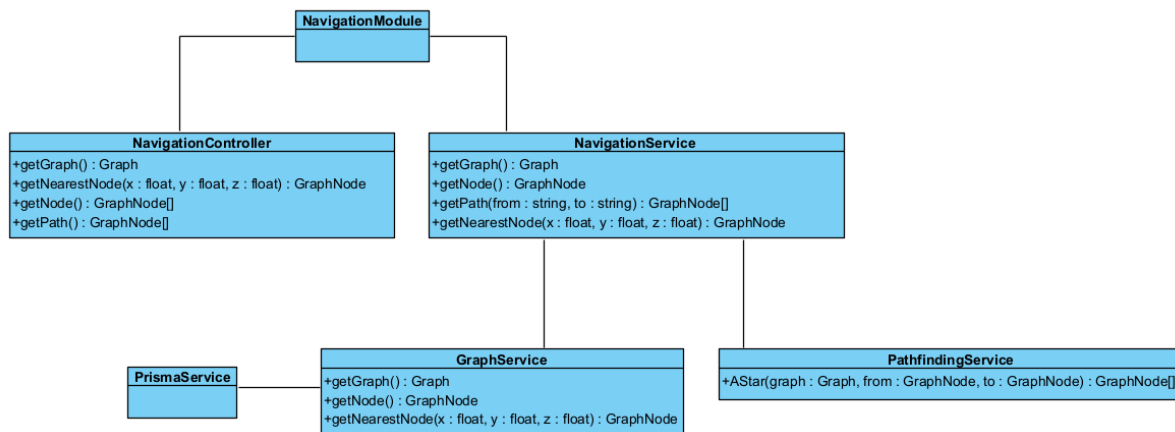
This helper method checks if the technician is close enough to a target node to consider it "reached". The default distance threshold is 0.5 meters.

PathVisualizer.PlaceNode / DrawPath

These methods visually represent the navigation path:

- PlaceNode instantiates a prefab at the current target position.
- DrawPath draws a line from the user's current position to the node.

Backend



Structure

The backend NavigationModule follows the standard modular structure used throughout NestJS applications, promoting maintainability and separation of concerns. The core structure consists of the following components:

- `navigation.controller.ts`: This file defines the HTTP endpoints exposed by the module. It handles incoming requests from the frontend (e.g., for path calculations or graph retrieval) and delegates processing to the appropriate services.
- `navigation.module.ts`: The module file ties everything together, declaring which controllers and services belong to the module and managing their imports and exports.
- `navigation.service.ts`: This service contains the high-level logic for processing requests. It orchestrates the interaction between the graph and pathfinding services to fulfill navigation-related queries.
- `graph.service.ts`: Responsible for accessing and returning the prebuilt graph data stored in the database. This service isolates the database layer from the rest of the logic, making it easier to maintain or extend.
- `pathfinding.service.ts`: This service implements the A* algorithm used to calculate the shortest path between two nodes in the graph. It is designed to work independently of how the graph data is stored or retrieved.

This layered architecture ensures clear responsibilities across the codebase and allows for unit testing and independent scaling of each service if needed.

Code

The backend Navigation Module is structured around NestJS's modular system and separates responsibilities across controllers, services, and algorithm logic. Below is an overview of the most important files and their key methods.

navigation.controller.ts.getGraphInfo:

Handles GET /navigation/graph/:id. Returns the origin point and scale factor of a specific graph. This information is used by the frontend to align model space with the physical world.

navigation.controller.ts.getNearestNode:

Handles GET /navigation/nearest. Accepts user coordinates in model space and returns the ID of the closest node by delegating the logic to the navigation service.

navigation.controller.ts.getPath:

Handles GET /navigation/path. Accepts a starting and target node ID, then returns a list of node IDs representing the shortest route. The actual path is calculated in the pathfind service.

navigation.controller.ts.getNode:

Handles GET /navigation/node/:id. Returns the coordinates of a specific node based on its ID. This endpoint allows the frontend to place the node correctly in the AR environment.

navigation.service.ts.getGraphInfo:

Fetches a graph by ID using the graph service and formats its origin and scale for frontend use.

navigation.service.ts.findNearestNode:

Takes a user's model-space coordinates and compares them against all nodes in the selected graph. Returns the ID of the node closest to the user using Euclidean distance.

navigation.service.ts.getPath:

Receives two node IDs and forwards them to the pathfind service. After computing the path, it returns a list of node IDs that make up the route.

navigation.service.ts.getNode:

Retrieves a node from the graph by ID using the graph service and formats its position data for transmission to the frontend.

graph.service.ts.getGraphById:

Queries the database using Prisma to retrieve a complete graph, including its origin, scale, and all associated nodes and edges.

graph.service.ts.getNodeById:

Searches for a single node by ID from a preloaded graph and returns its spatial coordinates.

graph.service.ts.getAllNodes:

Returns a list of all nodes within a specific graph. This is used by the nearest node logic and for building in-memory pathfinding structures.

pathfind.service.ts.findShortestPath:

Implements the A* (A-Star) pathfinding algorithm. Given a start and end node ID, it constructs an in-memory graph, calculates the optimal route based on cost and heuristic, and returns the resulting node ID sequence.

navigation.entity.ts.Node / Graph / Edge:

Defines the core data structures for a node graph. These interfaces ensure consistent typing and structure for graph data used throughout the controller, services, and pathfinding logic.

navigation.module.ts:

Binds all navigation components into a functional module. Registers the controller and injects all services (NavigationService, GraphService, PathfindService) into the application's dependency graph.

Return type

Some methods have an optional "type" parameter, this is to determine which return type the endpoint should use. Options are 'Shallow' and 'Deep'. With shallow always giving the least amount of data. Described below are the different endpoints and the different outputs per return type:

	Shallow	Deep
getGraph	Graph without 'nodes' property	Graph with 'nodes' property
getPath	An array of GraphNodes	An array of GraphNodeIds
getNearestNode	The Id of the nearest node	The nearest GraphNode object

2.4.3. Maintenance Detection Module

The Maintenance Detection Module is responsible for interpreting live machine sensor data and determining whether a machine requires maintenance. It uses a fuzzy logic system to transform numeric sensor values into qualitative maintenance priority levels, such as "Low", "Medium", "High", or "VeryHigh".

The module operates by continuously listening to MQTT sensor data through the MQTT module and evaluating them using a set of fuzzy rules configured per machine. The resulting evaluations are streamed to the Unity-based AR frontend using Server-Sent Events (SSE), enabling technicians to receive real-time maintenance advice directly in their field of view.

This module is designed to be extensible. New machines can be configured by adding a JSON configuration file specifying which sensors to use and how to interpret their values. The fuzzy logic system itself is fully custom-built, avoiding external dependencies, and implements fuzzy variables, membership functions, rule sets, and defuzzification in a modular and testable way.

More information about this module found in Maintenance Detection Module

Structure

The Maintenance Detection Module is organized into a clear, modular structure that separates responsibilities across services, controllers, and fuzzy logic components. This design supports a clean separation of concerns and ensures extensibility and maintainability.

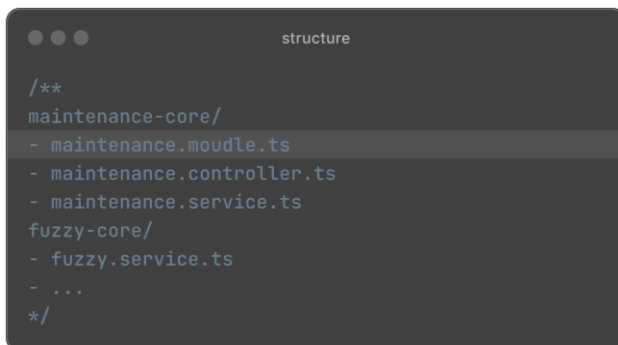
The main file, `maintenance.module.ts`, defines the module by importing the MQTT module and the Fuzzy module, and by registering the `MaintenanceController` and `MaintenanceService`. This serves as the bridge that links real-time sensor input with fuzzy logic evaluation and exposes results to the API.

The module includes the following relevant files:

- **maintenance.controller.ts**: Defines two endpoints: one for retrieving the latest evaluation snapshot (`/maintenance/latest/evaluation`) and one SSE endpoint for streaming live evaluation results (`/maintenance/evaluate`). It acts as the public interface for accessing fuzzy-based maintenance decisions.
- **maintenance.service.ts**: Subscribes to MQTT topics using the MQTT service and processes incoming sensor messages. It maintains a stateful view of the latest sensor values per machine, and triggers evaluations once all expected sensors for a machine are present. It also stores the latest sorted results in memory.
- **fuzzy.service.ts**: The core for the fuzzy evaluation logic. It loads fuzzy rule configurations from config files, creates `FuzzyEngine` instances per machine, and delegates evaluation tasks when sensor data is available.
- **fuzzy.engine.factory.ts**: Responsible for initializing fuzzy engines with specific fuzzy variables and rule sets. It maintains a `machineConfig` object that tracks which sensors are expected for each machine and binds the fuzzy engine instance.
- **fuzzy.engine.ts**: Implements the `FuzzyEngine` class, which evaluates input sensor data against fuzzy rules. It handles fuzzification, rule strength aggregation, defuzzification into crisp scores, and priority label assignment.

- **FLV.ts:** Stands for “Fuzzy Logic Variable”. It encapsulates the structure of a fuzzy variable with labels (e.g., Low, Medium, High) and associated membership values. It also provides interpolation logic for inputs not explicitly defined in the membership map.
- **MembershipFunctions.ts:** Defines functions such as LeftShoulder, Triangular, and RightShoulder which are used to construct the shape of each fuzzy label’s membership function.
- **fuzzy.types.ts:** Contains shared types and interfaces used across all fuzzy logic code. This includes FuzzyRule, SensorData, EvaluationResult, and SensorName.
- **config/fuzzy/*.config.json:** These configuration files define machine-specific fuzzy setups. Each file specifies a machine ID, the sensors to expect, and a list of fuzzy rules to apply. This allows easy reconfiguration and extension without code changes.

