Testing Evidence and Evaluation LO3

s2298559

January 2026

1 Range of Testing Techniques Used

Most of the tests running are shown at the bottom of the document as screenshots. Tests that produce files, save them in target/test_results as csv files

1.1 Unit Testing

Unit tests were used for deterministic, rule-based logic and pure functions:

- Order validation logic: OrderValidatorTests, OrderValidatorEdgeCaseTests, OrderValidationResultTests
- Geometric helpers and converters: GeoJsonConverterTests, GeoJsonConverterValidityTests
- Low-level path calculation logic: DeliveryPathCalculatorTests, DeliveryPathCalculatorPropertyTests

These tests focus on individual classes in isolation, using synthetic data and stubs where appropriate.

1.2 Integration Testing

Integration tests verify interactions between multiple components while still controlling external dependencies:

- Controller integration via HTTP binding: IlpControllerIntegrationTests
- REST service integration with stubbed remote endpoints: ILPRestServiceIntegrationWithStubbedRemoteTests
- Path calculation using stubbed no-fly zones and central area data

Mock REST servers and Spring's MockMvc were used to test JSON binding, validation annotations, HTTP status codes, and component wiring without relying on live external services.

1.3 System Testing

System tests treat the application as a black box accessed only through HTTP in IlpSystemTests; these tests verify:

- service availability: /isAlive
- correct HTTP error handling for malformed requests
- end-to-end behaviour from HTTP request to HTTP response

This level of testing builds confidence that the deployed service behaves correctly in realistic usage scenarios.

1.4 Property-Based and Statistical Techniques

Several tests adopt a property-style or statistical approach:

- Repeated testing across all 16 permitted drone angles
- Randomised path generation for GeoJSON validity checks
- Invariant checking (e.g. fixed movement distance of 0.00015 degrees)

In addition to example-based and invariant tests, the test suite was extended to include statistical and performance-scaling experiments targeting the path-planning component.

A dedicated performance test suite (DeliveryPathCalculatorPerformanceScalingTests and fuzz-based variants) systematically injected increasing numbers of synthetic no-fly zones (1, 5, 10, 25, 50, 100, 200, 500, 1000, up to 40 000) while varying restaurant starting locations, including realistic Edinburgh locations and extreme long-distance cases (Glasgow, Manchester, London).

Each configuration was executed repeatedly, and execution time was recorded to CSV files for offline analysis. This enabled aggregation of:

- maximum runtime
- average runtime
- variance across repeated trials

This approach goes beyond single threshold checks and provides empirical evidence of scalability trends rather than binary pass/fail outcomes.

1.5 Robustness and Negative Testing

Robustness was explicitly tested using:

- null inputs
- malformed JSON
- missing fields
- invalid numeric ranges

- open (non-enclosed) polygons

Some of these tests are expected to fail and are intentionally retained to document robustness weaknesses, rather than to artificially inflate pass rates.

2 Evaluation Criteria for Testing Adequacy

Different evaluation criteria were applied depending on the requirement and test type:

- Functional Correctness
  - Exact status codes (VALID, INVALID, DELIVERED)
  - Correct validation codes (e.g. TOTAL_INCORRECT, EMPTY_ORDER)
  - Boolean correctness for geometric predicates (isCloseTo, isInRegion)
- Numerical Tolerance
  - Floating-point comparisons use tolerances (typically 1e-9 or 1e-12)
  - Movement distance invariants are checked approximately, not exactly

This avoids false negatives due to floating-point representation.

2.1 Safety and Constraint Invariants (FR3 & FR4)

Rather than checking individual steps only, tests evaluate path-wide invariants:

- paths must not intersect no-fly zones
- central-area polygons must be enclosed
- once the drone enters the central area, it must not leave again

These criteria are pessimistic by design: a single violation fails the test.

2.2 Robustness Expectations

For malformed input, the evaluation criterion is:

- The system must not crash
- A defined HTTP response (typically 400) should be returned

Where direct method calls bypass Spring validation, NullPointerExceptions are documented as design weaknesses rather than test failures.

