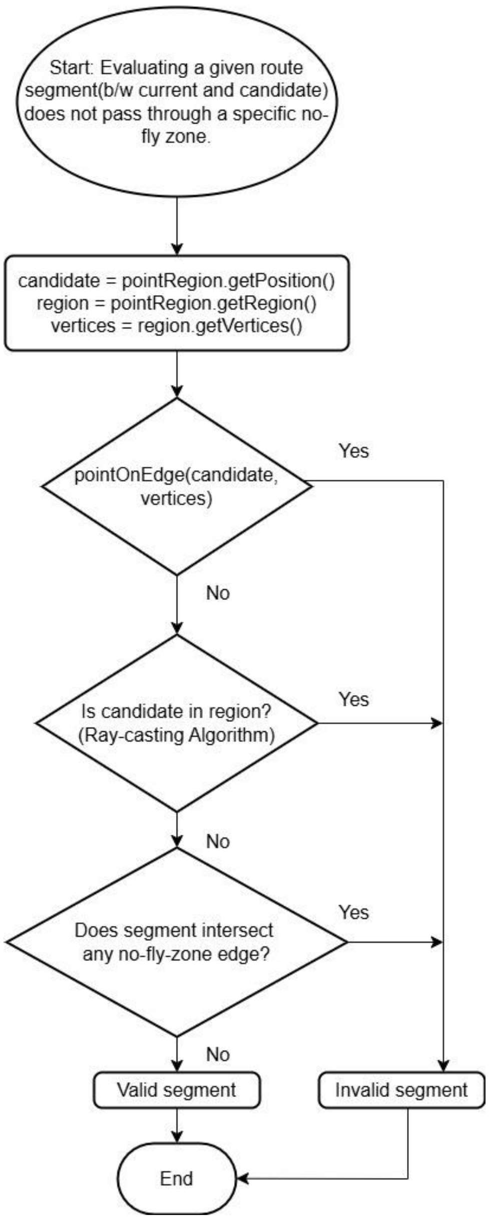
Testing Strategies and their Adequacy

This document provides evidence of a variety of testing techniques and the adequacy criteria considered for several requirements, followed by an evaluation of the test results. While the focus will be on the work completed for *R1.1.1* and *R2.1.1* (the requirements we shortlisted earlier) to demonstrate the full software testing process within the given time constraints, some attempt has also been made to meet this learning outcome for two additional requirements. However, the portfolio will refer only to the work completed for *R1.1.1* and *R2.1.1* for this Learning Outcome.

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

Testing technique (flowgraph-based testing): Flowgraph-based testing is appropriate because the logic enforcing the no-fly zone constraint is implemented as a deterministic sequence of decisions. The flowgraph makes explicit the three distinct ways in which a route segment may violate the requirement: the candidate point lying on the boundary, lying inside the region, or on the segment intersecting a region edge. By modelling this decision structure, test cases can be derived systematically rather than selected arbitrarily.



Start: Evaluating a given route segment(b/w current and candidate) does not pass through a specific no-fly zone.
candidate = pointRegion.getPosition()
region = pointRegion.getRegion()
vertices = region.getVertices()
pointOnEdge(candidate, vertices)
Yes
No
Is candidate in region? (Ray-casting Algorithm)
Yes
No
Does segment intersect any no-fly-zone edge?
Yes
No
Valid segment
Invalid segment
End

TC	On Edge	Inside	Intersects Corner	Expected Outcome	
TC-1	Yes	-	-	Invalid	✓
TC-2	No	Yes	-	Invalid	✓
TC-3	No	No	Yes	Invalid	✓
TC-4	No	No	No	Valid	✓

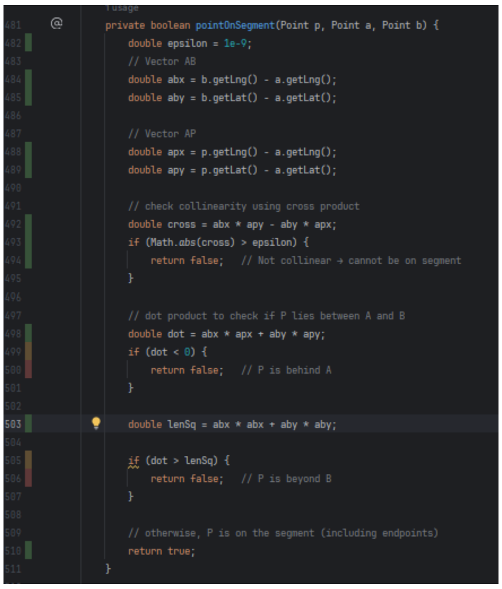
Adequacy Criteria (Path coverage + statement coverage): Path coverage was used as the primary adequacy criterion because the specification indicates three distinct ways the no-fly zone constraint can be violated at segment level, leading to a small, deterministic set of feasible execution paths in the top-level validation logic.

