LO2: Design and implement comprehensive test plans with instrumented code

Introduction

This test plan covers the Spring Boot backend of the Medical Dispatch Delivery Service. The backend exposes REST endpoints under /api/v1 and integrates with an external service (via ILPRestController) to fetch drones, service points, drones-for-service-points, and restricted areas. The front-end visualiser consumes /calcDeliveryPath, but full UI testing is not the primary focus; instead, we treat the UI as a consumer and validate data flow through contract tests.
The plan is designed for a Test-Driven Development (TDD) workflow: core unit tests are created early and expanded as integration and system behaviours emerge.

2.a. Construction of the test plan

Chosen requirements and priority

The portfolio prioritises the requirements selected in LO1, 1.c because they represent the greatest user-impact and technical risk.

System-level (high priority)
FR9: POST /api/v1/calcDeliveryPath returns delivery path(s), total cost, and total moves.
FR10: POST /api/v1/calcDeliveryPathAsGeoJson returns a valid GeoJSON object.
MR3: Typical API response time ≤ 2000ms.
MR4: Deterministic outputs for identical inputs (same request + same ILP inputs).

Integration-level (medium priority)
FR6: Retrieve planning inputs from external ILP service (drones, service points, drones-for-service-points, restricted areas).
FR8: queryAvailableDrones integrates dispatch requirements with drone/service-point availability.
FR11/MR5: Planner avoids restricted areas with at least one movement-step buffer (0.00015 degrees).
MR1: Invalid requests produce client errors rather than undefined behaviour.

Unit-level (low priority)
FR2/MR2: Distance calculations correct to 6 decimal places.
FR4/MR2: nextPosition moves one step (0.00015) and enforces legal angles (multiples of 22.5 degrees).
FR5/MR1: isInRegion correctness and region validation rules (closed polygon, greater than or equal to 4 vertices).

Main priority: FR9/FR10/FR11 represent the core end-to-end dispatch planning capability and are both high value and high risk. MR1 ensures safety/robustness at the API boundary. MR3 and MR4 ensure performance and reproducibility for interactive usage.

Planned test activities

Unit testing (fast feedback, runs continuously)
Primary goal: verify deterministic computations and validation logic early.
FR2/MR2 (distanceTo): equivalence class partitioning + boundary values (same point, close points, far points; lat/lng bounds), numeric tolerance to 6 decimal places.
FR4/MR2 (nextPosition): decision-table style combinations (valid/invalid angle × valid/invalid coords), boundary angles (0, 360, -1, 361), and step-distance assertions.

FR5/MR1 (isInRegion): category-partition for polygon shapes and point locations (inside/outside/boundary), plus invalid polygon tests (not closed, <4 vertices, null/NaN points). Structural testing support: aim for strong branch coverage in validation-heavy components (e.g., ErrorHandler) as a completeness indicator.

Integration testing (subsystems and interfaces)
Primary goal: verify correct interaction between modules and external ILP integration without relying on live ILP availability.
FR6 (ILP retrieval/mapping): integration tests where ILP responses are mocked/stubbed to validate DTO->entity mapping.
FR8 (availability selection): integration tests for queryAvailableDrones using controlled drones and availability fixtures.
FR11/MR5 (avoid restricted areas): integration tests of the planning pipeline with restricted polygons placed between start/goal, verifying the resulting path does not intersect restricted areas.
Fault-based support: error guessing is used to create tests for common integration failures.

System/acceptance testing (end-to-end API outcomes)
Primary goal: validate the user-facing service as a black box over HTTP.
FR9: scenario-based tests that call /calcDeliveryPath and assert path structure + totals are present/consistent for representative dispatch inputs.
FR10: contract tests for /calcDeliveryPathAsGeoJson, verifying GeoJSON structure.
MR1: negative tests posting invalid payloads, expecting 4xx.
MR4: determinism tests repeating requests (with fixed local version of ILP external service) to confirm stable outputs.
MR3: timing tests measuring response time distribution for typical requests.

Where the testing fits in a lifecycle

A TDD-compatible lifecycle for this project:

- Early development: implement unit tests for geometry/validation first (FR2/FR4/FR5).
- During integration: add integration tests when connecting subsystems (ILP retrieval/mapping; availability selection; restricted-area planning pipeline).
- System hardening: add scenario-based system tests for FR9/FR10 plus MR1/MR4 once endpoints are stable.
- Performance checks: implement MR3 timing tests late enough that code paths are stable, but early enough to inform optimisation decisions.
- Regression: retain unit + integration + key system tests as a regression suite; run unit tests on every change and integration/system tests regularly.