3 Results of Testing

3.1 Summary of Outcomes

- The majority of unit, integration, and system tests pass.
- Several robustness and edge-case tests fail, particularly where:
  - validation relies on Spring annotations that are bypassed by direct calls
  - assumptions about polygon enclosure are not enforced consistently
  - some boundary cases were underspecified in the original requirements

3.2 Defects and Improvements Identified

Testing directly led to:

- refactoring OrderValidator to allow injection of synthetic restaurant data
- identification of missing null-handling in controller methods
- clarification of expected behaviour for empty paths and malformed GeoJSON inputs

A running log of test execution and fixes was maintained during development.

3.3 Performance and Statistical Results

Performance-scaling results show a clear relationship between:

- number of injected no-fly zones
- distance of the restaurant from the delivery location

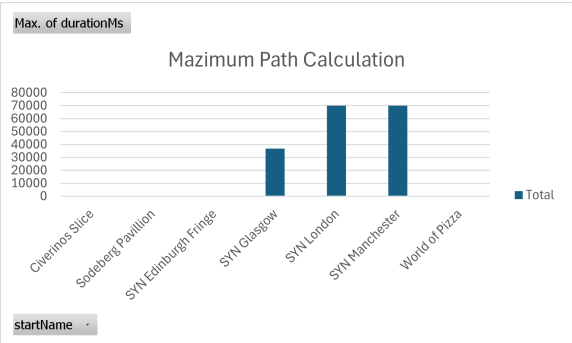

Figure 1 shows the maximum observed path calculation time per restaurant. For local Edinburgh restaurants, computation time remains negligible. For long-distance synthetic restaurants (Manchester and London), runtime increases significantly and exceeds the 60-second guideline once zone density becomes extreme.

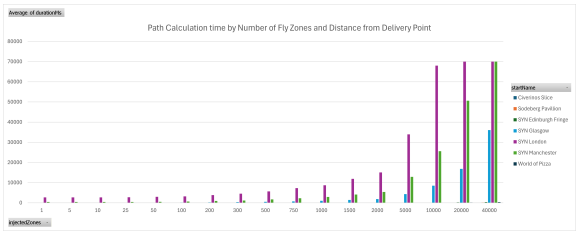

Figure 2 shows the average path calculation time as a function of injected no-fly zones and restaurant distance. Results demonstrate approximately super-linear growth with respect to the number of zones, confirming that performance is sensitive to environmental complexity rather than just travel distance.

These results were generated directly from CSV outputs produced by automated tests, enabling reproducibility and post-hoc analysis.

3.4 Interpretation of Performance Failures

The performance requirement (QR1) is specified as an approximate guideline rather than a hard real-time constraint. The observed violations occur only under synthetic stress scenarios (tens of thousands of no-fly zones and geographically unrealistic restaurants), which exceed the expected operational envelope.

Importantly:

- the system terminates cleanly even in extreme cases
- failures are reported as controlled outcomes (e.g. “No path found to the goal”)
- no infinite loops or uncontrolled crashes were observed

These findings indicate that the implementation is robust for realistic inputs while exposing clear scalability limits under adversarial conditions.

4 Evaluation of Testing Effectiveness

The combination of testing techniques significantly improves confidence that the system meets its requirements:

- Unit tests provide strong confidence in deterministic logic.
- Integration tests reveal issues that unit tests cannot (e.g. JSON binding, validation annotations).
- System tests confirm correct black-box behaviour.
- Property and statistical tests increase confidence across wide input spaces.
- Failing tests provide valuable evidence of limitations and guide future improvements.

While the test suite does not guarantee the absence of defects, the chosen evaluation criteria are appropriate for the safety-critical and data-driven nature of the system. The testing approach balances optimism (assuming correct behaviour within tolerance) with pessimism (rejecting any constraint violations) and explicitly documents known weaknesses.

The inclusion of statistical performance testing substantially strengthens the adequacy of the testing process. Rather than asserting compliance with a single runtime threshold, the tests:

- characterise performance trends
- identify break-points where behaviour degrades
- quantify the effect of environmental complexity

This approach provides stronger evidence than conventional performance testing and supports informed judgement about whether observed limitations represent defects or acceptable trade-offs given the system's intended use.