This modular layout supports a clean separation of concerns and makes it straightforward to integrate this module into other parts of the application or to expand it. Below an image to show the actual file structure:



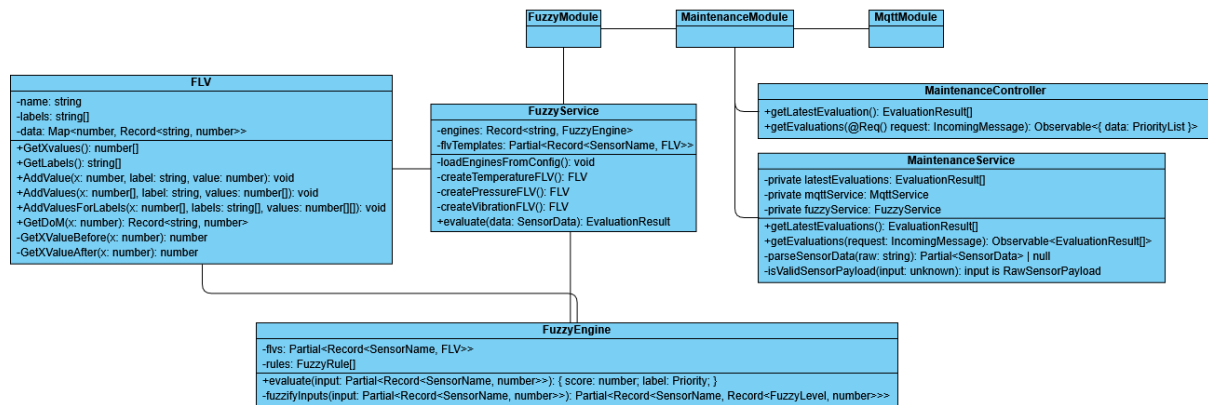
```

/**
maintenance-core/
- maintenance.moudle.ts
- maintenance.controller.ts
- maintenance.service.ts
fuzzy-core/
- fuzzy.service.ts
- ...
*/

```


Code

The core logic of the Maintenance Detection module is implemented in the FuzzyService class. The rest of the logic is split into the FuzzyEngine, FLV, MembershipFunctions and MaintenanceService classes.



Method Overview

Understood. Based on your original MQTT module example, you want the **Method Overview** structured with one labeled section per method, each followed by a short paragraph describing how it works and what it does.

FuzzyService.evaluate

This method receives a SensorData object containing sensor values and the machineId. It locates the corresponding fuzzy engine and uses it to compute a fuzzy logic evaluation. The result includes a crisp score (0–100) and a label (Low, Medium, High, or VeryHigh) representing the maintenance priority. If the machine is not registered or misconfigured, an error is thrown.

FuzzyService.loadEnginesFromConfig

Executed on service construction, this method loads all .config.json files from the config/fuzzy/ directory. Each config file specifies a machine ID, a list of expected sensors, and the fuzzy rule set. The method constructs fuzzy logic variables for each sensor using predefined templates, then uses createEngineFor to create and register a FuzzyEngine per machine.

FuzzyEngine.evaluate

This method performs the full fuzzy logic evaluation process. It first fuzzifies the input sensor values using the fuzzifyInputs method. Then it iterates over all fuzzy rules, calculating the strength of each rule based on the degree of membership of its conditions. The output of each rule contributes to a priority bucket. A weighted average (crisp score) is calculated, and a priority label is assigned based on predefined thresholds.

FuzzyEngine.fuzzifyInputs

This private method maps each numeric sensor value to a fuzzy membership distribution using the corresponding FLV. It returns a dictionary of sensor names to fuzzy label degrees (e.g., { temperature: { Low: 0.1, Medium: 0.8, High: 0.1 } }). This fuzzified input is used during rule evaluation.

FLV.GetDoM

This method returns the Degree of Membership for a given x-value. If the value exists in the FLV's internal map, it returns the exact membership values. If it falls between two known values, the method performs linear interpolation to approximate the membership for each label.

FLV.AddValuesForLabels

Used during FLV construction, this method fills the internal data structure with predefined membership values for each x-label pair. It validates that all labels exist in the FLV and that the x-values are within range. This forms the basis for fuzzification.

MembershipFunctions.Triangular / LeftShoulder / RightShoulder

These functions generate membership values for FLVs. Each function maps a set of x-values to a membership curve. Triangular defines a peak at a center point with slopes on each side. LeftShoulder has a flat high value on the left and slopes down to zero. RightShoulder is the mirror of LeftShoulder.

createEngineFor

This factory method initializes a FuzzyEngine for a specific machine. It stores the engine and its associated sensor list in the global machineConfig map. This map is used later by the evaluation pipeline to determine when all required sensor data has arrived.

MaintenanceService.getEvaluations

This method manages the SSE stream. It subscribes to MQTT topic # using the MQTT service and listens for raw sensor messages. Each message is parsed and added to an internal list of per-machine sensor sets. When all required sensors for a machine are available, it triggers evaluation via the FuzzyService, sorts the results by score, updates the cached evaluation list, and emits the results to all SSE clients.

MaintenanceService.parseSensorData

Parses a raw JSON string into a partial SensorData object. It validates that the payload contains a valid machineId, supported sensorType, and a numeric value. If valid, it maps the value to the correct sensor field. Otherwise, it logs a warning and returns null.

MaintenanceService.isValidSensorPayload

Performs strict type and field validation for incoming MQTT messages. It ensures that the payload has all required fields (machineId, sensorType, value) and that they are of the correct types. The message should have the following structure and returns it that way if it is valid:

```

    isValidSensorPayload()

    return (
        typeof obj.machineId === 'string' &&
        typeof obj.sensorType === 'string' &&
        typeof obj.value === 'number'
    );

```

2.4.4. Machine Detection

The Machine Detection module is responsible for identifying physical machines in the environment using visual markers and Unity's ARFoundation framework. It is a key part of the AR frontend, allowing the system to link real-world objects to their corresponding digital data by recognizing pre-defined images. It also gives the ability to link machine in world-space.

Originally, OpenCV-based marker detection was considered due to its flexibility and accuracy. However, we were unable to get it working within an AR context in Unity. Furthermore, support for AR integration in OpenCV requires a paid license, which made it unsuitable for this project. As a result, the implementation uses Unity's built-in ARTrackedImageManager with a static Image Reference Library.

Each machine has a corresponding visual marker (e.g. a AprilTag), which is included in the reference library and assigned a unique index between 0 and 30. When the AR camera recognizes a marker image, the system resolves it to a machine ID using the name of the tracked reference image.

More information about this module found in Machine detection

Structure

The Machine Detection module is implemented as a set of Unity MonoBehaviour scripts organized around ARFoundation's image tracking system. The main entry point is the ImageTrackingManager, which lives in the Unity scene and orchestrates detection, creation, and lifecycle management of tracked images. When a tracked image is recognized, it is associated with a TrackedImageController, which configures the UI, data subscriptions, and appearance of the overlay.

The system is initialized by the ARReadinessManager, which ensures stable AR tracking before activating the marker scanner. Once the AR session is deemed ready, it enables image tracking and hands off control to the detection pipeline.

Below are the relevant files and their responsibilities:

- **ARReadinessManager.cs:** Manages startup readiness. Displays a progress bar until the AR session state reaches SessionTracking and holds stable for a short time. Once ready, it activates the marker detection system.
- **ImageTrackingManager.cs:** Subscribes to ARTrackedImageManager.trackedImagesChanged. Instantiates a prefab for each detected image and manages updates or removals. Ensures a new controller is created for each newly recognized image and that updates are passed through.
- **TrackedImageController.cs:** Controls a single tracked image prefab. Initializes the machine panel, requests machine metadata from the backend, subscribes to MQTT or maintenance evaluation endpoints based on marker ID, and handles positioning and rotation relative to the camera.
- **PanelManager.cs:** Manages the sensor or fuzzy logic data UI for the detected machine. Subscribes to SSE streams, updates the textual display, and interacts with SensorGraphController.

- **SensorGraphController.cs:** Renders real-time graph data per sensor. Maintains line renderers, reference lines, colored dots, and legends for clarity and traceability.

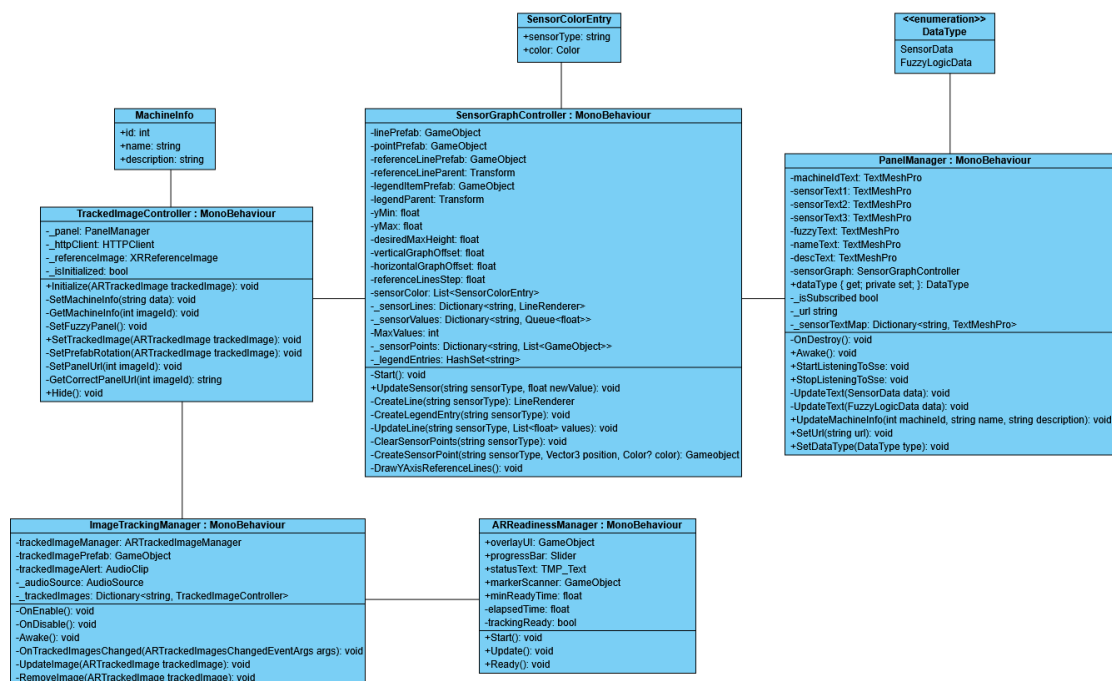
This architecture separates concerns cleanly: one script handles tracking, another handles per-machine logic, and a third manages UI. Machine detection scales naturally by adding new images to the Image Reference Library and naming them to match machine IDs. No code changes are needed to support new machines.

Code

The Machine Detection module is implemented using Unity MonoBehaviours and leverages the ARFoundation framework to detect machine markers and overlay context-aware information. The flow begins with the AR session initialization and readiness check (ARReadinessManager). Once tracking is stable, the ImageTrackingManager activates and begins listening for recognized images using the ARTrackedImageManager.

When a marker is detected, the system instantiates the ImageTracking_Panel prefab at the marker's location. The associated TrackedImageController initializes data subscriptions, UI panels, and layout. It resolves the image name (used as the machine ID), requests metadata via HTTP, and begins streaming live data (sensor or fuzzy) using Server-Sent Events (SSE).

Sensor data is rendered dynamically through the PanelManager and SensorGraphController, which update textual and graphical UI elements in real time. Each detected machine is handled independently through its own instance of the prefab and controller.



Code across this module is split into clearly defined responsibilities:

- tracking logic (ImageTrackingManager)
- per-image behavior (TrackedImageController)
- UI panel control (PanelManager)
- real-time graph rendering (SensorGraphController)
- startup readiness gating (ARReadinessManager)

This separation allows the system to scale easily by adding new markers or UI panels without coupling logic across components.

Method Overview

ARReadinessManager.Start

This method disables device sleep and prepares the AR readiness UI. It disables the marker scanner initially and shows a progress bar with a status message indicating that AR is initializing.

