Test Planning LO2

s2298559

December 2025

1 Context and Constraints

The PizzaDronz system was implemented during a previous academic year, before formal engagement with the Software Testing course. As a result, testing was necessarily retrospective, and it was not possible to apply Test-Driven Development (TDD) during initial implementation.

The testing strategy reframes the work as a legacy-system testing exercise. The test plan therefore aims to:

- characterise existing system behaviour
- expose hidden assumptions and weaknesses
- evaluate adequacy and limitations of testing techniques
- assess how a TDD-style approach would have improved outcomes had it been applied earlier

2 Construction of the Test Plan

2.1 Planning Philosophy

The test plan follows a risk-driven, incremental validation strategy. It is inspired by TDD principles but adapted for post-hoc testing:

1. stabilise behaviour through controlled tests,
2. validate requirements,
3. expose inconsistencies or implicit assumptions,
4. and expand coverage where weaknesses are identified.

This reflects widely accepted practice when testing previously implemented, safety-related software.

2.2 Requirement Prioritisation

Requirements were prioritised according to their risk level and impact of failure.

High-priority (safety- and correctness-critical)

- FR1: Order validation correctness,
- FR3: No-fly zone avoidance,
- FR4: Central Area constraint.

Failure in these areas could lead to unsafe drone behaviour or invalid deliveries. The test plan therefore demands:

- early and repeated verification,
- redundant testing using multiple techniques,
- pessimistic failure handling (any violation fails the test).

Medium-priority

- FR5: REST endpoint behaviour,
- FR2 and QR2: Numerical tolerances and movement constraints.

Lower-priority

- QR1: Performance guideline (approx. 60 seconds).

Since performance is defined as a guideline rather than a strict real-time requirement, performance testing is intentionally placed later in the plan.

2.3 Planned Evolution of the Test Suite

Although the system was complete prior to testing, the test plan was designed to evolve iteratively:

- initial correctness tests revealed hidden assumptions
- which motivated deeper invariant tests
- which in turn motivated robustness and negative tests
- eventually leading to performance scaling and statistical tests

This resembles the incremental expansion characteristic of TDD, even though tests did not drive implementation.

3 Instrumentation and Scaffolding

3.1 Scaffolding Strategy

To test a system heavily dependent on geometric logic and external REST data, extensive scaffolding was required:

External dependency control

- Stubbed REST services for restaurants, no-fly zones, and the Central Area.
- Deterministic integration tests independent of live data availability.

Synthetic data generation

- Valid and invalid synthetic orders,
- Geometric configurations including convex, nested, and adversarial no-fly zones.

Execution harnesses

- Repeated execution loops,
- Sweeping parameter ranges (e.g. number of no-fly zones),
- Systematic variation of start locations and drone angles.

This scaffolding supports correctness, robustness, and statistical analysis.

3.2 Instrumentation Techniques

Instrumentation was kept minimally invasive to avoid modifying legacy behaviour:

- Timing instrumentation around path-planning,
- CSV logging for performance and statistical analysis,
- Minimal diagnostic logs in routing logic for debugging failures,
- REST-layer fault injection (e.g., malformed JSON, missing fields).

This provided observability without altering system semantics, appropriate for validating an established codebase.

4 Evaluation of the Test Plan

4.1 Strengths

The test plan exhibits several strengths:

- Clear risk-driven prioritisation,
- Layered structure (unit → integration → system),
- Effective scaffolding enabling deterministic testing,
- Support for statistical and property-based testing,
- Adaptability in response to observed failures.

4.2 Limitations

The plan also has inherent limitations:

- Being retrospective, it cannot influence architecture or prevent early defects
- Some behaviours were only discovered after observing failures
- No strict numerical coverage targets (e.g., mutation score) are defined
- Instrumentation focuses on external behaviour rather than internal routing decisions.

These limitations are documented explicitly to avoid overstating test confidence.

5 Evaluation of Instrumentation

The instrumentation successfully supports:

- invariant checking,
- robustness evaluation,
- performance trend analysis,
- and reproducibility of experiments.

However, it has clear limitations:

- Internal routing heuristics remain largely opaque
- No fine-grained internal event logging
- Diagnosing geometric failures sometimes requires manual inspection

While the implemented instrumentation enabled correctness checking, robustness testing, and basic performance analysis, several important limitations remain. First, the routing algorithm's internal decision-making process is largely opaque: only the final path and selected directions are observable. This means that incorrect intermediate decisions (e.g., suboptimal detours, avoidance failures that self-correct later) cannot be detected unless they manifest as a final invariant violation. More fine-grained instrumentation, such as structured logging of per-segment decisions or recording the rationale for direction choices, would allow these subtle behavioural issues to be identified earlier.

Second, the performance instrumentation records total runtime but not per-component timing. As a result, it is not possible to isolate the relative contributions of geometry checks, REST data handling, or path extension logic to overall runtime variation. Component-level timers would help diagnose bottlenecks and support more precise regression testing.

Third, robustness instrumentation is limited to REST-layer fault injection and selected geometry failures. Mutation testing or systematic fault seeding within routing logic, would provide a more objective measure of the sensitivity and adequacy of the test suite.

Finally, the constraints of a legacy, pre-existing codebase restricted the amount of instrumentation that could be introduced without risking behaviour changes. This trade-off is appropriate for validating a stable system, but it limits the completeness of the evaluation. A TDD or early-stage development context would have permitted far richer instrumentation without such risk.