Test Planning

This test planning document is developed for a REST-based service used to calculate drone routes for medicine deliveries. The system is being constructed using an iterative development lifecycle, with verification and validation activities planned and introduced progressively as the system matures. The plan focuses on two system-level requirements identified at the start of the project, as they form the foundation for the correct and safe operation of the service, and illustrates how and when testing of such requirements is carried out within this lifecycle. The requirements are:

- *R1.1.1*: The system shall not generate a drone route that passes through a no-fly zone.
- *R2.1.1*: 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 3ms.

Priority and Pre-requisites

R1.1.1: It is a safety requirement and is therefore likely to be regulated and of high priority. This suggests that a reasonably high level of resource should be devoted to ensuring the requirement is met. Due to its safety-critical nature, the Visibility and Partition principles underlying analysis and testing techniques are applied. The partition principle suggests that, to ensure the drone never enters a no-fly zone, the requirement should be decomposed into checking that each segment making up a route is valid and further decomposed into the specific ways in which a segment can violate the requirement. A segment refers to one movement step of the drone. It is a straight line between the point where the drone currently is and the next point to which it wishes to move. The requirement may be violated if:

- the next point lies inside a no-fly zone,
- the next point lies on the boundary or edge of a no-fly zone, or
- the segment intersects any edge of the no-fly zone.

Since a route is constructed as a sequence of such segments, verifying that each segment satisfies these constraints provides strong confidence that the constructed route will satisfy the system-level requirement. This is also true because the starting point of any route will always be a valid medical dispatch point, and the consequent points are then verified using the logic above. The inputs to this check are four pairs of coordinates (vertices) defining a rectangular no-fly zone (*v1, v2, v3, v4*), the current drone position *current_point*, and the candidate next position *next_point*. The output is a Boolean value *safe_to_move*, which is true if the segment satisfies the requirement and false otherwise.

With this divide and conquer approach established, two tasks are scheduled in the plan:

- Some sort of inspection to ensure that only code enforcing these checks is subject to inspection and that no extraneous code is present beyond what is required to verify the requirement. (Visibility principle)
- An exhaustive test that checks every possible way of violating the segment-level sub-requirement for *R1.1.1* and confirms that the observed results conform to the specification.

R2.1.1: As response time depends on the complete execution of the system, this performance requirement can only be meaningfully tested at the system level. While it is not safety-critical, it directly affects the responsiveness of the service and is therefore assigned a relatively lower priority than safety requirements, with testing effort scaled accordingly. The following tasks are scheduled to support testing of this:

- Generating synthetic REST API request workloads representative of normal load.
- Establishing a controlled execution environment to minimise external sources of latency.
- Designing the logging system to capture request and response timing data.
- Designing and implementing the analytics to compute mean response time and compare it against the specified threshold.

Scaffolding and Instrumentation

R1.1.1: While the inspection will not require any scaffolding or instrumentation, a later exhaustive test that checks all combinations of the three identified ways in which a segment can violate the requirement will require generic test scaffolding. A JUnit-based test framework will be used to provide test execution, assertions, and comparison of observed results with expected outcomes. As the resulting test suite is small and consists of a limited number of hand-written test cases, generic scaffolding is sufficient and needs to be scheduled accordingly.

R2.1.1: This requires more scaffolding and instrumentation as we need to

- Simulate concurrent users sending REST API requests to the running server using a load-generation tool (e.g. Apache JMeter).
- Define normal load conditions including the number of concurrent users, request frequency, and representative REST endpoints to be exercised. This will involve some effort that needs to be scheduled.
- Establish a system-level test environment, in which the REST service is executed locally (within a Docker container) and exercised under synthetic load.
- Schedule the design of instrumentation to log response times within the JMeter test plan to enable analysis of system performance under load. For example, we need to add appropriate listeners (e.g. Aggregate Report) within the JMeter test plan to record response times, throughput, and error rates for each request
- Analyse the collected results to determine whether the mean response time remains below the required threshold under normal load.

Process and Risk

R1.1.1: The inspection and exhaustive test approach outlined for this requirement in the first subsection of this document for verification and validation can be attempted early, i.e. in the first and/or subsequent iterations. By inspection, we will know whether all the logic needed to check for the three constraints is in place, after which testing can be conducted. This introduces a risk that, although early inspection and validation are performed for each segment, some violations that appear only in a full route case may remain unidentified until the logic for creating an entire route between two given points is implemented. However, given the constrained input space and the safety-focused inspection performed earlier, the risk associated with full route testing is considered low and manageable because given the iterative lifecycle, unit testing of the routing logic will be done before proceeding to the next core logic like assigning deliveries.

R2.1.1: Performance testing can only be done when a complete, functioning system is in place. However, the representative HTTP requests within the JMeter test plan can be created early and concurrently, as the requirements and analysis form the very first stage of iteration. So, the types of requests being made are already known. The remaining elements of the test plan can then be completed just before deployment in the final iterations. The primary risk is that the simulated load may not be fully representative of a typical high-demand situations, such as sudden surges in delivery requests. Achieving a highly realistic workload model and operational environment may require additional resources, such as access to production-like infrastructure, larger-scale load generation capability, and extended soak testing. Due to the time and resource constraints of the coursework, these enhancements may not be feasible and therefore pose notable risks to the performance testing process.