5 Requirement-to-Test Evidence Mapping

Table 1: Master mapping of functional, quality, and qualitative requirements to implemented test evidence (multi-level, multi-technique).

Req	**What is tested (evidence target)**	**Level / technique**	**Concrete test evidence (classes and methods)**
FR1	Order validation rules: card fields, totals including delivery charge, pizza count limits, menu price correctness, restaurant open-days, and correct INVALID codes.	Unit + boundary + negative tests	`OrderValidatorTests`: `testValidateEmptyOrder()`, `testValidateExceedPizzaLimit()`, `testValidateIncorrectTotalPrice()`, `testValidateIncorrectPizzaPrice()`, `testValidateInvalidCardNumber()`, `testValidateInvalidCVV()`, `testValidateInvalidExpiryDate()`, `testValidateValidOrder_dynamic()`, `testValidateRestaurantClosed_dynamic()` `OrderValidatorEdgeCaseTests`: `testValidateExactlyOnePizza_boundaryShouldBeAllowed()`, `testValidateExactlyFourPizzas_boundaryShouldBeAllowed()`, `testValidatePizzaCountZero_isEmptyOrder()`, `testValidateTotalOffByOnePenny_shouldBeInvalid()`, `testValidateCardNumberNonDigits_shouldBeInvalid()`, `testValidateExpiryMonth13_shouldBeInvalid()`, `testValidateCvvNonDigits_shouldBeInvalid()`, robustness tests for nulls/malformed strings `OrderValidationResultTests`: getters/setters + defaults (consistency)

Continued on next page

Req	What is tested (evidence target)	Level / technique	Concrete test evidence (classes and methods)

FR2	Movement constraints: next position uses fixed step length (0.00015 degrees), permitted angles (16 compass directions), angle validation; hover presence at pickup/delivery points (where applicable).	Unit + integration + property-style loop + tolerance checks	`ILPRestControllerTests`: step-length tolerance tests for `/nextPosition`, angle invalid/edge behaviour
			`IlpControllerIntegrationTests`: `nextPosition_overHttp_stepLengthWithinTolerance_QR2()`, `nextPosition_overHttp_all16Angles_propertyCheck()`
			`IlpSystemTests`: repeated calls smoke/perf
			`DeliveryPathCalculatorTest` (`DeliveryPathCalculatorTests.java`): `testCalculatePathHoveringAtLocations()`
			`DeliveryPathCalculatorPropertyTests`: `testPathHasExactlyTwoHoversAtStartAndEnd_only()`

FR3	Flight path never intersects any no-fly zone polygon; no path point lies inside zones; zones must be enclosed; blocked pickup/destination should refuse delivery with defined message ("No path found to the goal").	Integration-style + invariant testing + synthetic datasets	`DeliveryPathCalculatorTest` (`DeliveryPathCalculatorTests.java`): `testNoFlyZonesAreClosedPolygons_FR3()`, `testRestaurantAndDestinationNotInsideNoFlyZones_FR3()`, `testPathPointsNeverInsideNoFlyZones_FR3()`, `testCalculatePathNoFlyZoneBlocked()`, `testCannotDeliver_whenDestinationInsideNoFlyZone_FR3()`, `testCannotDeliver_whenPickupInsideNoFlyZone_FR3()`
			`DeliveryPathCalculatorPropertyTests`: `testPathSegmentsNeverIntersectNoFlyZones_FR3_property()`, `testCentralAreaAndNoFlyZonesAreClosed_FR3_FR4_property()`
			`ILPRestServiceIntegrationWithStubbedRemoteTests`: zone parsing + closure checks
			`ILPRestControllerTests` / `IlpControllerIntegrationTests`: `/isInRegion` open polygon rejected

FR4	Central area polygon must be enclosed; destination inside central; once drone enters central area, it must not leave again before delivery completes.	Integration + path-history invariant (property-style)	`DeliveryPathCalculatorTest` (`DeliveryPathCalculatorTests.java`): `testCentralAreaIsClosedPolygon_FR4()`, `testDestinationIsInsideCentralArea_FR4()`, `testCentralAreaMustBeClosed_FR4_synthetic()`, `testCentralAreaNotClosed_currentlyNotRejected_documentWeakness_FR4()`
			`DeliveryPathCalculatorPropertyTests`: `testDeliveryLocationIsInsideCentralArea_FR4_property()`, `testPathEntersCentralAreaThenNeverLeaves()`, `testCentralAreaAndNoFlyZonesAreClosed_FR3_FR4_property()`
			`ILPRestServiceIntegrationWithStubbedRemoteTests`: central area parsing + closure checks