To strengthen this evaluation, IntelliJ statement coverage was also used to confirm that the implementation corresponding to these paths was actually executed during testing. While the top-level method achieved full statement coverage, the coverage report showed that one lower-level helper method (*pointOnSegment*) contained some unexecuted statements. This motivated three additional test cases to exercise those remaining outcomes. After adding these tests, the coverage report indicates full statement coverage across the methods involved in validating this requirement, increasing confidence that the constraint is thoroughly tested.

This leads us to formally state that the adequacy criterion for this requirement is satisfied by the test suite if each path in the above flowchart is taken by at least one test case, and the outcome of each identified test case is "pass".

Results:

- The tests for this requirement can be found in *src\test\java\ilp_submission_2\service\R1Test.java*.
- Tests passed on the first iteration.
- Using IntelliJ's feature "Run R1 Test with Coverage" indicates
  - 100% statement coverage in the top-level method *positionInRegionCheckForAStar* which is what is presented by the flowchart which reaffirms the claim that the four test cases exercise each feasible decision path in the flowchart.
  - However, this is not enough for complete assurance as some lower-level methods are also being accessed.
  - This top-level method accesses three lower-level methods – *pointOnEdge*, *pointOnSegment*, and *segmentsIntersect*.
  - *pointOnEdge* and *segmentsIntersect* also have 100% statement coverage.
  - However, the coverage report for *pointOnSegment* motivates three more test cases to achieve complete and thorough coverage of the ways in which segment validation truly works.
  - These three test cases are where candidate lies on the segment and the other two are where candidate lies on the line passing through the segment but not on the segment (so is before the first point or after the second point).
  - The new coverage report indicates 100% statement coverage throughout all the methods used to verify this requirement.



```
1 usage
private boolean pointOnSegment(Point p, Point a, Point b) {
    double epsilon = 1e-9;
    // Vector AB
    double abx = b.getLng() - a.getLng();
    double aby = b.getLat() - a.getLat();

    // Vector AP
    double apx = p.getLng() - a.getLng();
    double apy = p.getLat() - a.getLat();

    // check collinearity using cross product
    double cross = abx * apy - aby * apx;
    if (Math.abs(cross) > epsilon) {
        return false;   // Not collinear → cannot be on segment
    }

    // dot product to check if P lies between A and B
    double dot = abx * apx + aby * apy;
    if (dot < 0) {
        return false;   // P is behind A
    }

    double lenSq = abx * abx + aby * aby;

    if (dot > lenSq) {
        return false;   // P is beyond B
    }

    // otherwise, P is on the segment (including endpoints)
    return true;
}
```

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 3s.

Testing technique (Black-box, systematic functional testing): Functional testing where the test space is defined by normal load conditions (e.g. number of concurrent users, ramp-up time, and loop count). Representative workload configurations are selected systematically rather than arbitrarily. As the requirement concerns a measurable, externally observable attribute, a black-box, system-level testing approach is appropriate. This enables concurrency-related faults and performance issues that only arise under load to be identified.

TC	Level/Purpose	Threads (users)	Ramp-up Time(s)	Loop Count	Pass Criteria	
TC-1	Warm-up	1	1	10	System runs successfully	✓
TC-2	Baseline (single user)	1	1	20	All requests succeed	✓
TC-3	Expected low-normal load	5	5	20	All requests succeed and response times are acceptable	✓
TC-4	Expected normal load	10	5	20	All requests succeed and response times are acceptable	✓
TC-5	Expected high-normal load	20	5	30	All requests succeed and response times are acceptable	✓
TC-6	Concurrency stability check	20	3	20	All requests succeed with no concurrency errors	✓
TC-7	Soak under normal load	10	5	75	System remains stable for the full duration	✓
TC-8	Stress	50	10	50	Analyse application's behaviour and ensure that the software displays the appropriate error messages under extreme conditions.	✓