ARReadinessManager.Update

This method monitors the AR session state and tracks time once the session is stable (SessionTracking). If stability is maintained for the configured minReadyTime, it calls Ready(). Otherwise, it resets the timer and updates the UI with appropriate feedback.

ARReadinessManager.Ready

Once AR tracking is considered stable, this method hides the readiness UI and activates the image tracking system (markerScanner). On Android, it triggers a short vibration to signal readiness.

ImageTrackingManager.OnEnable / OnDisable

These lifecycle methods subscribe and unsubscribe to the trackedImagesChanged event of the ARTrackedImageManager. This ensures image tracking events are handled while the object is active.

ImageTrackingManager.Awake

Initializes an AudioSource to play a notification sound when a new tracked image is detected. This sound is configured via the trackedImageAlert field.

ImageTrackingManager.OnTrackedImagesChanged

Called automatically when tracked images are added, updated, or removed. For each added image, it instantiates and initializes a prefab. For updated images, it refreshes position and state. For removed images, it hides the corresponding overlay.

ImageTrackingManager.UpdateImage

If a tracked image is not yet managed, this method instantiates the prefab at the correct position and rotation, initializes it using TrackedImageController.Initialize, plays the alert sound, and stores the reference. If already tracked, it calls SetTrackedImage to update it.

ImageTrackingManager.RemoveImage

This method hides the UI and unsubscribes from SSE if a tracked image is no longer detected. It delegates to the tracked image's controller via controller.Hide().

TrackedImageController.Initialize

Initializes the controller with a new ARTrackedImage. It sets up internal references, determines the machine ID by parsing the reference image name, configures the data panel URL and type, and fetches machine metadata.

TrackedImageController.SetTrackedImage

Positions the UI prefab above the detected marker. Adjusts its height based on camera position and applies a smooth facing rotation using SetPrefabRotation.

TrackedImageController.SetPrefabRotation

Calculates the direction from the image to the AR camera and adjusts the prefab's Y-axis rotation to face the user. If the camera is unavailable, it falls back to the image's own rotation.

TrackedImageController.SetPanelUrl

Determines and sets the correct SSE endpoint for the current panel, based on the machine ID and selected data type.

TrackedImageController.GetCorrectPanelUrl

Returns the appropriate SSE URL for either sensor data or fuzzy logic data. For sensor data, the machine ID is embedded in the URL. For fuzzy data, it returns the shared evaluation endpoint.

TrackedImageController.GetMachineInfo

Performs an HTTP GET request to fetch the name and description of the machine by ID. The response is deserialized and forwarded to the panel.

TrackedImageController.SetMachineInfo

Parses the machine metadata JSON and forwards the ID, name, and description to the PanelManager for display.

TrackedImageController.SetFuzzyPanel

Switches the panel to fuzzy logic mode. Hides the sensor graph and repositions the data panel. Used for a special case when image ID 0 is detected.

TrackedImageController.Hide

Deactivates the tracked image prefab and unsubscribes from all SSE events by clearing the panel URL and calling StopListeningToSse.

2.4.5. Storage API

The Storage Module is responsible for collecting and storing real-time data from machine-mounted sensors via an MQTT broker. It plays a critical role in the system backend by capturing sensor data and persisting it in a PostgreSQL time-series database for further analysis, visualization, or diagnostics.

The module is designed using a modular, singleton-based Node.js architecture.

Structure

The module is composed of three core components:

- **MQTT.ts**
Handles MQTT client connection and message subscriptions. It listens for all messages on the broker and routes incoming sensor data for processing.
- **Database.ts**
Manages the PostgreSQL database connection using a connection pool. Provides an interface to insert sensor readings into the database.
- **index.ts**
Acts as the application entry point. It initializes the configuration, MQTT client, and database connection, and binds the MQTT message handling logic to the data storage pipeline.

All components are implemented using the singleton pattern to avoid multiple instances and ensure consistent behavior across the application lifecycle.

Functional Responsibilities

Sensor Data Ingestion

Listens to MQTT messages from machine sensors.

Data Parsing

Extracts structured information from the message payload.

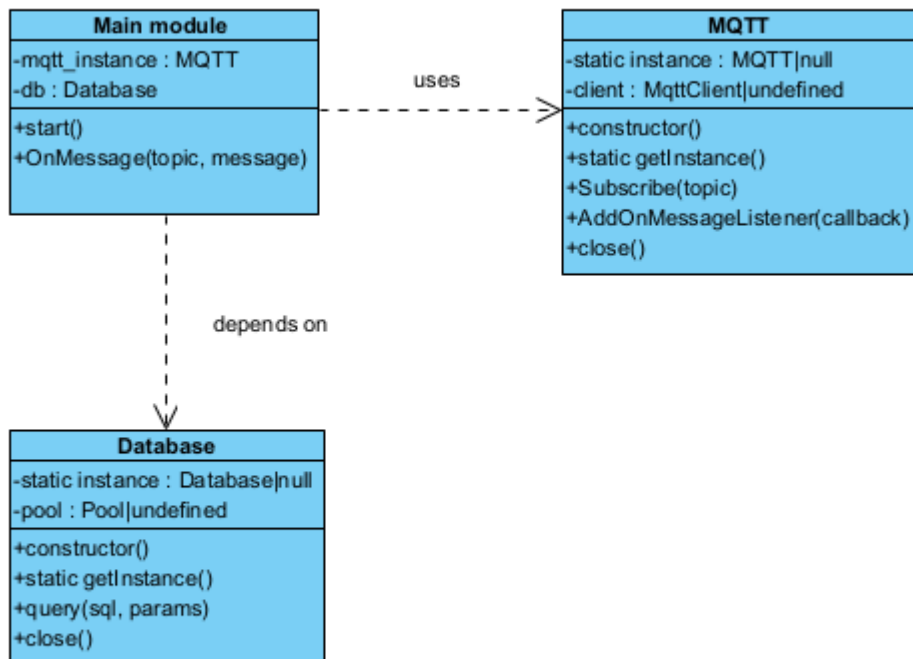
Persistent Storage

Inserts sensor values into a PostgreSQL database.

System Coordination

Initializes and manages the lifecycle of MQTT and database modules.

Code



Method Overview

MQTT.getInstance

Returns the singleton MQTT instance, creating it if necessary. Ensures only one connection to the MQTT broker exists during runtime.

MQTT.constructor

Connects to the MQTT broker using the configured environment variable `MQTT_HOST_IP`. If an instance already exists, it returns the existing one (singleton pattern).

MQTT.Subscribe(topic)

Subscribes the MQTT client to the given topic. Resolves to true if successful, or rejects with an error if the subscription fails. Logs the subscription result.

MQTT.AddOnMessageListener(callback)

Registers a message listener that gets called whenever a message is received on a subscribed topic.

MQTT.close

Disconnects the MQTT client from the broker and logs the disconnection.

Database.getInstance

Returns the singleton Database instance. If none exists, it creates one using the PostgreSQL connection string from the `TIMESERIES_DATABASE_URL` environment variable.

Database.constructor

Initializes the PostgreSQL connection pool. If environment variables are missing, it throws an error. Returns the singleton if one already exists.

Database.query(sql, params)

Executes an SQL query using the PostgreSQL pool. Accepts a SQL string and an array of parameters. Throws an error if the pool is not initialized.

Database.close

Closes the PostgreSQL pool connection and resets the singleton instance.

start()

Entry point function.

Subscribes to the MQTT topic (#), and registers the OnMessage callback to handle incoming MQTT messages.

OnMessage(topic, message)

Triggered when an MQTT message is received. Parses the message payload as JSON and inserts the data into the sensor_data table using the database connection.

process.on("SIGINT")

Handles graceful shutdown on process termination (e.g., Ctrl+C). Closes the MQTT and DB connections before exiting.

3. Software Architecture

This chapter outlines the core architectural structure, detailing how backend and frontend components interact to support real-time navigation, sensor monitoring and maintenance guidance.

The backend, built with NestJS, manages pathfinding, MQTT integration, fuzzy logic evaluation and database communication. A separate Storage API handles high-frequency sensor ingestion into TimeScaleDB. The frontend, built in Unity using AR Foundation, focuses on AR visualization, marker detection, and navigation rendering.

Each module is described in terms of its responsibilities, internal structure and role within the larger system. Together, they form a modular, scalable and maintainable architecture designed for industrial AR environments.

3.1. MQTT Module

3.1.1. Communication


The MQTT module acts as a bridge between the MQTT broker and HTTP clients by subscribing to specific topics on the broker using the MQTT protocol. As data is published to these topics, typically by IoT devices or sensors, the module receives and processes the incoming messages in real time. It then forwards this data to connected web clients through a dedicated HTTP endpoint, utilizing the Server-Sent Events (SSE) protocol to establish a continuous, one-way data stream. This approach allows browser-based applications or other SSE-compatible clients to receive live updates without polling, while decoupling the MQTT infrastructure from the front-end layer for better modularity and interoperability.

3.1.2. Dependencies

The MQTT module relies on several key libraries and external components to function correctly within the system. These include:

- mqtt (via mqtt.js), which is used to establish and maintain the connection to the MQTT broker and to subscribe to specific sensor topics.
- rxjs, which enables reactive data streams by using Subject and Observable. These streams form the basis for delivering live updates via Server-Sent Events (SSE).
- @nestjs/common and @nestjs/core, which provide the standard structure and dependency injection mechanisms used throughout the NestJS framework.
- @nestjs/swagger, which is used to automatically generate OpenAPI documentation for all MQTT-related endpoints.
- The Unity frontend, which connects to the backend via SSE to display real-time sensor overlays in the AR view. This module provides the necessary endpoints and streaming interfaces.
- Future backend modules such as Maintenance Detection and Route Planning, which will use this module to get real-time sensor data for further analysis and decision-making.
- dotenv, which is used to load environment variables from configuration files. These variables configure the connection settings for the MQTT broker and the API server. An example configuration file is shown below.

The following is a typical ``.env.development`` file:



```
MQTT_HOST_IP="127.0.0.1"
HOST_PORT="3000"
```

3.2. Navigation Module

The Navigation Module is a coordinated component present on both the frontend and backend, responsible for managing all logic related to route navigation. On the backend, it maintains a static graph representing the navigable space, typically a network of nodes and edges describing locations and connections within a building or environment. When the frontend requires navigation, it queries the backend with a start and end point. The backend then applies the A* pathfinding algorithm to compute the optimal route through the graph and returns this path as a sequence of node identifiers or coordinates.

On the frontend, the Navigation Module receives this path and handles its traversal step-by-step. It interacts with components such as the location tracker to determine the user's current position, and the path visualizer to render guidance (e.g., arrows or lines) to the next node in the path. As the user progresses and reaches each node, the frontend updates the navigation state and continues rendering directions until the destination is reached. This separation of responsibilities ensures that computationally intensive tasks like pathfinding remain server-side, while the client focuses on user interaction and real-time feedback.