FR5	Endpoints return correct status codes and JSON/GeoJSON formats; malformed requests return defined errors (not 500).	System + integration + robustness testing	`IlpSystemTests`: /isAlive, /uuid, /distanceTo, /isCloseTo, /nextPosition, /isInRegion + body/missing body/wrong content-type `IlpControllerIntegrationTests`: missing fields, wrong types, open polygon rejected, numeric shape checks `ILPRestControllerRobustnessTests`: documents direct-call bypass of validation `GeoJsonConverterTests`, `GeoJsonConverterValidityTests`: GeoJSON output correctness + parsing

Continued on next page

Req	What is tested (evidence target)	Level / technique	Concrete test evidence (classes and methods)
QR1	Runtime performance guideline: flight-path calculation completes within expected bound; prevent extreme regressions.	Performance + micro-regression checks	`DeliveryPathCalculatorTest`: `testCalculatePathResponseTime()` (<60s) `IlpSystemTests`: `nextPosition_system_100Calls_completesQuickly()`

QR2	Precision and tolerance: step length 0.00015 degrees, distance/coordinate calculations within floating tolerance; GeoJSON preserves coordinate precision.	Tolerance-based assertions + numeric checks	`ILPRestControllerTests`: step-length tolerance using Euclidean distance
			`IlpControllerIntegrationTests`: `nextPosition_overHttp_stepLengthWithinTolerance_QR2()`
			`GeoJsonConverterValidityTests`: `testGeoJsonPreservesPrecision_QR2()`
			`GeoJsonConverterTests`: coordinate arrays parse and match expected values

QR3	System does not crash on malformed input; returns defined HTTP responses; services tolerate remote failures and malformed remote JSON.	Robustness + negative testing at multiple levels	`IlpSystemTests`: malformed JSON, missing body, wrong content-type `IlpControllerIntegrationTests`: missing fields/wrong types return 400 `ILPRestControllerRobustnessTests`: direct-call NPE behaviour documented + nested-null handling `OrderValidatorEdgeCaseTests`: null lists/payment/date; malformed expiry `IlpRestServiceTests`: RestTemplate exception / null response returns empty list `ILPRestServiceIntegrationWithStubbedRemoteTests`: malformed JSON, 404/500 handling

NR1	Determinism and repeatability: repeated calls with same input yield consistent outputs.	Repeatability / metamorphic checks	`ILPRestControllerTests`: distance symmetry (metamorphic), repeatability tests
			`ILPRestControllerRobustnessTests`: `nextPosition_sameInput_repeatedCalls_sameOutput_NR1()`
			`GeoJsonConverterTests`: `toGeoJson_isDeterministic_sameInputSameOutput()`

NR2	Modularity/testability: ability to test components independently (validator, REST service, controller, path planner, converter).	Design-for-test evidence (separation of concerns)	Separate suites exist per component: `OrderValidator*` (validation) , `IlpRestService*` (remote data access) , `ILPRestController*` (controller logic + robustness) , `DeliveryPathCalculator*` (routing invariants) , `GeoJsonConverter*` (format/precision).
NR3	Dynamic external data retrieval: restaurants/no-fly zones/central area fetched and parsed from external REST service (not hard-coded).	Integration with stubbed remote + live integration	`IlpRestServiceTests`: verifies correct endpoints called and failure behaviour `ILPRestServiceIntegrationWithStubbedRemoteTests`: JSON contract + parsing + closure checks `DeliveryPathCalculatorTest`: uses live ILP REST data for realistic routing constraints

Some validation tests are data-driven (marked *_dynamic) because they depend on restaurant/menu information retrieved via the ILP REST layer. This increases realism but can reduce

