

16-6-2025

Version: 1.0

Test Strategy

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	14-04-2025	Initial structure	N/A
0.2	11-06-2025	Implemented main strategy	N/A
0.3	12-06-2025	Code quality and release	N/A
1.0	16-06-2025	First final version	Send for final assessment

Distribution

Name	Role	Date	Version
	Coach	18-06-2025	1.0

Contents

Introduction.....	3
1. Test approach	4
1.1. Test process.....	4
1.2. Types of tests	4
1.3. Test levels	5
1.4. Roles	5
2. Risk analysis	6
2.1. Likelihood vs. Impact matrix	6
2.2. Testing focus	6
2.3. Backend risk classification	7
2.4. Frontend (Unity) risk classification	8
3. Methods	9
4. Test environment	10
4.1. Environment.....	10
4.2. Test street	10
4.3. Database	10
5. Code quality.....	11
5.1. Conventions.....	11
5.2. Measurability	11
6. Release control	12
6.1. Branching strategy.....	12
6.2. Deployment and testing.....	12

Introduction

This document describes the test strategy for the AR Maintenance & Inspection Assistant, a proof-of-concept (PoC) application developed as part of the module *ICT.GP.PRJCT.V22_2425* at Windesheim HBO-ICT. The product combines a Unity-based AR interface with a NestJS backend connected to an MQTT broker and evaluates machine sensor data using fuzzy logic. The primary goal is to enable real-time, hands-free machine diagnostics and maintenance insights through augmented reality.

This test strategy outlines the testing approach that will be followed to ensure the product functions as intended during development. It provides a structured process for verifying both functional and system-level behavior of the product.

Given the nature of this project as a PoC, the emphasis lies on validating core features that demonstrate technical feasibility, particularly:

- The correct connection to the MQTT broker and reliable subscription to sensor topics
- The correctness of real-time data streaming via SSE
- The accuracy of the fuzzy evaluation logic
- The ability of the Unity AR frontend to correctly display dynamic evaluation results

This document is intended for use by the development team and the project coach to guide test planning, execution, and quality assurance throughout the project.

1. Test approach

The AR Assistant is developed as a proof-of-concept, which means the focus is on validating whether the key functionalities work as intended, rather than meeting production-grade standards. The application consists of a NestJS backend that communicates with an MQTT broker, and a Unity-based AR frontend that displays machine status information using tracked images.

This test strategy explicitly prioritizes the functional correctness of features over non-functional aspects such as performance, scalability, or security. The goal is to ensure that the main use cases, such as live data processing, fuzzy logic evaluation, and augmented reality display, work reliably under expected conditions.

The testing strategy is centered around two test types: unit tests for the backend and manual tests for both the backend and frontend. End-to-end (e2e) tests are also included in the backend, mainly to verify the behavior of REST endpoints in a more realistic, full-stack context. For clarity and simplicity, e2e tests are grouped under “unit tests” throughout the rest of this document.

1.1. Test process

Testing is done iteratively alongside development. Each core feature is tested immediately after implementation and retested after changes. Unit and e2e tests are maintained in the backend repository and are executed during development. Manual tests are performed on demand, especially during sprints, milestones, or integration phases.

1.2. Types of tests

To ensure that the most important features work as expected, we apply two types of testing that fit the scope and goals of the project: unit testing for backend logic and manual testing for both backend and frontend functionality.

Unit tests focus on the backend API. These are written in NestJS. Unit tests focus on specific services like sensor data parsing, fuzzy logic evaluation, and MQTT routing. End-to-end tests target API endpoints and simulate real-world use cases, such as machine lookups or MQTT status checks. These two types are grouped together under “unit tests” for the purposes of this report.

Manual tests cover both the backend and the Unity frontend:

- On the backend, Swagger UI is used to manually call endpoints and verify that responses are correct and meaningful.
- On the frontend, manual testing is done using physical AprilTags and real-world usage of the Unity AR app. Key scenarios include scanning a marker, rendering the correct machine panel, and testing the navigation UI.

1.3. Test levels

Testing is performed at two levels. At the unit level, individual backend functions (e.g., parsing MQTT data, evaluating fuzzy logic rules) are tested in isolation, e2e tests to test the endpoints. At the system level, manual testing verifies the full data pipeline from live MQTT input to Unity visualization.

1.4. Roles

All team members participate in testing. Backend developers are responsible for writing and maintaining unit tests, and for manually verifying API behavior through Swagger. The Unity team focuses on manual testing of the AR functionality, including marker recognition and UI interaction. Testing tasks are planned per sprint and distributed based on feature ownership.