3.2.1. Dependencies

Backend

The backend implementation of the NavigationModule relies on several key dependencies to operate effectively within the NestJS ecosystem:

- `@nestjs/common` and `@nestjs/core`: These foundational NestJS packages provide the core application structure, including decorators, dependency injection, and lifecycle hooks.
- `@nestjs/swagger`: This module is used to generate OpenAPI documentation automatically, allowing for easy inspection and testing of navigation-related endpoints.
- **Modelspace coordinate requests**: The backend expects coordinate data from the Unity frontend, which is used to determine the starting point for the pathfinding algorithm.
- **Prebuilt graph**: A static graph representing the spatial layout of nodes is stored in the database. This graph is loaded and used for path calculations using the A* algorithm.
- `dotenv`: Environment variables, such as the database connection string, are managed securely using the `dotenv` package.
- **Prisma with PrismaClient**: an object relational mapper that handles the connection with the database.

Frontend

The frontend portion of the NavigationModule operates within a Unity-based AR environment and relies on the following components:

- **Image Recognition Module:** This module is used to detect reference points (e.g., AprilTags), which help align and anchor the graph data to the physical world by determining scale and origin.
- **HTTP connection:** The frontend communicates with the backend over HTTP to request the full graph, path data, and individual node positions required for visualization.
- **(Unity/AR dependencies):** While not specific to the module, the Unity engine and AR libraries (such as AR Foundation) are essential for rendering and placing the navigation cues in the physical space.

This modular and well-defined dependency structure ensures that both backend and frontend can evolve independently while maintaining robust coordination for accurate navigation.

3.3. Maintenance Detection Module

The Maintenance Detection module adds a layer of smart decision-making to the AR Assistant by using fuzzy logic to evaluate sensor data and prioritize maintenance needs. It builds on the existing MQTT system to get real-time sensor values and uses machine-specific config files to determine how that data should be interpreted. Each machine gets its own fuzzy logic setup, making the system flexible and easy to extend without touching the core logic. This section covers the dependencies, configuration structure, and how the module integrates into the backend architecture.

3.3.1. Dependencies

The Maintenance Detection Module builds on several existing backend components, most notably the MQTT module, and introduces its own fuzzy logic system for maintenance prioritization. Below are the key dependencies specific to this module:

- **fs** and **path** (Node.js core modules): Used to load machine-specific fuzzy logic configurations (*.config.json) from the filesystem during application startup.
- **MQTT Module (see MQTT Module section)**: Provides the reactive data stream of live sensor data. The `getLiveUpdates` method is used to receive real-time sensor messages via MQTT topics.

All standard NestJS framework dependencies such as `@nestjs/common`, as well as the core MQTT and SSE infrastructure, are shared with and already described in the MQTT Module section.

3.3.2. Machine Configuration

Each machine in the Maintenance Detection system is configured using a dedicated JSON file located in the `config/fuzzy/` directory. These configuration files define the machine-specific fuzzy logic evaluation setup, including which sensors are expected and which rules should be applied to determine the maintenance priority.

A configuration file consists of three key fields:

- **machineId**: A string identifier that uniquely distinguishes the machine within the system.
- **sensors**: An array of sensor names (e.g., temperature, pressure, vibration) that the fuzzy logic engine expects to receive before performing an evaluation.
- **rules**: A list of fuzzy rules, where each rule contains a set of conditions (sensor label mappings) and an output priority label (Low, Medium, High, or VeryHigh).

During service startup, the system loads all *.config.json files from the configuration directory. Each file is parsed and used to initialize a FuzzyEngine instance for the corresponding machine.

This configuration-driven approach allows the system to be easily extended. New machines can be supported by simply dropping in a new JSON file, no code changes are required. Similarly, adjusting sensor expectations or rule behavior for an existing machine only requires updating its config file. New sensor types must be defined in the FLV templates within `fuzzy.service.ts`. To change membership shapes or ranges, modify the parameters passed to the membership functions. No controller or service logic needs to be altered when extending the system using this configuration-first approach.

Example configuration (`machine1.config.json`):



```
machine1.config.json
{
  "machineId": "1",
  "sensors": ["temperature", "pressure"],
  "rules": [
    {
      "conditions": { "temperature": "High", "pressure": "High" },
      "output": "VeryHigh"
    },
    {
      "conditions": { "temperature": "Low", "pressure": "Low" },
      "output": "Low"
    }
  ]
}
```

This file configures a fuzzy engine for machine 1, instructing the system to wait for both temperature and pressure values before evaluating.

3.4. Machine detection

The Machine Detection module is responsible for recognizing machines in the AR view using printed markers. It uses Unity's ARFoundation system to detect and track these markers in real time and links them to machine-specific UI and data. This part of the app ties together image tracking, runtime prefab spawning, and SSE-driven data updates. In this section, we'll go over the dependencies, how the system is set up in the Unity scene, which prefabs are involved, and how image tracking works under the hood.

3.4.1. Dependencies

The Machine Detection module relies on several Unity packages and components to provide reliable and performant image tracking within the AR environment. These dependencies form the backbone of the image-based recognition system that drives machine detection.

- **ARFoundation:** Unity's cross-platform framework for AR development. Provides a high-level abstraction over ARKit (iOS) and ARCore (Android), and enables access to AR camera feeds and tracking data.
- **ARTrackedImageManager:** A Unity component that manages detection and tracking of reference images in real time. It handles lifecycle events such as when an image is found, updated, or removed from view. This is the core component responsible for recognizing machine markers in the camera feed.
- **XRIImageTrackingSubsystem:** Provided indirectly by the ARFoundation backend (ARKit/ARCore), this subsystem performs the actual low-level image recognition and tracking.
- **Image Reference Library:** A manually curated asset containing all registered machine markers. Each image is imported into Unity and marked as a reference image with a name corresponding to the machine ID (e.g. machine_0, machine_1, etc.). This library is assigned to the ARTrackedImageManager.
- **UnityEngine.XR.ARSubsystems:** Contains base types and interfaces used by ARFoundation, including support for tracked images and session management.

These dependencies are integrated into the Unity scene and communicate via events or direct method calls to the rest of the AR Assistant system.

3.4.2. Unity scene and prefabs

The Machine Detection system is embedded in the main AR scene (main.scene.unity), where it manages both recognition of machine markers and UI activation. All logic is encapsulated in components attached to GameObjects or prefabs, ensuring modularity and reuse across different machines or views.

3.4.3. Scene Hierarchy

The main scene contains the following key GameObjects relevant to this module:

- **AR Session Origin:** Contains the **ARTrackedImageManager**, which drives image recognition and tracking.
- **ImageTrackingManager:** Handles the core logic for managing tracked images and instantiating prefabs.
- **ARReadinessManager:** Displays readiness UI and controls when the tracking system becomes active.
- **ReferenceLineContainer:** Holds Y-axis reference lines for graphs. Automatically populated at runtime by **SensorGraphController**.

Key Prefabs

- **ImageTracking_Panel.prefab:** The primary overlay prefab instantiated for each detected image. It contains the **TrackedImageController**, **PanelManager**, **SensorGraphController**, and supporting UI components. This prefab is positioned above the marker image in the AR view.
- **LineRenderer.prefab:** A visual line renderer used to display the sensor's historical values on a graph. Each sensor gets a dedicated line initialized by the **SensorGraphController**.
- **GraphPoint.prefab:** A visual point placed on the graph for each sensor data entry. Points are updated in sync with the corresponding line segments and colored per sensor type.
- **LegendItem.prefab:** UI element showing sensor name and color in the graph's legend. Instantiated once per sensor type as new data is received.
- **ReferenceLine.prefab:** Horizontal guide lines rendered on the Y-axis of the graph at fixed intervals. Labeled with numeric values. These lines are dynamically generated at runtime.

These prefabs are linked into the main system through the **ImageTrackingManager** and the components instantiated within **TrackedImageController**. By combining AR tracking with runtime prefab instantiation and SSE-driven data rendering, the system provides responsive, context-aware visual feedback directly over the recognized machine.

3.4.4. Image Tracking

Machine detection is performed using ARFoundation's image tracking system. Each physical machine is assigned a printed marker image, which is included in the Unity project's Image Reference Library. This library is imported and configured via the `ARTrackedImageManager`, which is active in the main scene under the AR Session Origin.

Each reference image in the library must have a unique and meaningful name. In this system, the image name directly encodes the machine's ID. For example, an image named "3" will trigger recognition as machine ID 3. This convention eliminates the need for external lookup tables or manual mapping logic.

The detection lifecycle works as follows:

1. The AR session begins, and `ARReadinessManager` waits until tracking is stable.
2. Once ready, `ImageTrackingManager` subscribes to the `trackedImagesChanged` event from `ARTrackedImageManager`.
3. When a new image is recognized, the manager instantiates the `ImageTracking_Panel` prefab at the detected location.
4. The image's name (used as machine ID) is retrieved via `trackedImage.referenceImage.name`.
5. The prefab is initialized using the `TrackedImageController`, which configures rotation, requests machine metadata, and subscribes to the correct SSE endpoint.

Tracked images are also handled on update and removal:

- On update, the prefab position and data remain in sync with the tracked marker.
- On removal, the associated overlay is hidden, and SSE subscriptions are terminated.

This image tracking mechanism is fast, stable, and easily extensible. Adding support for a new machine simply requires importing a new marker image into the reference library and naming it with the correct ID. No code changes are required.

3.5. Storage API

The Storage API is a backend service that listens to sensor data sent over MQTT and saves it into a PostgreSQL (TimescaleDB) database. It's designed to run continuously in the background, acting as a bridge between the machine sensors and the database where all the data is stored.

When the service starts, it reads the necessary config from environment variables (like the MQTT broker IP and database URL), connects to the MQTT broker, and subscribes to all topics using #. It expects incoming messages to be in JSON format with fields like timestamp, machineId, sensorType, value, and unit. Once a message is received, it parses the JSON and inserts the data into a table called sensor_data.

The whole module is built using Node.js and follows a singleton pattern for both the MQTT and database connections, so there's only one active instance of each during runtime. If the app is stopped (for example, with Ctrl+C), it shuts down cleanly by closing the MQTT and database connections.

This service assumes that the MQTT broker and TimescaleDB (with the correct table setup) are already running and accessible.

3.5.1. Dependencies

The Storage API module relies on several backend technologies and libraries. It is implemented in Node.js and designed to run independently of the main NestJS backend.

- **Node.js:** Provides the runtime environment for executing the service. Node's event-driven architecture supports high-throughput, low-latency message handling, which is critical for real-time data ingestion.
- **mqtt (via mqtt.js):** Used to connect to the MQTT broker and subscribe to sensor topics. This library handles connection management, message reception, and reconnection logic in case of network failures.
- **pg:** Node's PostgreSQL client, used to connect to the TimescaleDB instance and perform insert operations.
- **dotenv:** Used to load configuration values (e.g., database URI, MQTT broker IP) from environment variable files.
- **TimescaleDB (PostgreSQL extension):** A time-series-optimized database backend used to store incoming sensor values. It is expected to be provisioned externally and configured with appropriate hypertables before the service is launched.