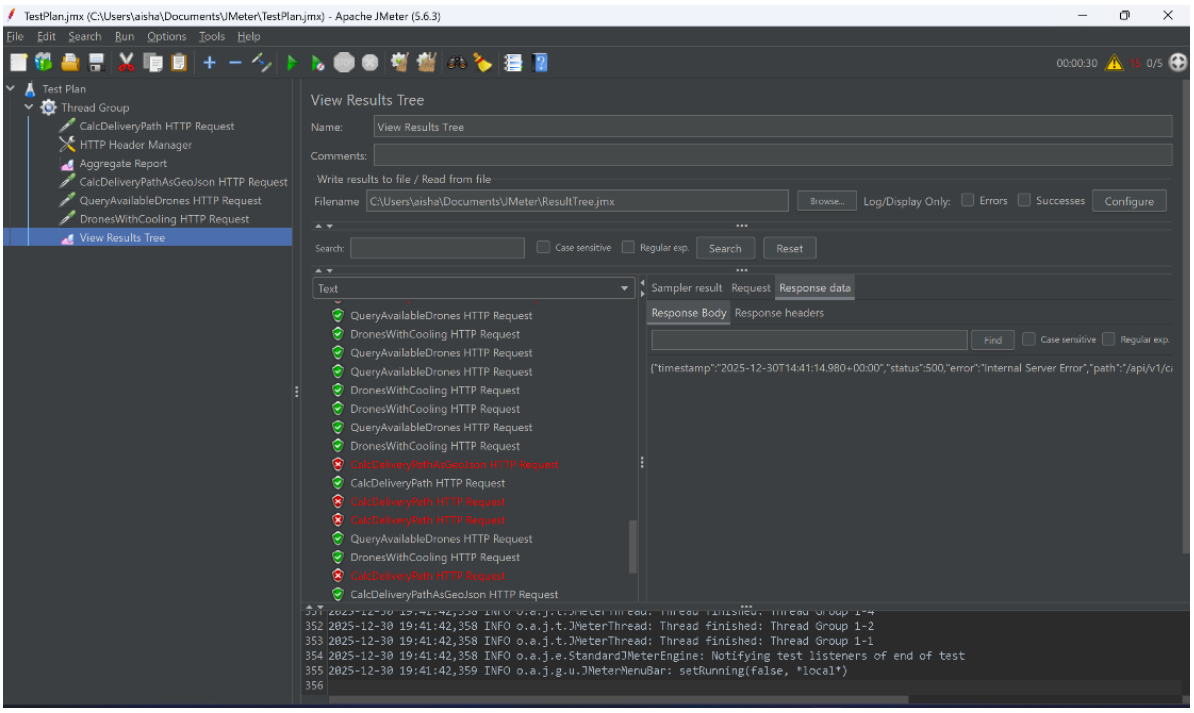
Adequacy Criteria (Statistical Performance Measurement): Adequacy is evaluated using statistical performance metrics such as mean response time, percentile response times and error rate. The latter two collectively demonstrate whether the results are meaningful and can be accepted because mean response time hides spikes.

Results:

- The JMeter test plan used to run this requirement can be found at "*JMeter\TestPlan.jmx*".
- On the first iteration, TC-3 exposed that the system was not safe for concurrent use due to shared mutable state. In particular,
  - private Map<String, Integer> mapDroneToService; (mapping drone IDs to service point IDs) and
  - private Map<Integer, Point> mapServiceIdToServicePoint; (mapping service point IDs to service points)

were recreated and repopulated inside getAvailableDrones. Under concurrent requests, these maps were being partially built, overwritten, and hence contained missing keys if another request modified them mid-execution. This led to an error rate greater than 70% for TC3.

Here is a view of the Results Tree from the JMeter Test Plan run before(i) and after(ii) fix:
i)


TestPlan.jmx (C:\Users\aisha\Documents\JMeter\TestPlan.jmx) - Apache JMeter (5.6.3)

- Test Plan
  - Thread Group
    - CalcDeliveryPath HTTP Request
    - HTTP Header Manager
    - Aggregate Report
    - CalcDeliveryPathAsGeoJson HTTP Request
    - QueryAvailableDrones HTTP Request
    - DronesWithCooling HTTP Request
    - View Results Tree

View Results Tree

Name: View Results Tree

Write results to file / Read from file
Filename: C:\Users\aisha\Documents\JMeter\ResultTree.jmx

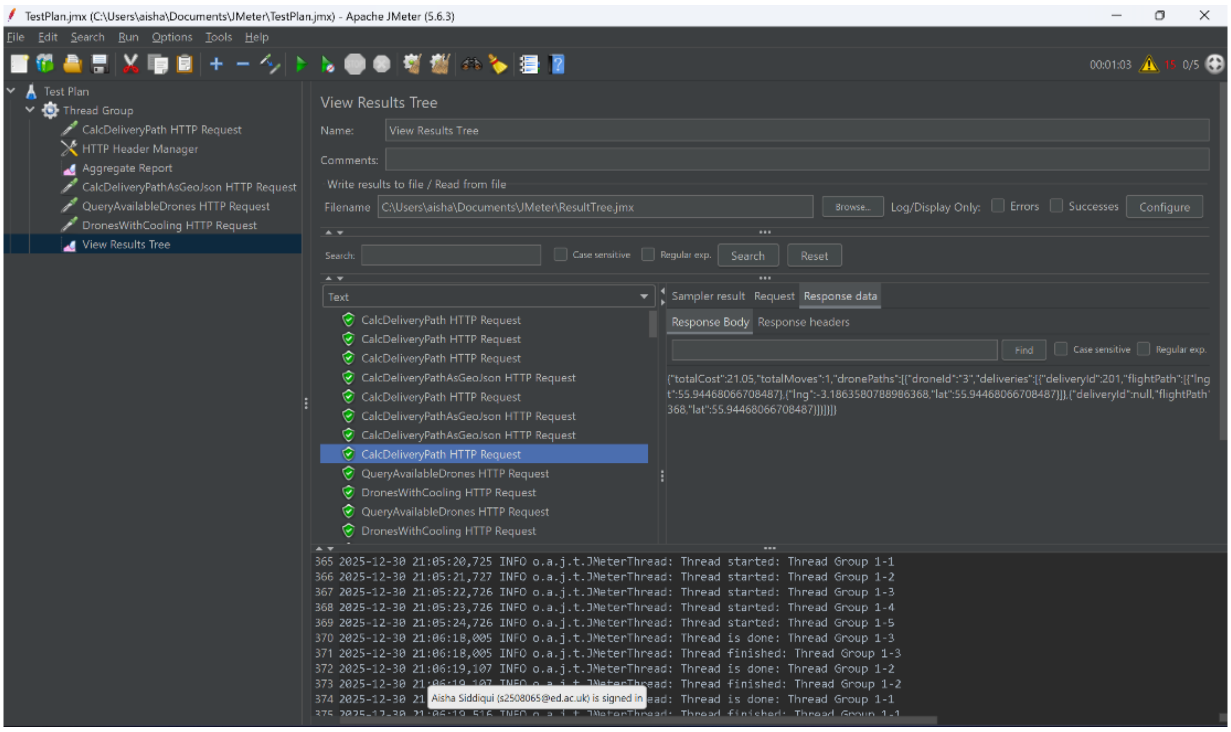
- ✓ QueryAvailableDrones HTTP Request
- ✓ DronesWithCooling HTTP Request
- ✓ QueryAvailableDrones HTTP Request
- ✓ QueryAvailableDrones HTTP Request
- ✓ DronesWithCooling HTTP Request
- ✓ DronesWithCooling HTTP Request
- ✓ QueryAvailableDrones HTTP Request
- ✓ DronesWithCooling HTTP Request
- ✗ CalcDeliveryPathAsGeoJson HTTP Request
- ✓ CalcDeliveryPath HTTP Request
- ✗ CalcDeliveryPath HTTP Request
- ✗ CalcDeliveryPath HTTP Request
- ✓ QueryAvailableDrones HTTP Request
- ✓ DronesWithCooling HTTP Request
- ✗ CalcDeliveryPath HTTP Request
- ✓ CalcDeliveryPathAsGeoJson HTTP Request