2. Risk analysis

This risk analysis identifies functional risks for the AR Assistant PoC, split into backend and frontend components.

Since the AR Assistant is a proof-of-concept and not intended for production use, the primary risks lie in functional correctness and feature reliability. Non-functional risks such as performance and security are excluded from this risk analysis.

The focus is on identifying where the system may fail to deliver correct results, block core functionality, or disrupt the real-time data flow between MQTT, the backend API, and the Unity AR interface.

The risks listed here focus on core functionality, especially real-time data flow and visual feedback.

2.1. Likelihood vs. Impact matrix

The risk level of each feature is determined using two dimensions:

- **Likelihood:** The chance that a given issue might occur during the project.
- **Impact:** The severity of consequences if the issue occurs.

The table below visualizes how these two factors combine into a final risk level.

Impact \ Likelihood	Low	Medium	High
Low	Low	Low	Medium
Medium	Low	Medium	High
High	Medium	High	High

2.2. Testing focus

Risk classifications guide the allocation of testing effort:

- **High-risk backend features** such as MQTT communication and SSE streams are tested via unit tests and manual Swagger calls.
- **High-risk frontend features** like marker tracking and real-time updates are tested extensively in real-world AR setups.
- **Medium-risk items** like fuzzy scoring or graph rendering receive targeted manual testing.
- **Low-risk features** are still observed during tests but are not prioritized.

This strategy ensures that key functionality is protected and that the most likely failure points are addressed within the project's available time and scope.

2.3. Backend risk classification

Backend Functionality	Likelihood	Impact	Risk level	Justification
MQTT connection to broker	Medium	High	High	No data can be received or processed, breaking all downstream features.
MQTT topic subscription	Medium	High	High	If subscriptions fail, data from machines never reaches the backend.
Sensor data parsing	Medium	Medium	Medium	Incorrect structure may prevent evaluation or crash the backend.
Fuzzy logic rule loading	Low	Medium	Low	Rules are read from JSON. If misconfigured, only that machine's evaluation breaks.
Fuzzy membership function errors	Low	Medium	Low	If labels or values are misaligned, evaluations may be incorrect or misleading.
Fuzzy evaluation scoring	Low	Medium	Low	Bad scoring logic could lead to inaccurate priority levels, lowering decision quality.
SSE stream to Unity	Medium	High	High	If SSE fails, Unity receives no live updates, removing key value from the system.
Swagger API not responding	Low	Medium	Low	Makes manual testing harder but doesn't affect the Unity flow.
Data not reaching Unity via SSE	Medium	High	High	Unity panels depend on a live connection; no updates mean stale data or empty views.

2.4. Frontend (Unity) risk classification

Unity Functionality	Likelihood	Impact	Risk level	Justification
Image tracking (AprilTags)	High	Medium	High	Tag detection may fail in poor lighting or angle, preventing panel rendering.
Initial panel positioning	Medium	Medium	Medium	If panels spawn incorrectly, they may appear offscreen or at wrong depth.
Data panel not updating	Medium	High	High	Real-time data is core to the user experience; broken updates make the app misleading.
Legend not showing correct sensors	Medium	Medium	Medium	This affects clarity but not data processing or detection.
Machine info (name/description) bug	Low	Low	Low	Minor UX issue if labels or info are missing or incorrect.
Graph panel rendering (line graph)	Medium	Medium	Medium	If graph panels are blank or misaligned, user feedback is unclear.
Navigation and haptic feedback	Low	Low	Low	Enhances UX, but it is not essential to functionality demonstration.

3. Methods

This chapter describes the test methods that will be applied to verify the core functionalities of the AR Assistant. Only unit tests and manual tests are used, as they are sufficient to validate whether the main features behave as expected.

The table below outlines which test method is used per feature:

Feature	Unit Test	E2E Test	Manual Test	Description
MQTT connection to broker		X		Tested manually by observing connection logs and using Swagger to check status.
MQTT topic subscription		X		Verified through subscription status endpoints and live data flow.
Sensor data parsing	X	X		Unit-tested with valid and invalid payloads; manually tested via MQTT injection.
Fuzzy logic rule evaluation	X			Unit tests check correct scoring per rule set; results also validated via Unity display.
SSE streaming to Unity			X	Verified in Unity by observing if real-time updates arrive and update the UI.
Swagger endpoints (GET, SSE, etc.)		X	X	Checked with unit tests and manually via Swagger UI for expected output.
Unity marker tracking (AprilTags)			X	Tested with real printed markers in different lighting and angles.
Unity data panel visibility and updates			X	Verified by scanning a tag and watching the data panel update in real time.
Unity sensor graph rendering			X	Tested by checking if graphs update correctly with live sensor values.
Unity machine info display (ID, name, desc.)			X	Verified manually by scanning each tag and checking displayed info.
Unity navigation and feedback			X	Explored manually to ensure usability and correct responses to triggers.