These dependencies are configured and orchestrated within a singleton-based structure. The MQTT and database modules each expose a single shared instance to avoid redundant connections and to maintain consistent behavior throughout the service's lifecycle.

3.5.2. Configuration & Environment Variables

Variable	Purpose
MQTT_HOST_IP	MQTT broker host IP
HOST_PORT	Port this app may expose (not currently used)
TIMESERIES_DATABASE_URL	PostgreSQL connection string

An example of a .env file based on these variables can be seen below:

```
MQTT_HOST_IP=127.0.0.1
TIMESERIES_DATABASE_URL=postgres://user:pass@host:5432/db
HOST_PORT=3000
```

3.6. SSE & HTTP Client

To facilitate communication between the frontend and backend systems, we implemented custom clients for both HTTP requests and Server-Sent Events (SSE) streaming. These clients abstract the low-level networking details and provide a clean, reusable interface for other components in the Unity application.

3.6.1. HTTP Client

The frontend includes a lightweight, reusable HTTP client implemented as a Unity MonoBehaviour. This client is designed specifically for performing HTTP GET requests, which are commonly used to retrieve static resources or query backend endpoints.

Key Features:

- **Simple API:** The client method accepts a target URL, an onSuccess callback, and an onError callback.
- **Asynchronous:** Uses Unity's coroutine system to perform non-blocking web requests.
- **Reusability:** Can be attached to any GameObject or used as a service component across different modules.

This design keeps HTTP logic centralized and decouples networking from domain-specific logic, improving maintainability and testability.

3.6.2. SSE Client

For real-time communication, we developed a custom SSE (Server-Sent Events) client based on a MonoBehaviour called SSEManager. This client enables persistent one-way data streaming from the backend to the frontend and is particularly useful for live sensor data, alerts, or other time-sensitive updates.

Design Overview:

- **Factory Pattern:** The SSEManager provides a factory-like method for creating custom SSE clients tied to specific data models.
- **Event-Driven Architecture:** Each SSE client emits parsed data as strongly typed events via the central SSEEventBus.
- **Flexible Subscription:** Any module within the Unity application can subscribe to the SSEEventBus to receive updates without direct dependency on the data source.

Benefits:

- **Loose coupling** between data producers (SSE clients) and consumers (UI, logic modules).
- **Modular design** enabling multiple parallel SSE streams if needed.
- **Strong typing** with generic model support for safe and structured data handling.

Example Flow:

1. The SSEManager creates an SSEClient<T> for a specific data model (e.g., SensorData).
2. The client connects to the backend SSE endpoint and begins receiving streamed events.
3. Upon receiving a new event, the client parses it and dispatches it through the SSEEventBus.
4. Subscribed modules (e.g., dashboard visualizations, loggers) handle the event accordingly.

This combination of a simple, coroutine-based HTTP client and a flexible, event-driven SSE system ensures that our frontend has robust, scalable access to both on-demand and real-time backend data.

3.6.3. Dependencies

- UnityEngine.Networking.UnityWebRequest: Used to send and receive http-data.
- UnityEngine.Coroutine: Required for non-blocking request execution.
- UnityEngine.MonoBehaviour: Base class for managing component lifecycle.
- System.Net.Http.HttpClient: Used to establish and read the stream from the SSE endpoint.
- System.Threading.Tasks: For async stream handling outside of Unity's coroutine system.
- UnityEngine.MonoBehaviour: Lifecycle management for SSEManager.
- Custom:
 - SSEClient<T>: Generic client handling deserialization and dispatching.
 - SSEEventBus<T>: Event hub for subscription and publishing.
 - SSEManager: Factory and lifecycle handler for multiple SSE-clients.
- An HTTP endpoint.
- An SSE endpoint.

3.7. Code Quality Practices

To ensure maintainable, scalable, and reliable software, our project adheres to a set of well-defined code quality practices. These practices are enforced throughout the entire development lifecycle, from initial design to production deployment. Key aspects include type safety, consistent code style, dependency management, and comprehensive documentation.

3.7.1. Type Safety with TypeScript

We used TypeScript as our primary programming language to benefit from static type checking. TypeScript enables developers to define explicit types for variables, function parameters, return values, and class properties, which prevents many common bugs at compile time. This improves overall code robustness, facilitates better IDE support (e.g., autocompletion and refactoring), and enhances long-term maintainability by making contracts between modules more explicit and predictable.

3.7.2. Code Consistency with ESLint

To maintain a consistent and clean codebase, we integrated ESLint, using the official NestJS ESLint configuration as our base. This linter configuration enforces strict coding conventions, helps avoid anti-patterns, and flags potential issues early in the development process. By integrating ESLint into our CI/CD pipeline and development workflow (e.g., pre-commit hooks), we ensure that every contribution adheres to the same quality standards.

3.7.3. Dependency Management and Singleton Pattern with @Injectable

Since our backend architecture is based on NestJS, we follow its modular and declarative design principles for managing dependencies. Specifically, we leverage NestJS's dependency injection system, where classes are annotated with the `@Injectable()` decorator. This signals to Nest's IoC (Inversion of Control) container that the class can be instantiated and injected as a singleton throughout the application.

This approach brings several benefits:

- **Loose coupling:** Components depend on abstractions rather than concrete implementations.
- **Reusability:** Services can be easily reused across modules without needing to manage their lifecycle manually.
- **Testability:** Dependencies can be mocked or overridden during unit testing, enabling isolated and reliable tests.
- **Scalability:** As the application grows, new services and modules can be introduced with minimal friction, and existing dependencies are automatically resolved.

By ensuring that all core services are injectable and singleton-scoped by default, we reduce resource overhead and promote a clean, consistent design across the backend.

3.7.4. Comprehensive API Documentation with Swagger

To document and expose our REST API effectively, we integrated Swagger using NestJS's built-in support via the `@nestjs/swagger` module. Swagger automatically generates an interactive OpenAPI specification from our controller definitions, which includes:

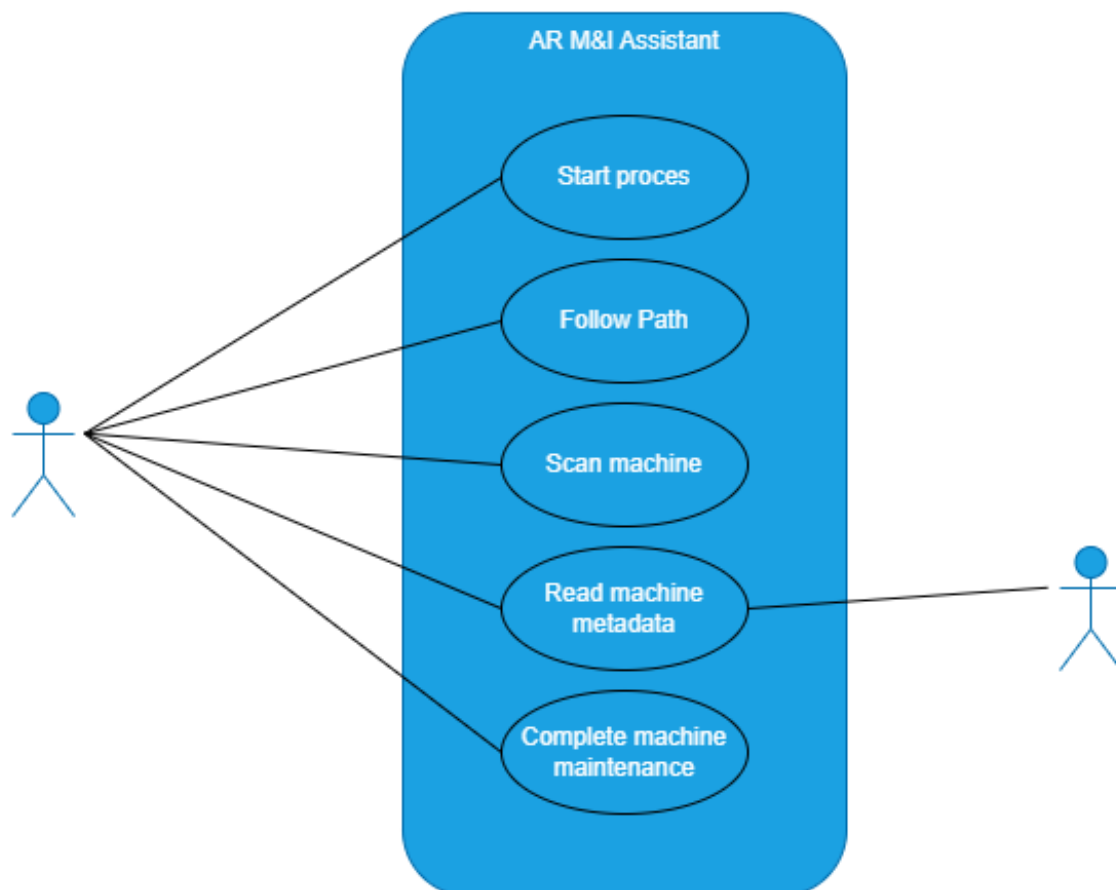
- Detailed descriptions for all endpoints, including query parameters, path variables, request bodies, and responses.
- Type-based schema generation, ensuring that the API documentation is always in sync with the actual codebase.
- Grouping and tagging of endpoints for easy navigation and readability.
- A live UI for developers and stakeholders to test endpoints directly from the browser.

This automated, always-up-to-date API documentation plays a critical role in onboarding new developers, enabling client-side integration, and improving collaboration between backend and frontend teams.

4. Functional Scope and User Interactions

This chapter outlines the core features of the product. It describes what the product is designed to do, how it supports technicians during their tasks, and the different ways it can be used in practice. We focus on the domain and use cases.

4.1. Use cases



Figuur 1 Use cases (complete machine maintenance not implemented)

4.1.1. Actors

Technician

A human user responsible for performing maintenance and inspections on machines using the AR app. The technician interacts directly with the AR interface to scan machines, follow navigation guidance, view machine data, and confirm maintenance tasks.

- Initiate the maintenance process via the AR interface
- Navigate to assigned machines
- Scan QR codes on machines
- Interpret metadata and sensor readings
- Act on maintenance recommendations
- Confirm task completion and status updates

Sensor (Secondary Actor)

An automated data source embedded in or attached to a machine. Sensors stream real-time data (e.g., temperature, vibration, pressure) to the API, enabling condition monitoring and fuzzy logic analysis.

- Continuously transmit live operational data to the API
- Provide input for fuzzy logic analysis to determine machine condition
- Support post-maintenance validation by reflecting status changes

4.1.2. Usecase Descriptions

Start Process

Title:	Start process
Actors:	Technician
Preconditions:	<ul style="list-style-type: none"> • The technician has launched the AR app on a supported device (AR glasses or phone). • The app is connected to the backend API. • The factory map and static data have been loaded from the API (within 2 seconds).
Main scenario:	<ol style="list-style-type: none"> 1. Technician opens the app and scans the origin point (marker with id "0"). 2. The app fetches the technician's real-time position using indoor positioning systems. 3. The app retrieves the prioritized machine maintenance list from the API. 4. The app requests an optimal path to the machines using the A* algorithm. 5. Visual AR markers for navigation are overlaid onto the physical environment.
Exceptions:	<ul style="list-style-type: none"> • <i>No connection to the API:</i> Show cached tasks if available, otherwise display error. • <i>Positioning failure:</i> App cannot determine location prompt user to move to another area.