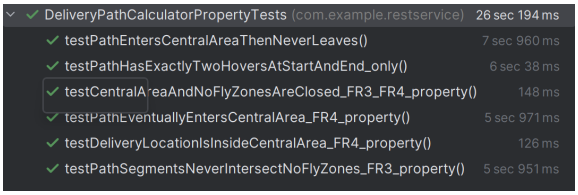

Figure 1: Property Tests

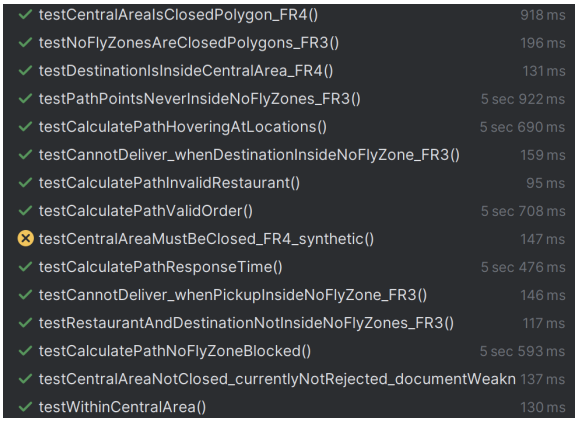

Figure 2: Validity Testing

determinism. Edge-case tests are intentionally pessimistic: they aim to reveal robustness weaknesses such as null-handling and malformed payment formats; failures are retained as evidence of limitations rather than removed.