Response Body:

```
{"timestamp":"2025-12-30T14:41:14.980+00:00","status":500,"error":"Internal Server Error","path":"/api/v1/c
```

```
2025-12-30 19:41:42,358 INFO o.a.j.t.JMeterThread: Thread finished: Thread Group 1-4
2025-12-30 19:41:42,358 INFO o.a.j.t.JMeterThread: Thread finished: Thread Group 1-2
2025-12-30 19:41:42,358 INFO o.a.j.t.JMeterThread: Thread finished: Thread Group 1-1
2025-12-30 19:41:42,358 INFO o.a.j.e.StandardJMeterEngine: Notifying test listeners of end of test
2025-12-30 19:41:42,359 INFO o.a.j.g.u.JMeterMenuBar: setRunning(false, *local*)
```

ii)


TestPlan.jmx (C:\Users\aisha\Documents\JMeter\TestPlan.jmx) - Apache JMeter (5.6.3)
File Edit Search Run Options Tools Help
00:01:03 15 0/5
Test Plan
Thread Group
CalcDeliveryPath HTTP Request
HTTP Header Manager
Aggregate Report
CalcDeliveryPathAsGeoJson HTTP Request
QueryAvailableDrones HTTP Request
DronesWithCooling HTTP Request
View Results Tree
Browse...
Log/Display Only:
Errors
Successes
Configure
Case sensitive
Regular exp.
Search
Reset
Text
Sampler result
Request
Response data
Response Body
Response headers
Find

View Results Tree

Name: View Results Tree

Comments:

Write results to file / Read from file

Filename C:\Users\aisha\Documents\JMeter\ResultTree.jmx

Search:

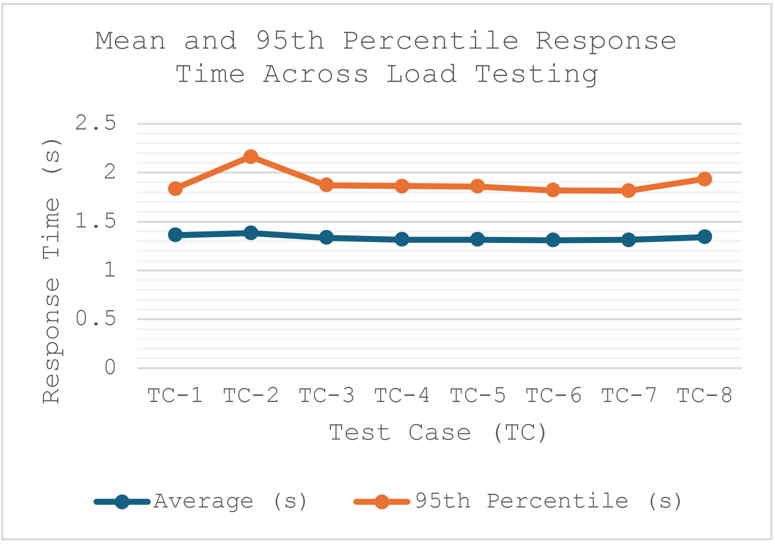
- CalcDeliveryPath HTTP Request
- CalcDeliveryPath HTTP Request
- CalcDeliveryPath HTTP Request
- CalcDeliveryPathAsGeoJson HTTP Request
- CalcDeliveryPath HTTP Request
- CalcDeliveryPathAsGeoJson HTTP Request
- CalcDeliveryPathAsGeoJson HTTP Request
- CalcDeliveryPath HTTP Request
- QueryAvailableDrones HTTP Request
- DronesWithCooling HTTP Request
- QueryAvailableDrones HTTP Request
- DronesWithCooling HTTP Request

```
{"totalCost":21.05,"totalMoves":1,"dronePaths":[{"droneId":"3","deliveries":[{"deliveryId":201,"flightPath":[{"lng
t":55.94468066708487},{"lng":-3.1863580788986368,"lat":55.94468066708487}]},{"deliveryId":null,"flightPath'
368,"lat":55.94468066708487}]}]}]}
```

```
2025-12-30 21:05:20,725 INFO o.a.j.t.JMeterThread: Thread started: Thread Group 1-1
2025-12-30 21:05:21,727 INFO o.a.j.t.JMeterThread: Thread started: Thread Group 1-2
2025-12-30 21:05:22,726 INFO o.a.j.t.JMeterThread: Thread started: Thread Group 1-3
2025-12-30 21:05:23,726 INFO o.a.j.t.JMeterThread: Thread started: Thread Group 1-4
2025-12-30 21:05:24,726 INFO o.a.j.t.JMeterThread: Thread started: Thread Group 1-5
2025-12-30 21:06:18,005 INFO o.a.j.t.JMeterThread: Thread is done: Thread Group 1-3
2025-12-30 21:06:18,005 INFO o.a.j.t.JMeterThread: Thread finished: Thread Group 1-3
2025-12-30 21:06:19,107 INFO o.a.j.t.JMeterThread: Thread is done: Thread Group 1-2
2025-12-30 21:06:19,107 INFO o.a.j.t.JMeterThread: Thread finished: Thread Group 1-2
2025-12-30 21 Aisha Siddiqui (s2508065@ed.ac.uk) is signed in ead: Thread is done: Thread Group 1-1
2025-12-30 21:06:19,516 INFO o.a.j.t.JMeterThread: Thread finished: Thread Group 1-1
```

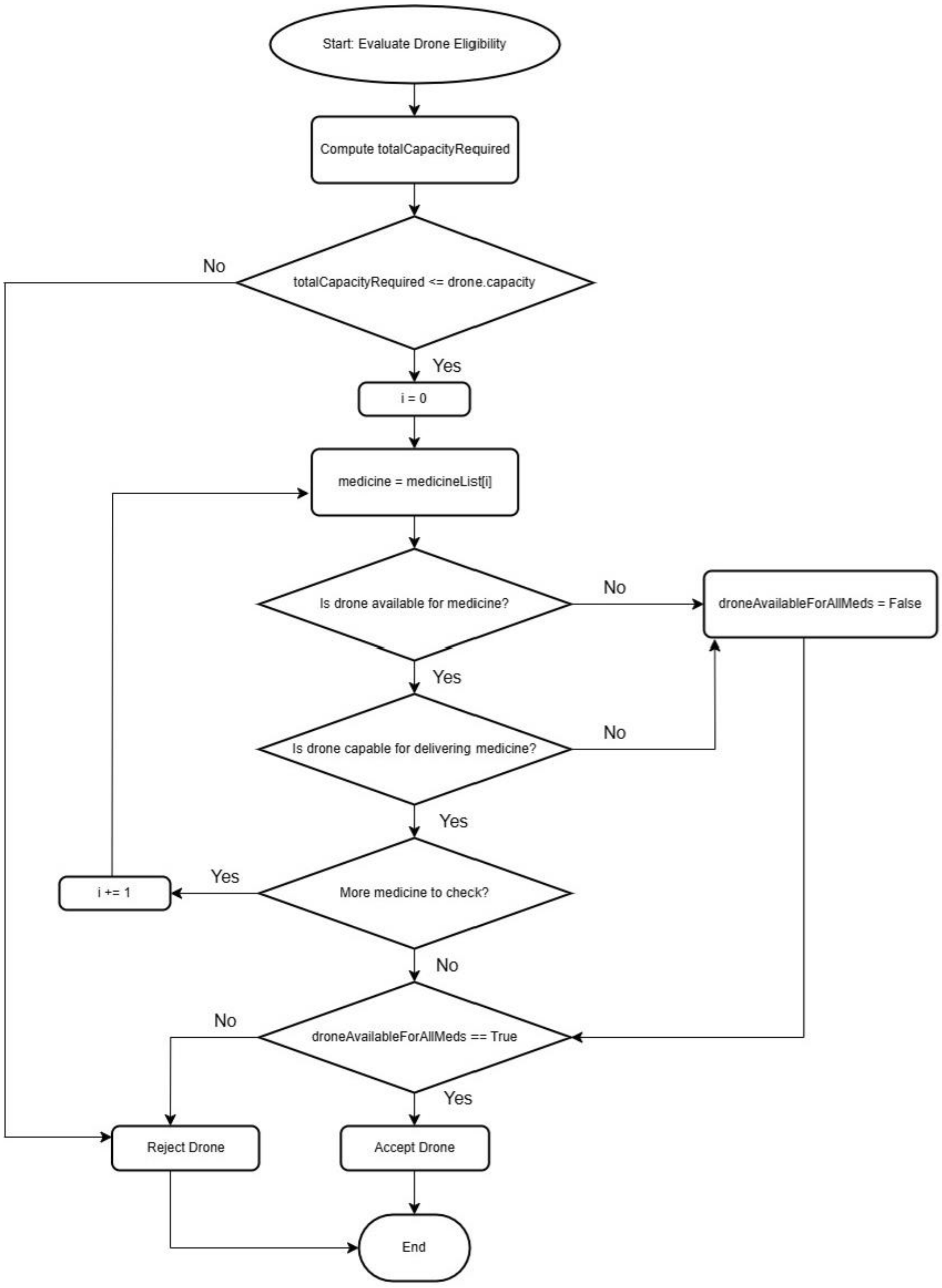
The error rate for all runs of the test plan was 0%. For ease of evaluation, the average response time for all requests in each run, along with its corresponding 95th percentile, was recorded and plotted on a graph.