Follow Path

Title:	Follow Path
Actors:	Technician
Preconditions:	<ul style="list-style-type: none"> • The technician has selected a target machine through the Start Process use case. • Path to the machine has been calculated and markers are ready for AR rendering.
Main scenario:	<ol style="list-style-type: none"> 1. The technician follows the AR visual markers on the floor or walls. 2. The app continuously updates the technician's location. 3. If the technician deviates from the path, the app recalculates the route in real time.
Exceptions:	<ul style="list-style-type: none"> • Location data lost or drifting: The app alerts the technician and pauses guidance. • Obstacles detected or destination changes: Path is recalculated dynamically using updated environmental data.

Scan Machine

Title:	Scan Machine
Actors:	Technician
Preconditions:	<ul style="list-style-type: none"> • Technician is physically near the target machine. • A visible AprilTag is present on the machine.
Main scenario:	<ol style="list-style-type: none"> 1. The technician points the device camera at the tag. 2. The app detects and decodes the tag. 3. The app identifies the machine using the decoded information. 4. The app fetches: <ul style="list-style-type: none"> ○ Static metadata about the machine. ○ Tag position. ○ Real-time sensor data. 5. The app renders the fetched data onto a panel in AR.
Exceptions:	<ul style="list-style-type: none"> • <i>Tag not recognized.</i>

Read Machine Metadata

Title:	Read Machine Metadata
Actors:	Technician, Sensor (Secondary Actor)
Preconditions:	<ul style="list-style-type: none"> • Tag has been scanned and machine identified. • Real-time sensor data is being streamed to the backend API.
Main scenario:	<ol style="list-style-type: none"> 1. The app fetches and displays: <ul style="list-style-type: none"> ○ Sensor readings (temperature, vibration, etc.) ○ Static metadata (name, description) 2. The result is shown to the technician with explanation in AR. 3. A linear graph is shown to view data “trends”.
Exceptions:	<ul style="list-style-type: none"> • <i>Sensor data unavailable:</i> Show warning and fallback to last known values.

Complete Machine Maintenance

*Due to different priorities this functionality has not yet been implemented.

Title:	Complete Machine Maintenance
Actors:	Technician
Preconditions:	<ul style="list-style-type: none"> Machine has been scanned and metadata reviewed. Condition analysis has been presented to the technician. Maintenance tools and access are available.
Main scenario:	<ol style="list-style-type: none"> The technician inspects and services the highlighted components. Real-time data is monitored to verify that values normalize. Technician confirms completion in the AR app. The app sends a status update to the API. API logs the maintenance event, timestamp, and final sensor values for analysis.
Exceptions:	<ul style="list-style-type: none"> <i>Sensor values remain abnormal:</i> Technician is warned and prompted to retry or escalate. <i>API submission fails:</i> Action is queued locally and re-attempted when online.

5. Data Structures and Relationships

5.1. Domain Model

The system provides real-time monitoring and maintenance support for industrial machines via an Augmented Reality (AR) interface. Each machine is physically tagged with a unique AprilTag, which serves both as a spatial anchor in the AR world and as a unique identifier within the digital system. Machines are equipped with one or more sensors that continuously report time-series data, enabling the detection of anomalies and decision-making for preventive maintenance.



5.1.1. Key Domain Entities

Below the key entities are described for the context of this project:

- **Marker:** Represents a physical AprilTag used in the real world to anchor a machine's digital twin in AR space. It provides a reference for AR world alignment and uniquely identifies a machine.
- **Machine:** Represents a physical machine being monitored. A machine has a unique marker associated to it. It also has several sensor associated with it and it is the core entity for maintenance logic and AR visualization.
- **Graph:** Represents the tree which can be navigated. Mimics the paths of the building of the factory.
- **Node:** Represents a point in the graph and a physical object used as anchor point for the navigation.
- **SensorData:** Represents a timestamped value emitted by a sensor. It forms a time series of measurements and enables monitoring, trend analysis, and predictive maintenance.

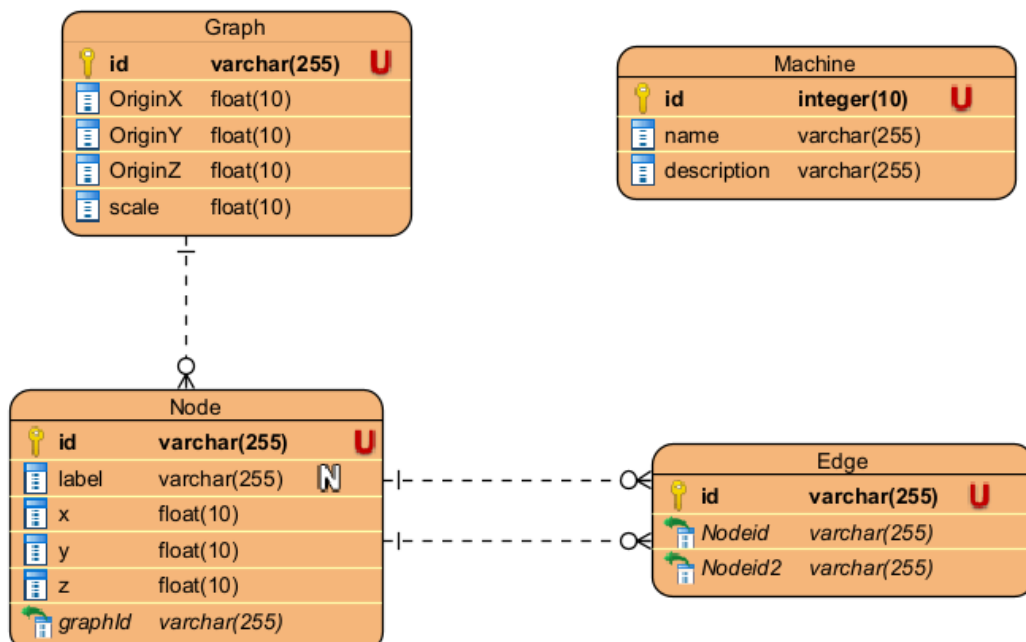
5.2. Databases

The application relies on a dual database structure to effectively manage both static factory information and rapidly updating real-time sensor data. To ensure optimal performance and scalability, we chose to separate the system's data handling into two distinct databases and APIs: one for static data and another for real-time telemetry. Real-time data, such as machine sensor values, can generate high-frequency traffic that puts strain on a traditional relational database. By isolating this load into a dedicated time-series database, we prevent performance degradation in critical areas like pathfinding and metadata queries. Additionally, this architectural split aligns with a clear separation of concerns.

5.2.1. Static data

For static data, such as machine metadata, spatial graph nodes, and factory layout information, the backend uses PostgreSQL, accessed via Prisma, a modern and type-safe Object-Relational Mapping (ORM) tool. The structure of this database is defined through a dedicated Prisma schema, which models entities like Machine, Graph, Node, and Edge. Prisma enables clean, structured queries and ensures tight integration with the TypeScript-based NestJS backend. Migrations and schema evolution are handled through Prisma's built-in tooling, ensuring consistency and ease of development.







Prisma connects to the database using a connection URL defined in the environment configuration, and the data model is defined in a "schema.prisma" file. This schema maps application-level entities to database tables. Prisma generates type-safe client code, which is used within the backend services to perform queries, mutations, and transactions efficiently.



5.2.2. Live telemetry data

For handling live telemetry, such as sensor values, status updates, and environmental data, the system uses a time-series-optimized PostgreSQL extension, TimescaleDB. This is connected separately from the static data layer and is accessed directly through SQL queries or a lightweight data access layer, depending on performance requirements. Incoming data is streamed into this database via an external API, and the backend is responsible for querying relevant time slices, aggregations, or thresholds when responding to frontend requests or evaluating fuzzy logic conditions. This separation of data domains allows the system to scale independently for long-term metadata storage and high-frequency live data processing.

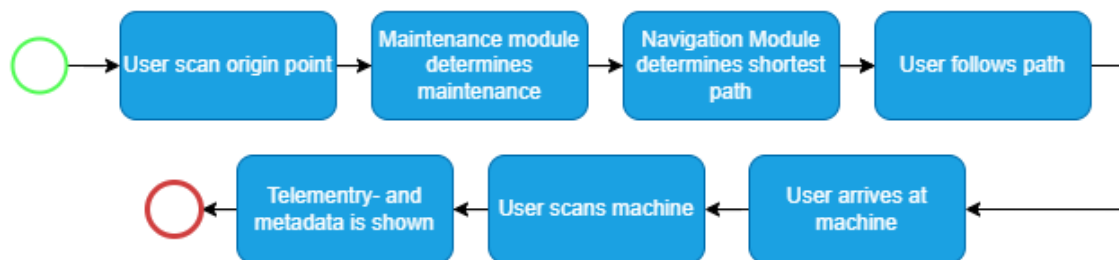
Unlike the static data layer, this connection is managed using raw SQL queries or lightweight query builders to avoid the overhead of an ORM and to better support high-throughput insertions and time-based aggregations. The backend ingests sensor data via an external API and writes it into hypertables designed for fast time-based access. Querying this data allows the system to evaluate trends, detect anomalies, or support fuzzy logic decisions based on recent machine behavior.

sensor_data		
	id	varchar(255) U
	machine_id	integer(10)
	sensor_type	varchar(255)
	value	double(10)
	unit	varchar(10)
	timestamp	timestamp(3)

6. User Flows and Feature Usage

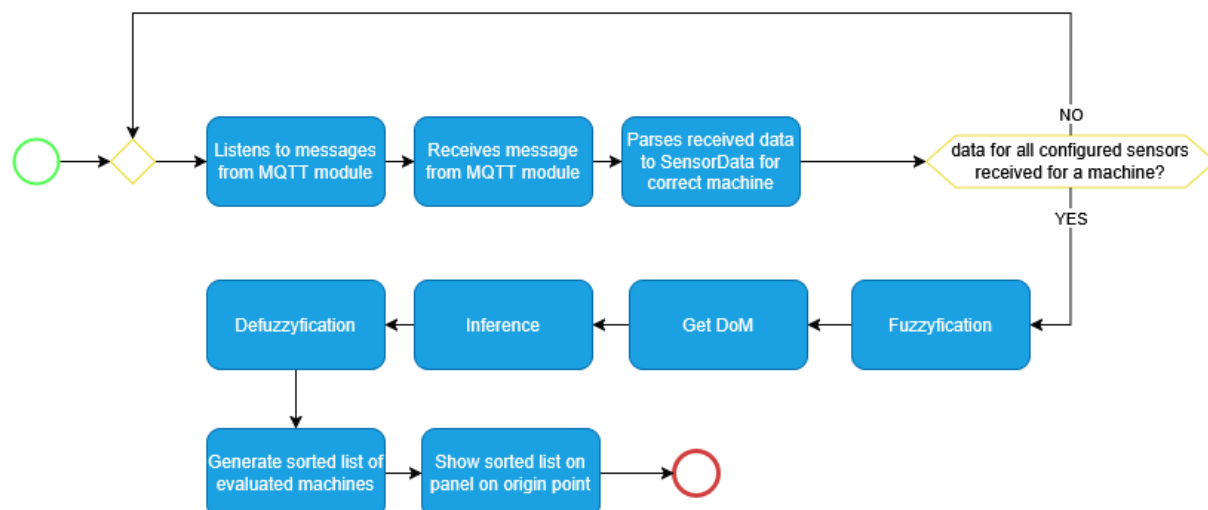
6.1. Entire app

Scan start point -> fuzzy logic determines maintenance -> Navigation module determines shortest path -> user follows shortest path -> user arrives at machine -> user scans machine -> machine telemetry data and metadata is shown.