2.b. Evaluation of the quality of the test plan

Strengths

- Balanced across levels: unit (correctness), integration (interfaces and component wiring), system (service delivery). This matches LO1, 1.b/1.c and supports confidence.
- Risk-driven focus: planning and restricted-area avoidances are prioritised because they are complex and user-visible.
- Repeatable testing: avoids reliance on live ILP by using mocked data for integration/system tests, improving reliability.
- Regression suitability: scenario tests for FR9/FR10 plus determinism checks (MR4) provide stable anchors as algorithms evolve.

Omissions/vulnerabilities

- External ILP realism risk: mocks can drift from real ILP behaviour (schema changes, network failures, unexpected nulls).
- Geometry completeness risk: testing can demonstrate avoidance for many scenarios, but cannot prove correctness for all polygon configurations and edge cases.
- Performance representativeness risk (MR3): response time depends on machine specs and dataset size, so tests may not reflect real operational load.
- UI usability gap: calcDeliveryPath tests ensure structural compatibility but do not guarantee the visualiser experience is correct.
- Security: as the frontend interacts with the backend via HTTP, there is a large risk for XSS/CSRF attacks, data interception/manipulation, etc.

Mitigations

- Maintain a small set of known ILP payloads as fixtures for contract-style validation.
- Expand restricted-area scenarios using category-partition (concave shapes, narrow corridors, destinations near boundaries).
- Document test environment for MR3 and report the median result.
- Complement contract tests with lightweight exploratory checks in the visualiser.
- Implement security with validation checks via a service such as Kubernetes.

2.c. Instrumentation of the code

To better support MR3 (performance), MR4 (determinism analysis), and the diagnosis of FR11 failures, small additional instrumentation was added in a targeted way.

Instrumentation added (backend)

1. Request correlation ID + end-to-end timing

Added a generated requestId and durationMs measurement inside calcDeliveryPath.
Purpose: supports MR3 timing measurement and makes logs traceable per request (useful when multiple tests are run).

2. Structured diagnostics returned in responses (test-support feature)

Introduced an optional diagnostics model PlanningDiagnostics and added it to CalculatedDeliveryPathRequest as an optional diagnostics field.
Diagnostics include:
requestId, durationMs, reasonCode
counters: legsPlanned, legsFailed, aStarInvocations, straightLineFallbacks
Purpose: allows integration/system tests to assert extra properties (e.g., “no fallbacks occurred”, “legsFailed==0”, duration is recorded) without relying solely on parsing log output.

3. Reason codes for key outcomes

Implemented minimal reasonCode values such as:
NO_AVAILABLE_DRONES (when no drones are eligible)
OK or PLANNED_WITH_WARNINGS (when some legs encountered failures/fallbacks)
Purpose: makes failure analysis clearer and supports LO2 justification for instrumentation.

4. Pathfinding-leg instrumentation and fallback visibility

safeFindPath now records:
timeouts/interruption/execution errors as counted fallbacks
increments counters and logs warnings with requestId
returns a straight-line fallback path when an A* execution fails
Purpose: supports diagnosing FR11/MR5 failures and ensures the system response remains bounded and analysable under pathfinding issues.

2.d. Evaluation of the instrumentation

What the instrumentation supports well

- MR3 performance evidence: durationMs provides a direct measure of backend planning time per request.
- FR11 diagnosis: leg-level counters and fallback counters help explain unexpected routes or empty/short paths.
- Repeatability and debugging: requestId makes it easier to correlate system test requests with logs and outputs.
- Test evidence clarity: returning diagnostics avoids needing manual inspection of log files to justify performance and planning behaviour.
- Returning diagnostics to clients is not desirable in production; it's useful for testing but would be restricted behind a debug flag in real deployment.

Limitations/how instrumentation could be improved

- Diagnostics are not a full tracing system: they summarise behaviour but don't expose deeper causes (e.g., why A* failed: bounding box, max iterations, goal inside restricted area).
  Improvement: add more, explicit failure reason codes (e.g., A_STAR_MAX_ITERATIONS, GOAL_IN_RESTRICTED_AREA) and include them per leg.
- No explicit timestamp or audit trail for the whole request lifecycle: logs have timestamps, but there is no explicit request-start timestamp or stable request structure.
  Improvement: structured JSON logs or a simple structure model.
- Performance timing includes only the planner method: it does not separately time ILP fetch vs planning computation.
  Improvement: instrument controller-level timings for ILP retrieval and compute a breakdown (fetch time vs plan time).