TC	Average (s)	p95 (s)
TC-1	1.360	1.836
TC-2	1.382	2.164
TC-3	1.336	1.871
TC-4	1.315	1.861
TC-5	1.317	1.859
TC-6	1.308	1.820
TC-7	1.311	1.815
TC-8	1.341	1.934

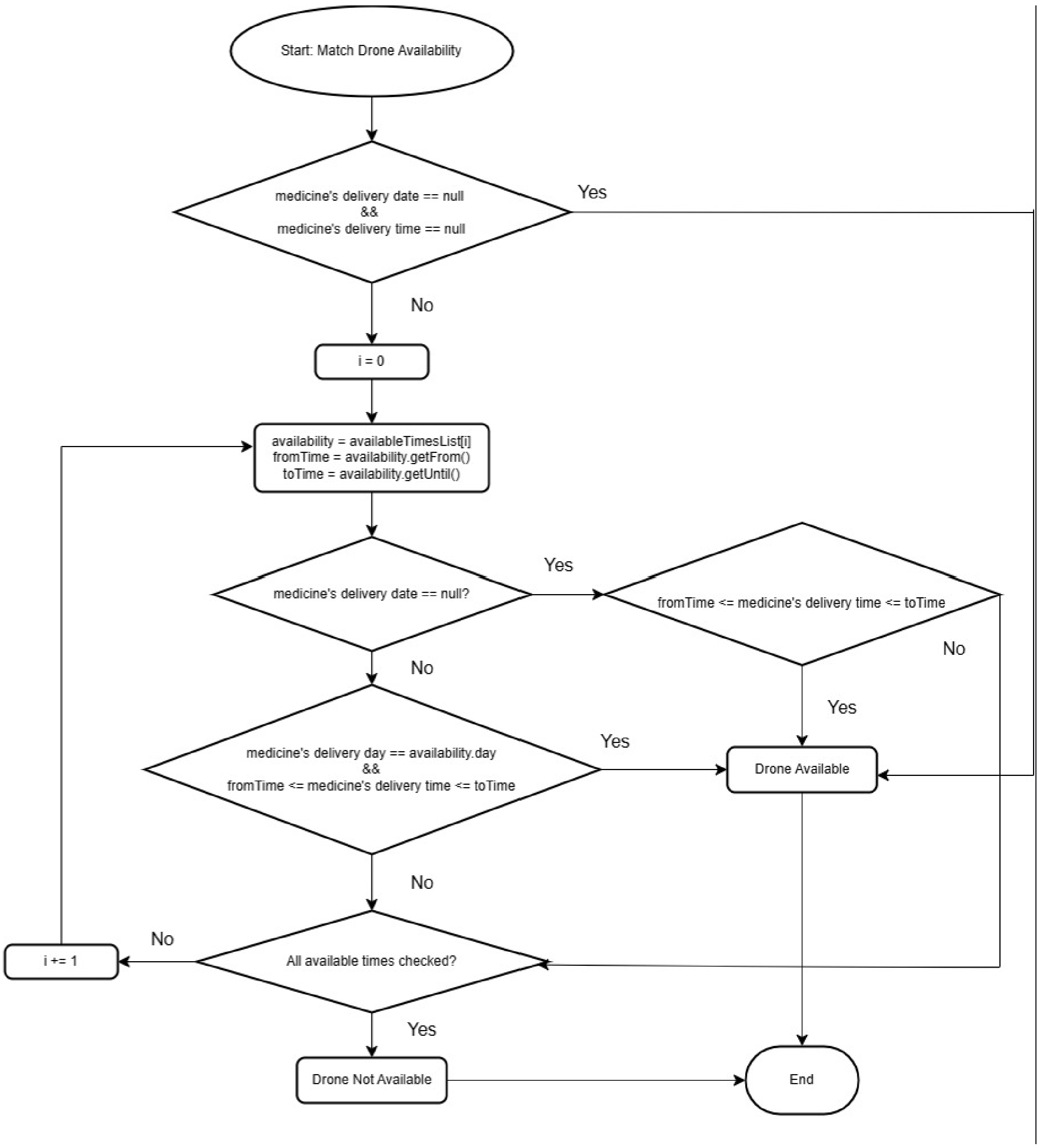


R1.2.1: The system shall assign deliveries only to drones that satisfy cooling/heating, capacity, and availability constraints.