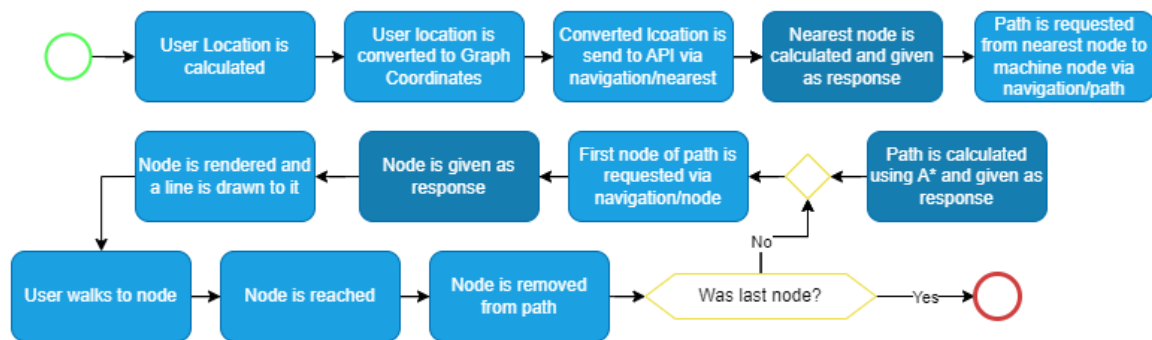
6.2. Maintenance detection Module

When the origin point is scanned -> the module will subscribe to all topics withing the MQTT module -> and then listen to all messages -> on message received the data will be parsed and added to the data of the corresponding machine -> when all configured sensor of a machine have received data -> evaluation will start otherwise it will keep listening -> then the step to perform fuzzy logic will be executed -> then a sorted priority list will be generated and shown.



6.3. Navigation Module

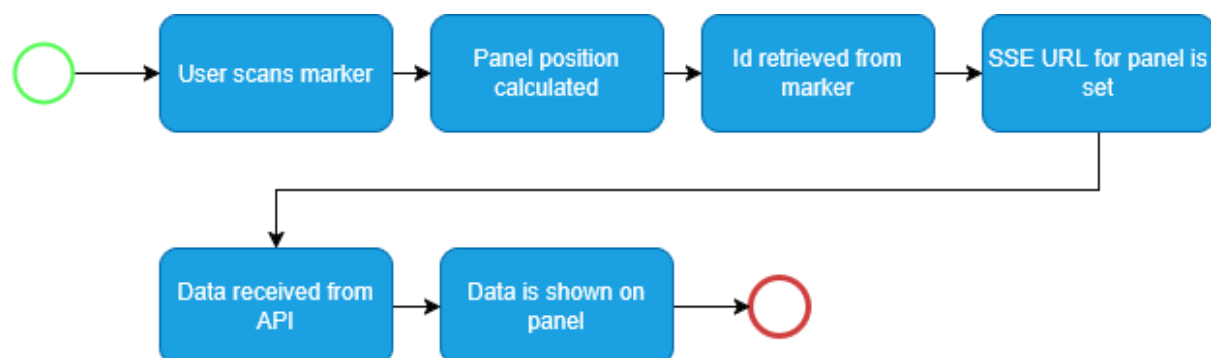
User location is calculated -> user location is converted to graphcoordinates -> user location is send to API via navigation/nearest endpoint -> nearest node is calculated and given as response -> a path is requested from nearest node to machine node via navigation/path -> path is calculated and given as response -> the first node of the path is requested via navigation/node -> the first node is rendered and a line is drawn to the node -> the user walks to the node -> the node is reached and the next node is fetched -> until final node is reached



Darker nodes are API and lighter are Frontend

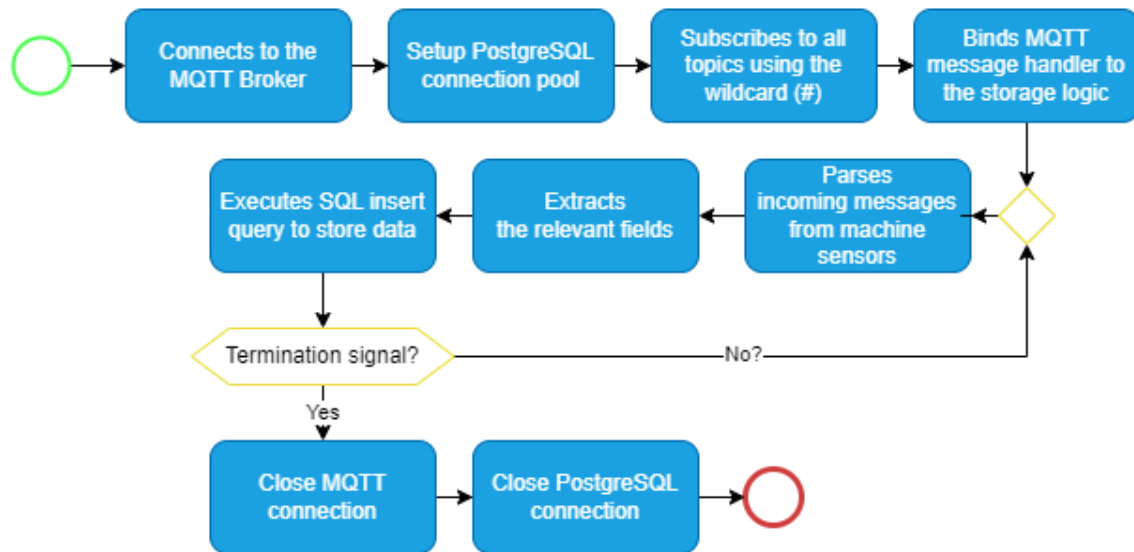
6.4. Visualization Module

When a user scans a machine marker -> first the position of the data panel will be determined - > then the ID is extracted from the marker -> the URL for the SSE endpoint for the data panel will then be set -> on data received the data will be shown on the panel



6.5. Storage API

Environment variables are loaded via dotenv -> connection to the MQTT broker is established using MQTT_HOST_IP -> PostgreSQL connection pool is initialized -> all MQTT topics are subscribed to using wildcard (“#”) -> MQTT message handler is bound to storage logic -> MQTT messages are received in JSON format -> messages are parsed and relevant fields are extracted (timestamp, machineId, sensorType, value, unit) -> data is inserted into the sensor_data table using a parameterized SQL query -> termination signal (SIGINT) is received -> MQTT and database connections are gracefully closed -> process shuts down.



7. API Interface Design

This API powers the backend of an AR-enabled industrial assistant system, providing real-time access to sensor data, spatial navigation, and machine maintenance evaluations. It is designed to integrate with a Unity-based frontend and supports both RESTful endpoints and Server-Sent Events (SSE) for continuous data streaming.

The API is organized into three main modules: MQTT for managing sensor data streams and broker connections, Navigation for spatial queries and pathfinding, and Maintenance Detection for streaming machine maintenance priorities based on fuzzy logic evaluation.

All endpoints are documented using Swagger/OpenAPI, with a live API reference available at <https://ar-mi-assistant.readme.io/>.

7.1. MQTT

Controllers/Endpoints

The MQTT module is structured around three dedicated controllers, each serving a specific purpose in relation to the real-time data flow. These controllers define the API endpoints that external systems or clients interact with to control the MQTT connection, manage topic subscriptions, and retrieve sensor data for debugging or monitoring.

MqttClientController

This controller is responsible for managing the lifecycle of the MQTT broker connection. It exposes endpoints that allow for manual connection, disconnection, and status checking of the MQTT client. These endpoints are not intended for production use but are valuable during development or debugging scenarios. They are typically used to confirm connectivity, re-establish a broken connection, or intentionally reset the client state.

MQTT Client Management

Method	Route	Purpose
GET	/mqtt/client/status	Returns current MQTT connection state
GET	/mqtt/client/connect	Reconnects the client to the MQTT broker
DELETE	/mqtt/client/disconnect	Forcefully disconnects from the broker

MqttSubscriptionsController

This controller provides the main interface for real-time communication between the backend and the Unity-based AR frontend. It allows frontend clients to subscribe to sensor data for specific machines using Server-Sent Events (SSE), and to unsubscribe when data is no longer needed. It also offers functionality to monitor the current subscription state and to subscribe to all topics using wildcard logic. This controller supports dynamic, per-session subscription management to ensure that only necessary data is streamed at any time.

MQTT Subscription Management

Method	Route	Purpose
GET	/mqtt/subscriptions/status	Lists all currently active subscriptions
GET	/mqtt/:machineId/:sensor	SSE stream of a sensor for a specific machine
GET	/mqtt/all	SSE stream of all MQTT messages
DELETE	/mqtt/:machineId/:sensor	Unsubscribe from a specific topic
DELETE	/mqtt/all	Unsubscribe from all topics

MqttDebugController

This controller is intended for development and backend testing. It provides endpoints to retrieve the most recent message received for a specific machine and sensor combination, or to render live data in a minimal HTML page for quick inspection. These endpoints are helpful for validating incoming messages, verifying topic correctness, and confirming end-to-end connectivity.

MQTT Debug and Testing

Method	Route	Purpose
GET	/mqtt/latest/:machineId/:sensor	Retrieve latest message snapshot
GET	/mqtt/debug/:machineId/:sensor	Render live data in a minimal HTML page

7.2. Navigation Module

Endpoints

The NavigationModule exposes a set of HTTP endpoints to interact with the graph and pathfinding system. These endpoints are defined in the navigation.controller.ts file and documented using Swagger decorators for automatic OpenAPI generation.

Navigation and Routing

Method	Route	Purpose
GET	/navigation/graph/:id	Retrieves a specific graph by its ID.
GET	/navigation/node/:nodeId	Fetches a single node by its unique ID.
GET	/navigation/path	Calculates the shortest path between two nodes using the A* algorithm.
GET	/navigation/nearest	Returns the nearest node to a given 3D coordinate.

Some of these endpoints require or accept specific properties which are documented below.

GET /navigation/graph/:id

Retrieves a specific graph by its ID.

- type: Optional. Set to 'deep' to include node data (default), or 'shallow' to exclude node data.

GET /navigation/path

- type: Optional. Path type ('deep' includes full node data, 'shallow' includes only IDs).