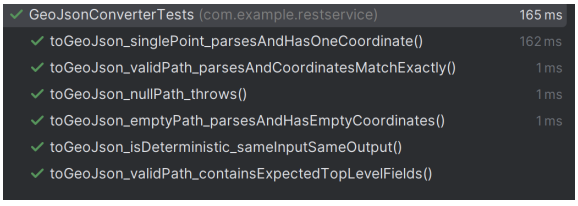

Figure 3: GeoJSON tests

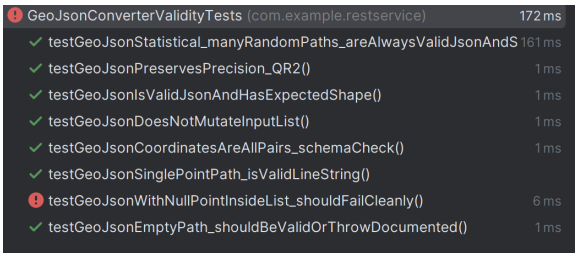

Figure 4: Further GeoJSON Validity tests

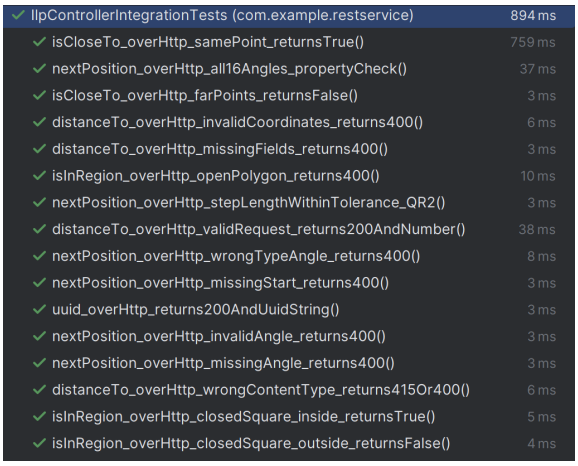

Figure 5: Integration testing

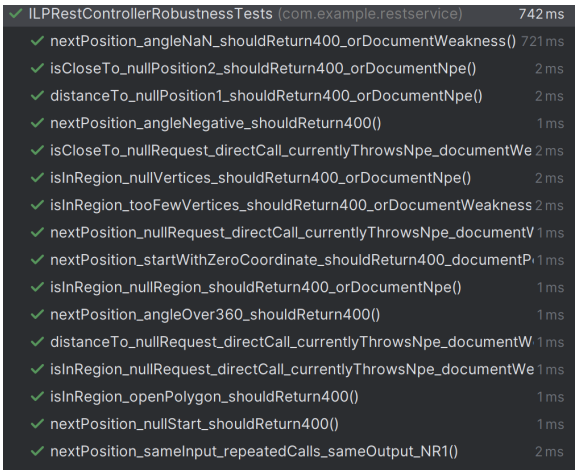

Figure 6: Robustness tests

Figure 7: Unit tests

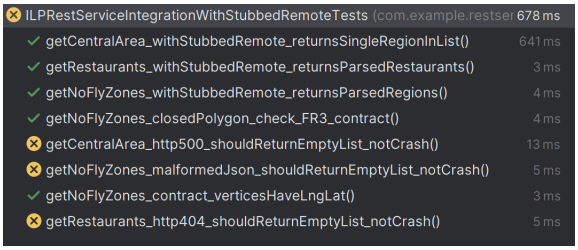

Figure 8: Stubbed tests

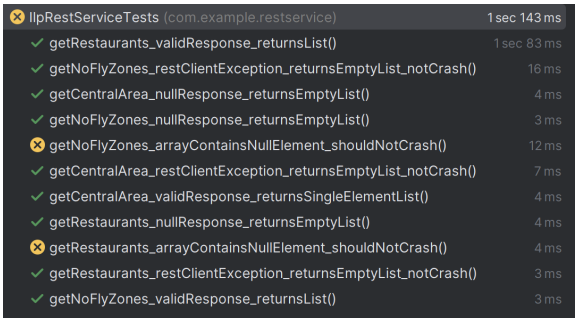

Figure 9: Null Pointer Issues

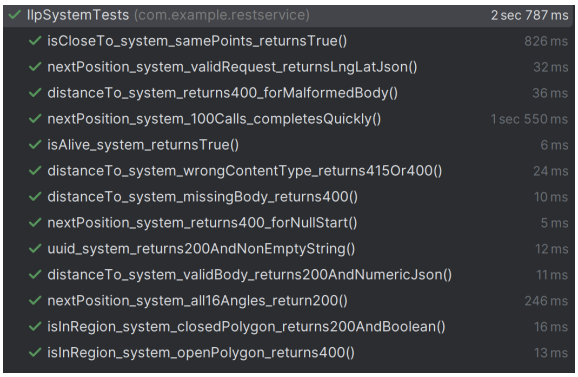

Figure 10: System Testing

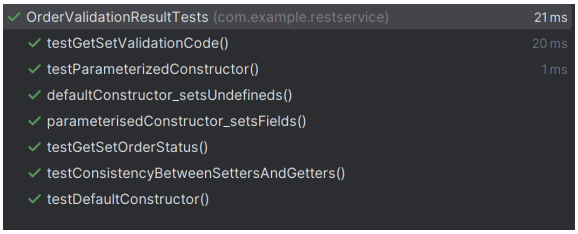

Figure 11: Unit testing

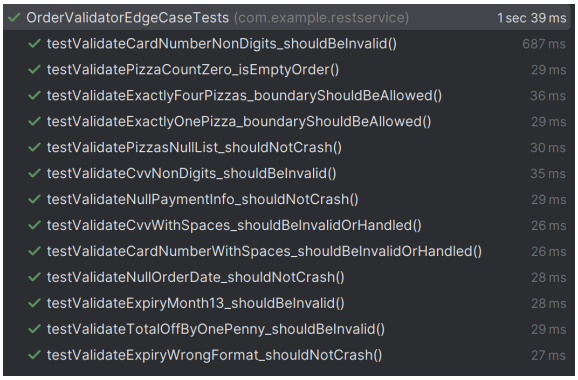

Figure 12: Edge Case Testing

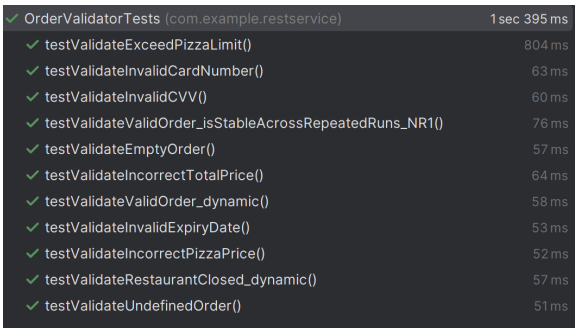

Figure 13: Unit Testing