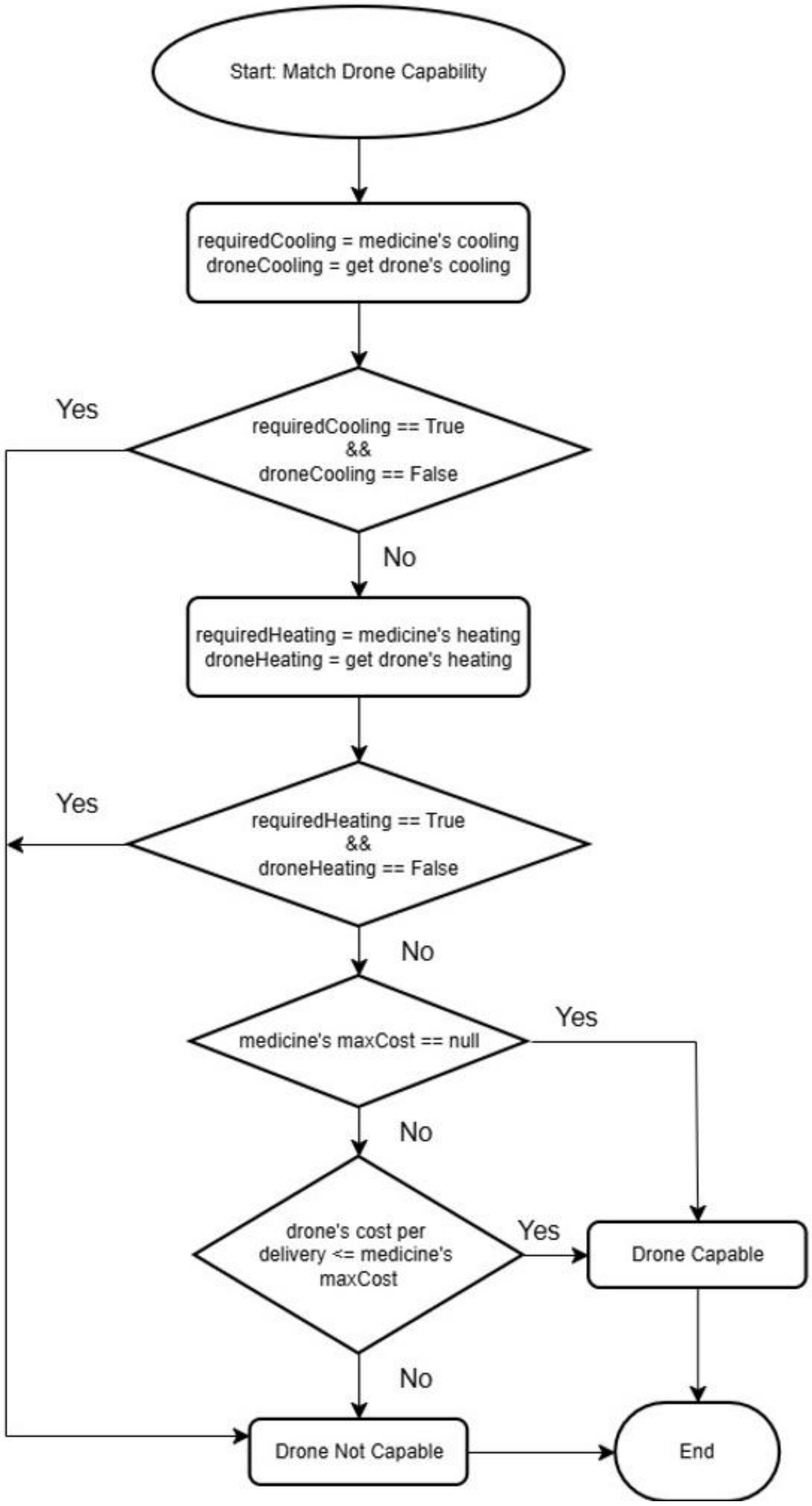
The eligibility of a drone depends on i) the drone's temporal availability and ii) capability and cost constraints. The first flowchart represents the top-level logic.


Start: Evaluate Drone Eligibility
Compute totalCapacityRequired
totalCapacityRequired <= drone.capacity
No
Yes
i = 0
medicine = medicineList[i]
Is drone available for medicine?
No
droneAvailableForAllMeds = False
Yes
Is drone capable for delivering medicine?
No
Yes
More medicine to check?
Yes
i += 1
No
droneAvailableForAllMeds == True
No
Yes
Reject Drone
Accept Drone
End

The second flowchart represents the logic for checking a drone's availability.