Unit tests are primarily used for backend logic, including sensor data parsing and fuzzy evaluation. Manual tests are used for both backend verification via Swagger and Unity frontend validation using real AR scenarios. E2E tests are mainly used to test important endpoints.

4. Test environment

This chapter describes the environments and infrastructure used to execute tests on both the backend and the frontend. The AR Assistant is tested in a local development setup that simulates real-world usage as closely as possible, including MQTT data flow, Unity builds, and manual Swagger endpoint calls.

4.1. Environment

The project is developed and tested locally. The backend runs on a local NestJS development server, with `.env.development` providing configuration for MQTT broker connection and port bindings. The MQTT broker itself is run locally or in a Docker container depending on the test setup.

The Unity AR application is deployed on Android test devices using AR Foundation and is connected over a shared Wi-Fi network to the backend. Testing is done using printed AprilTags in a physical room to simulate realistic usage.

No separate staging or production environment exists, as the project is a single-environment proof-of-concept.

4.2. Test street

There is no automated CI pipeline or deployment setup. Tests are run locally by developers:

- **Backend unit and e2e tests** are executed via the NestJS test runner (jest) during development.
- **Manual tests** are performed using Swagger UI and Android builds of the Unity app.
- New features are tested individually on each developer machine before being merged into the main branch.

Testing responsibilities are shared across the team and are planned per sprint. The test street consists of:

- Manual validation of endpoints via Swagger
- Running unit test suites using npm test
- Deploying new Unity builds to an Android phone and testing behavior using real AprilTags

4.3. Database

The backend does not use a persistent database. All live data is processed in-memory and transmitted through MQTT or SSE. As a result, there is no need to manage a separate test database. Instead, simulated MQTT payloads are used to test real-time data flows.

For manual testing, tools like MQTT Explorer or custom scripts are used to publish test messages to MQTT topics, simulating machine sensor output.

5. Code quality

Code quality is essential to ensure that the project remains maintainable and understandable, both during development and for any future improvements. Although the AR Assistant is a proof-of-concept, attention is still paid to keeping the codebase clean, readable, and consistent.

5.1. Conventions

For the backend (NestJS), we follow consistent naming conventions and TypeScript style best practices. All services, controllers, and pipes follow the official NestJS module structure and are organized by feature.

In the frontend (Unity), C# code follows PascalCase for class names and camelCase for methods and variables. Classes are organized by responsibility, and AR-related logic is separated from UI logic to keep the code modular.

Git commits follow a clear convention by referencing the associated GitHub issue number and describing the implemented change. Feature branches are named using the format feature/short-description.

5.2. Measurability

Backend code quality is supported through unit testing and type safety provided by TypeScript. ESLint is used to catch basic formatting and logic issues during development.

Although no formal code coverage targets are set, unit tests focus on critical logic paths such as:

- Parsing and validating MQTT payloads
- Fuzzy logic evaluation and rule application
- SSE stream construction and message formatting

Code is reviewed by peers via pull requests before being merged into the dev branch. PR reviews include checks for clarity, responsibility separation, and consistent implementation of logic.

The Unity code is not formally linted or tested but is reviewed during team collaboration and verified through manual testing.

6. Release control

To ensure stability and prevent regressions, the team uses a structured branching and release workflow. Even though the AR Assistant is a proof-of-concept, this controlled approach helps keep the codebase clean and the application testable at each stage.

6.1. Branching strategy

The team uses a simplified Git workflow based on feature branches:

1. All work begins with a GitHub issue.
2. A new feature branch is created from the dev branch, named feature/....
3. When the feature is complete and tested locally, it is merged into the dev branch through a pull request.
4. After successful manual and unit testing, changes from dev are merged into the testing branch for integrated frontend/backend testing.
5. Finally, after final validation, the code is merged into the main branch.

This flow ensures that:

- Experimental work is isolated.
- Only tested and reviewed features reach the main branch.
- Developers always work against an up-to-date, stable base.

6.2. Deployment and testing

The live backend API is deployed to: <https://api.arassistant.nl>

This environment is used for:

- Testing the Unity frontend against a realistic server.
- Verifying Swagger endpoint responses outside of local development.
- Checking SSE streams with live MQTT inputs.

There is no automatic CI/CD pipeline in place. Deployment is performed manually when the main branch is updated. The deployment is considered part of the final testing phase.

All manual tests are repeated against this live environment before the project is finalized.