Test Planning Document

> ⓘ **I've taken the liberty of combining multiple functional requirements chosen in my 1.3 document into 4 core testable requirements for planning and testing**

Requirement 1

No response for an endpoint should take more than 30 seconds

This is a lower level requirement that measurable, as such, we will need a way to measure it:

- As mentioned in the 1.2 document, this is a system level test. As such, the whole system must be in place before this can be tested.
- As this is a complicated pathfinding problem, at some level of complexity the computation times will get longer than 30 seconds - this is not a fault with the software unless this happens for a small number of delivery locations. This would indicate a reliability issue. To deal with these longer times, we will return 400 rather than letting the computation run indefinitely. This adheres to the principle of ***Sensitivity***, outlined in chapter 3 of Y&P, stating *"it is better to fail every time than sometimes"*. We would rather consistently fail for 30 second long computation than occasionally take a much longer time to respond - and likely timeout out of our control.
- To do this, we must:
    1. Generate test data - as outlined in 1.3, pathfinding queries will take the longest. As such, we will need a way to reliably generate random and strenuous pathfinding problems, and then ways to test against this.
    2. Establish a method of measuring computation time.

Requirements 2, 3, and 4

A drone flight should deliver all medDispatches assigned to it

Ensure flight paths begin and end at the same service point

Ensure flight paths are otherwise correct

These are functional correctness requirements that span route planning, movement constraints, and delivery semantics. This is high priority because failure to deliver assigned MedDispatches correctly represents a core functional failure of the system.

Delivery feasibility depends on constraints such as no-fly zones, movement rules, and cost limits. As a result, the requirement cannot be validated solely at unit level and requires integration of multiple components.

Crucially, this is not testing *whether a drone should be able to deliver something*, but rather *is a returned flight path correct*.

Test approach and level

These requirements will be verified primarily through integration-level testing, exercising the route-planning logic via REST endpoints while using real implementations of movement, geometry, and cost calculations.

Unit testing will also be employed in adherence to the Partition principle.

A handful of black-box system-level tests will also be used to ensure functionality is preserved when the service is deployed using docker.

Integration-level approach

The general testing approach consists of:

- Generating a list of MedDispatch
- Requesting a flight path via the REST endpoint
- Analysing the returned flight path to determine whether:
  - Each assigned MedDispatch delivery location is visited
  - Paths start and end at the same service point
  - Ensure flight paths are otherwise correct

Where feasible, negative cases should also be included, in which delivery is known to be impossible - a trivial example being a single MedDispatch with a delivery location very far away.

Unit-level approach

Following the Partition principle from chapter 3 of Y&P, the last point ensuring flight paths are otherwise correct has been broken down into 3 unit-level test cases to be covered:

1. Drones can only move with an angle that is a multiple of 22.5˚
2. Drones can only move by *exactly* 0.00015˚ in a given direction
3. Drones should not be able to fly over no-fly zones, including corner cutting.

The testing approach for these is as follows:

- Generate a handful of rectangular no-fly zones (*this accurately reflects the amounts we were to consider in the ILP coursework*)
- Generate two random points in Edinburgh city limits that are not inside any of the no-fly zones
- Attempt to generate a path between those two points
- Analyse returned path (if any) to ensure it respects the above requirements

Lifecycle

Within an XP/TDD-inspired lifecycle, this requirement is addressed as follows:

- Unit tests validate movement and geometry constraints
- Integration tests verify that these components combine to achieve complete delivery
- System tests confirm that the deployed service preserves this behaviour

This placement supports early fault detection while maintaining confidence in end-to-end correctness.

Instrumentation and Scaffolding

Requirement 1 - *No response for an endpoint should take more than 30 seconds*

To test this requirement, I needed two things:

1. A way to generate random dispatches around Edinburgh
2. A way to measure the execution time of the `/api/v1/calcDeliveryPath`

To generate random medDispatches, I created three scaffolding functions: one to generate a random point within Edinburgh, one to generate n random points in Edinburgh, and one to generate n randomly located medDispatchRecs. As we're not testing whether the correct drone is selected, other than the delivery location I selected the least-restricting requirements for all dispatches so that they would all be deliverable.

To measure the response time, I needed some instrumentation. I measure the system time in milliseconds before the request is sent, and after the response is received. I then use the difference to assert whether the request was made in time or not.

Requirement 2 - *A drone flight should deliver all medDispatches assigned to it (if possible)*

To test this requirement, I needed two things:

1. A way to generate random dispatches so that a path can be generated and checked against
2. A way to determine whether all medDispatches have been delivered or not.

For the first, I was able to reuse my scaffolding functions from before.

For the second, I wrote some instrumentation to determine whether all delivery locations had been visited. A medDispatchRec is marked as "delivered" if the drone hovers at the delivery location, and a hover is denoted by a path containing the same point twice in a row. Using

these criteria, I loop through all delivery paths to ensure this is the case for all medDispatchRecs.

Requirement 3 - *Ensure flight paths begin and end at the same service point*

To test this requirement, I needed two things:

1. A way to generate random dispatches so that a path can be generated and checked against
2. A way to determine whether the path begins and ends at the same point

For the first, I was also able to reuse my scaffolding functions from before.

For the second, I wrote more instrumentation to ensure the first position of the first delivery path and the last position of the last delivery path was the same.

Requirement 4 - *Ensure flight paths are otherwise correct*

To test this requirement, I split it into three unit-level requirements.:

1. Drones can only move with an angle that is a multiple of 22.5˚
2. Drones can only move by *exactly* 0.00015˚ in a given direction
3. Drones should not be able to fly over no-fly zones, including corner cutting.

For all of these, I re-used the random point generation to create the paths, and then wrote some instrumentation to analyse the properties of the path and ensure they matched with the requirements.

Evaluation of Instrumentation

I think all instrumentation was exactly as necessary to ensure the robustness of the requirements in the given test area - that being within the envelope of the ILP coursework.

A place to improve in the future might be generating random no-fly zones, rather that using the provided ones. This would require lots more work to validate, but would help increase the level at which we can trust the tests prove the functional correctness of the implementation.

This would help in the hypothetical scenario that the medical drone delivery service were to expand beyond Edinburgh.

Limitations of the test plan

As the plan relies on re-using some functionality to generate random pathfinding or deliver scenarios, any bug in this generation propagates into the rest of these tests, creating a potential weak point in the testing system.

By keeping this generation functionality easy to understand (smaller functions, commenting), this should help mitigate any risk of error, given that this generation is relatively simple.

This plan also has intentionally not made a decision on how tests against a docker image will be performed - this is because the author doesn't quite know how they're going to do that at this time. Their best guess would be a separate controller that runs at the same time and performs tests against those exposed endpoints.

This document also keeps explicit implementation detail very surface level, this is to provide as much room to iterate on how someone would like to test a requirement as possible whilst also keeping myself aligned with a more agile stile of development.

Along with these, this test plan intentionally does not mention the following:

- Concurrency or parallel requests
- Long-running degradation (memory leaks, cumulative state)

Whilst these are certainly something worth testing if we were doing full scale testing of the project, they don't align with the requirements I've chosen for this portfolio, and so are intentionally neglected.

Overall, this plan is adequate for coursework scope but not production readiness. Everything explicitly chosen is considered, whilst unnecessary or potentially distracting aspects have been removed.

These limitations are considered acceptable given the project scope, as the primary risks addressed are incorrect flight-path generation and unacceptable response times, both of which directly impact core system functionality.