Requirements Analysis to Determine Testing Strategies

Requirements

1. Functional Requirements
   - 1.1. Safety
     - 1.1.1. The system shall not generate a drone route that passes through a no-fly zone.
     - 1.1.2. The system shall not assign a delivery to a drone if the drone's remaining moves are insufficient to complete the delivery and return to its base.
   - 1.2. Correctness
     - 1.2.1. The system shall assign deliveries only to drones that satisfy cooling/heating, capacity, and availability constraints.
     - 1.2.2. The system shall generate flight paths using only the defined 16 compass directions with a step size of 0.00015 degrees.
     - 1.2.3. The system shall accept well-formed JSON requests and reject malformed JSON requests.
   - 1.3. Liveness
     - 1.3.1. After completing assigned deliveries, a drone shall return to its base location.
     - 1.3.2. For any invalid request, the system shall return a valid JSON error response without terminating the service.
   - 1.4. Security
     - 1.4.1. The system shall return a generic HTTP 404 response for invalid requests, without exposing detailed error information.

2. Measurable Attributes
   - 2.1. Performance
     - 2.1.1. If the REST service is under normal load, then the mean time between sending a REST API request to the system and the system responding shall be less than 3s
   - 2.2. Resource Usage
     - 2.2.1. The system shall operate within the memory limits of its Docker container.
   - 2.3. Reliability and Availability
     - 2.3.1. Given the same valid delivery request, the system shall generate drone routes using the same route-planning logic applied to the latest data retrieved from the REST data service.
     - 2.3.2. The health endpoint shall report status "UP" when the service is operational.
   - 2.4. Resilience
     - 2.4.1. The system shall continue to determine delivery eligibility when optional drone attributes are missing by treating them as absent.

3. Qualitative Requirements
   - 3.1. Usability
     - 3.1.1. The system should be easy to use for external clients.
   - 3.2. Maintainability
     - 3.2.1. The system should be easy to modify and extend.

4. Other Aspects
    4.1. Operational Constraints
        4.1.1.The system shall retrieve external REST data at runtime.
        4.1.2.The system shall run inside a Docker container
    4.2. Regulatory/Compliance
        4.2.1.The system must comply with coursework specification.
    4.3. Assumptions and Limitations
        4.3.1.No-fly zones are rectangular.
        4.3.2.Drone speed and power consumption are constant.
        4.3.3.The system does not handle real-time drone failures.

Sample Analysis of Requirements

The levels of a few requirements, along with a testing approach and justification, have been provided below. However, moving forward, only requirements *1.1.1* and *2.1.1* will be analysed and prepared in depth.

R1.2.3

Level: *Unit*
Test Approach: *Unit testing* will be applied to request parsing and validation logic using malformed JSON inputs, such as syntax errors, incorrect data types, missing required fields, and unexpected fields.
Appropriateness: Request parsing concerns isolated, deterministic logic so unit testing is appropriate. However, as of now parsing tests will not include SQL injection resistance, as the implementation does not execute SQL queries. Moreover, the use of a generic HTTP 404 response prevents distinguishing between different types of invalid requests but is a deliberate trade-off to reduce information disclosure.

R1.1.2

Level: *Integration*
Test Approach: *Integration testing* will be applied to *1.1.2*, as delivery assignment depends on route calculation and drone state to determine feasibility. Test scenarios will ideally include drones with just sufficient remaining moves and drones that become infeasible after an additional delivery assignment.
Appropriateness: This depends on interaction between multiple components i.e. delivery assignment with route calculation. However, the delivery assignment logic is not fully optimised, so some feasible assignments may not be reachable, for which the selected cases may not expose a shortage of remaining moves. Nevertheless, the tests will align with the assignment logic implemented in the service.

R1.2.1

Level: *Integration*
Test Approach: For *1.2.1*, *integration testing* will cover delivery assignments using combined drone and order data, including cases where all constraints are satisfied and where one or more constraints are violated.

Appropriateness: Once again, this requirement is dependent on the interaction between several components namely delivery assignment and input parsing for which integration testing suffices.

R2.4.1

Level: *Integration*
Test Approach: Likewise, for *2.4.1*, *integration testing* will use drone data with omitted optional attributes such as cooling or heating to verify interaction between input parsing and delivery assignment logic, ensuring the service continues operating and treats missing attributes as not present.
Appropriateness: The justification for using integration testing here is similar to that of *R1.2.1*.

R1.1.1

Level: *System*
Test Approach: *Unit testing* will support verification of the system-level requirement by testing the function that validates a candidate neighbour during route expansion in the A algorithm*, covering cases where the segment between the current point and the neighbour crosses a region edge, touches a region boundary, or touches a region corner.
Appropriateness: Drone step validation is an isolated, deterministic logic and hence, unit testing seems to be the appropriate choice. Moreover, validating individual movement steps is fast and reduces the likelihood of system-level violations by preventing invalid segments from being incorporated into generated routes. It may appear that the validation of the drone step logic cannot speak for the entirety of a route. However, as of now, it suffices to know that a route is made up of drone steps and each step in the route will only be taken if valid. The test planning document will explore such limitations in more depth.

R2.1.1

Level: *System*
Test Approach: *System-level testing* will be applied to *2.1.1* by issuing repeated REST API requests to the deployed service and measuring response times.
Appropriateness: Response-time testing verifies overall system functionality, including network connectivity and backend processing. So, since this requirement concerns end-to-end system behaviour, the decided testing approach seems appropriate.

R2.3.1

Level: *System*
Test Approach: *System-level testing* will use controlled external data by mocking the REST data service providing consistent information about drones; identical delivery requests are issued repeatedly to verify consistent routing behaviour, and the dataset is then changed to confirm that any route differences are attributable to data changes rather than nondeterministic behaviour.
Appropriateness: Through system testing, we will be able to confirm that routing behaviour remains consistent with the specification and so, reliable during complete execution.