Start: Match Drone Availability
medicine's delivery date == null && medicine's delivery time == null
Yes
No
i = 0
availability = availableTimesList[i]
fromTime = availability.getFrom()
toTime = availability.getUntil()
medicine's delivery date == null?
Yes
fromTime <= medicine's delivery time <= toTime
No
Yes
No
medicine's delivery day == availability.day && fromTime <= medicine's delivery time <= toTime
Yes
Drone Available
No
All available times checked?
No
i += 1
Yes
Drone Not Available
End

The third flowchart represents the logic for checking a drone’s capability and cost constraints (cooling, heating, maxCost).


Start: Match Drone Capability
requiredCooling = medicine's cooling
droneCooling = get drone's cooling
requiredCooling == True
&&
droneCooling == False
Yes
No
requiredHeating = medicine's heating
droneHeating = get drone's heating
requiredHeating == True
&&
droneHeating == False
Yes
No
medicine's maxCost == null
Yes
No
drone's cost per delivery <= medicine's maxCost
Yes
No
Drone Capable
Drone Not Capable
End

Testing technique (flowgraph-based testing): A model-based, flowgraph-based testing approach is suitable because the requirement is based on a set of clear decision rules. Flowgraphs make the checks for capacity, availability, and capability explicit and show how a drone is rejected as soon as any constraint fails. This allows test cases to be selected systematically to cover all decision outcome. The approach also reflects the modular structure of the implementation, making the tests easier to relate to the code. We can then use branch testing in hope of covering all the branches outlined in the flowcharts above.

TC	**Capacity OK**	**Availa-bility OK**	**Availability Case**	**Cooling Required**	**Cooling OK**	**Heating Required**	**Heating OK**	**Max Cost Spec**	**Cost <= Max**	**All Meds OK**	**Drone Accepted**
TC-1	Yes	Yes	Date & time both null	No	-	No	-	No	-	Yes	Yes
TC-2	No	-	-	-	-	-	-	-	-	-	No
TC-3	Yes	No	No match	-	-	-	-	-	-	No	No
TC-4	Yes	Yes	Match	Yes	No	-	-	-	-	No	No
TC-5	Yes	Yes	Match	Yes	Yes	Yes	No	-	-	No	No
TC-6	Yes	Yes	Match	Yes	Yes	Yes	Yes	Yes	No	No	No
TC-7	Yes	Yes	Match	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
TC-8	Yes	Yes	Only date null	No	-	No	-	Yes	Yes	No (2nd Med Fails)	No

Adequacy Criteria (Branch Testing): Branch coverage is suitable because the logic contains several decision points, and each has two possible outcomes. The test suite is designed so that each decision is taken both ways (true and false) in at least one test case. This means all acceptance and rejection conditions for a drone are exercised, giving confidence that the constraint logic work as intended if the test cases pass.

R1.2.3: The system shall accept well-formed JSON requests and reject malformed JSON requests. ***Testing technique (systematic partitioning using equivalence classes)***: The input domain is divided into two top-level equivalence classes based on the oracle: well-formed JSON (accepted) and malformed JSON (rejected). The malformed class is further partitioned into equivalence classes representing distinct fault categories: JSON syntax errors, missing required fields, incorrect data types, out-of-bound data, and unexpected fields. Also, assuming only id, requirements.capacity, and delivery (with lat/lng) are required, the test suite then is:

1. **Valid minimal request**
2. **Valid request (includes optional fields)**
3. **JSON syntax errors**
    i) Missing double quotes on a JSON key
4. **Missing required fields**
    i) Missing id test failed on first run because @Valid tag missing so no validation runs.
    Before: calcDeliveryPath(@RequestBody List<@Valid Medicine> medDispatchRec)
    After: calcDeliveryPath(@Valid @RequestBody List<@Valid Medicine> medDispatchRec)
    ii) Missing requirements
    iii) Missing requirements.capacity
    Test failed on first run because @Valid tag missing from Medicine class so cascading validation not enabled.
    Before:
    @JsonProperty("requirements")
    @NotNull(message = "requirements is required")
    private MedRequirements requirements;
    After: @Valid
    @JsonProperty("requirements")
    @NotNull(message = "requirements is required")
    private MedRequirements requirements;
    iv) Missing delivery
    v) Missing delivery.lat (and/or delivery.lng)
5. **Incorrect data types**
    i) requirements.capacity entered as String instead of number (Double)
    ii) id entered as String instead of Integer *(or delivery.lat as String)*
    iii) cooling entered as String instead of Boolean (optional-field type check)
6. **Out-of-bound data (for lat/lng)**
    i) lat/lng valid on the boundary
    ii) out-of-bound lat
    iii) out-of-bound lng
    iv) lat/lng just outside bounds
7. **Unexpected fields**
    i) All required fields present + unknown key-value pair