GET /navigation/nearest

Returns the nearest node to a given 3D coordinate.

- yThreshold: Optional. Filters nodes based on vertical distance.
- returnType: Optional. 'shallow' returns the node ID, 'deep' returns the full node object (defaults to 'shallow')

These endpoints allow the frontend to retrieve the graph, query specific nodes, calculate optimal paths, and find nearby nodes based on real-world position data. Each is documented for API consumers through Swagger, making integration and testing straightforward.

7.3. Maintenance detection module

Controllers/Endpoints

The Maintenance Detection module is structured around a single controller. This controller is the entry point to receiving the results of the fuzzy logic system. This controller defines the API endpoints that external systems or clients interact with to receive Maintenance Detection results.

MaintenanceController

The MaintenanceController serves as the primary entry point for external clients to access the results of the fuzzy logic evaluation system. Its main purpose is to expose evaluated machine maintenance priorities, either as a real-time stream or as an on-demand snapshot.

It is used by the Unity frontend to stream live priority evaluations.

Controller Endpoints

Method	Route	Purpose
GET	/maintenance/latest/evaluation	Returns the latest maintenance evaluations for all known machines.
GET	/maintenance/evaluate	Streams live evaluation results using Server-Sent Events (SSE). Clients stay connected and receive updates as new sensor data is processed.

7.4. Machine

Controllers/Endpoints

The Machine module is structured around a single controller. This controller is the entry point to receiving machine info.

MachineController

Provides external clients to the static information about the machines within the system.

The Unity frontend uses it to show the static machine data.

Controller Endpoints

Method	Route	Purpose
GET	/machines	Responds with a list of all machines, with their static info.
GET	/machines/{id}	Get static machine info for the chosen id.
GET	/machines/{id}/node	Responds with the node id that is linked to the machine.

8. Real-Time Data Handling

8.1. MQTT Topic

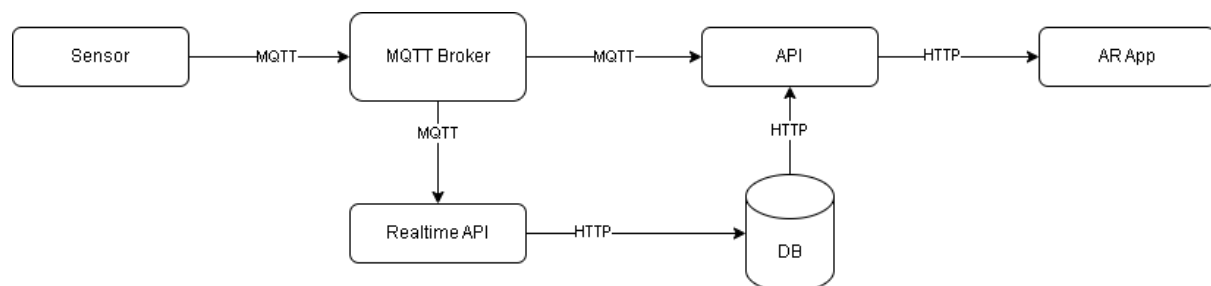
The industry standard for topics in MQTT is **site/building/machine/sensor**. For this project, the structure **Machine/Sensor** will be used instead. The data that the sensors transfer follows the following format:

```
MQTT Sensor Data

{
  "machineId": 1,
  "sensorId": 1,
  "sensorType": "Temperature|Pressure|Vibration",
  "value": 22.24,
  "unit": "°C",
  "timestamp": "2025-05-07T16:25:00+02:00"
}
```

8.2. MQTT Flow

Real-time data follows a certain path during its life. It starts at the sensor that produces the data and sends it to the MQTT broker. This broker then sends it to all the subscribers of that topic, in our case, this is the real-time API and the API. The real-time API stores this data in the database so that the API can fetch it later to gather historic data. The API receives the data and forwards it to the AR App using Server-Sent Events. This life is also depicted in the diagram below.



8.3. Server Send Events

To enable real-time updates of sensor data (e.g., temperature), the Unity frontend integrates Server-Sent Events (SSE) using a custom SseClient. This client establishes a persistent HTTP connection to the backend API, which streams updates using the text/event-stream format. The connection is managed via

UnityWebRequest, with a custom SseDownloadHandler (extending DownloadHandlerScript) that processes the continuous byte stream line-by-line, respecting the SSE protocol's newline-delimited format. Incoming data: lines are aggregated, and once a full message is received (indicated by a blank line), it is parsed into a SensorData object using Unity's JsonUtility.

```
SseClient.cs

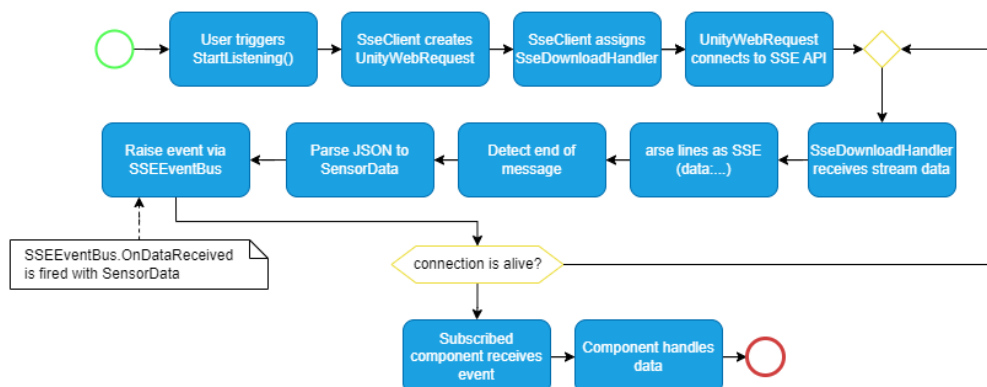
public class SensorData
{
    public string machineId;
    public int sensorId;
    public string sensorType;
    public float value;
    public string unit;
    public string timestamp;
}
```

Messages are expected in the format [channel] {JSON}, allowing channel-specific routing or filtering if needed. After successful parsing, the client dispatches the data through a decoupled event system using SSEEEventBus, which broadcasts the SensorData object via OnDataReceived. Any system component, such as a UI element, can subscribe to this event to react to new data.

This modular design separates the networking concerns (connection and parsing) from application logic (display and processing), enabling easy maintenance and extension. For example, components can start or stop listening dynamically, and multiple listeners can react to the same data without tight coupling.

```
Example.cs

SSEEEventBus.OnDataReceived += data => UpdateDisplay(data);
```



This setup supports stable, extensible architecture for real-time communication and is suitable for features like live machine monitoring, warnings, or interactive diagnostics. Future improvements may include reconnection strategies, support for more SSE fields (e.g., event:, id:), and message type dispatching based on custom logic.

9. Infrastructure and Deployment

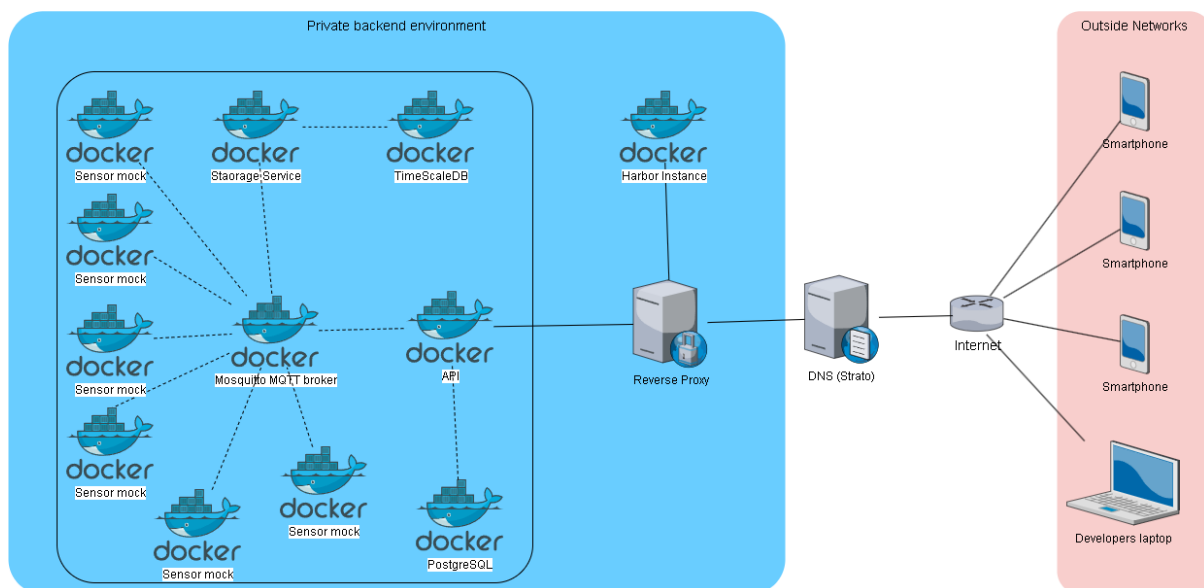
The deployment infrastructure for this system follows a locally hosted, containerized architecture managed via Docker Compose. All services, including APIs, messaging infrastructure, and databases, are deployed on a dedicated machine that functions as the operational backend environment. This approach prioritizes local control and privacy while offering selected remote accessibility.

9.1. Hosting and Orchestration

The system is orchestrated through a single 'compose.yaml' file that defines all the required services:

- **API Service:** Exposes the main application via port 3000 internally
- **Storage service:** Separately ingests and stores sensor data in the timescale database.
- **Databases:**
 - **PostgreSQL** for relational and static data.
 - **TimescaleDB** for high-frequency sensor data.
- **Mosquitto MQTT broker:** manages all the real-time sensor communication.
- **Mocked Sensor services:** six containers simulating sensor data.

Each service is isolated using containers, and the databases have volumes to facilitate data persistence. The Mosquitto container also has a mounted volume to configure it using a local config file.



9.2. Domain and Access Configuration

The domain **arassistant.nl** is used to access the API remotely. Strato handles this domain. A DNS record is used to point to the public IP for the backend environment. At the backend environment, a reverse proxy (Nginx Proxy Manager) is used to forward the connection (port 443 and domain **api.arassistant.nl**) to the Docker containers' IP and port (in this case, port 3000). The reverse proxy also facilitates SSL encryption, using Let's Encrypt.

9.3. Image Registry and Deployment workflow

A self-hosted Harbor instance serves as a private Docker registry. This instance is also exposed using the same reverse proxy under the domain **harbor.casperschouwenaar.nl** and is secured using *HTTPS* and login credentials.

The deployment process follows a manual routine:

1. Developers build a Docker image locally.
2. Developers tag the image.
3. Developers push the image to Harbor.
4. Images are manually pulled on the backend environment using the docker compose pull command.
5. The services are updated using the Docker commands; docker compose down and docker compose up -d.

No CI/CD tooling is currently implemented. Although GitHub is used for version control across repositories.

9.4. Monitoring and Maintenance

The current monitoring is being performed manually by checking the Docker logs. There is currently no centralized monitoring and alerting collection configured.