LO5: Conduct reviews, inspections and design and implement automated testing processes

5.a. Identify and apply review criteria to selected parts of the code and identify issues

For this project, I applied lightweight review criteria to a small number of high-risk Java classes where defects would have the highest impact, such as DroneService, FlightPathAlgorithm, ErrorHandler, and the controller layer (ServiceController, ILPRestController). I focused on criteria relevant to testability, correctness, maintainability, and robustness.

Review criteria used

1. Correctness & defensive programming
   - Are null/empty inputs handled consistently?
   - Are errors reported with appropriate HTTP status codes and messages?
   - Are assumptions about external data explicit?
2. Testability & separation of concerns
   - Is domain logic separated from controller logic (so it can be unit tested)?
   - Are external services mockable?
3. Clarity & maintainability
   - Are method responsibilities clear and cohesive?
   - Are names meaningful and consistent?
   - Are magic constants justified (e.g., 0.00015 movement step)?
4. Performance & resource safety
   - Do algorithms have clear limits to avoid runaway runtime?
5. Determinism & reproducibility
   - Will identical inputs produce identical outputs?

Issues identified

- JaCoCo integration issue: initially, I tried to produce a coverage report using JaCoCo; however, no report was produced. Eventually, I instead used IntelliJ's own coverage report maker, which did work.
- Mocking org.slf4j.Logger caused runtime failures under the Java 23 environment (between the time of testing and the time of writing code, an update to Mockito caused instabilities). A review-based fix was to avoid mocking the logger and instead use a real logger instance in tests.
- calcDeliveryPath contains dense logic with many branches (single vs multi delivery, origin selection, fallback behaviour, cost aggregation, and return-leg behaviour). This needs explicit unit/system tests for key branches and careful instrumentation, which I implemented by looking at the branch coverage and implementing diagnostics logging

This review is adequate for identifying the highest impact issues (difficulty with tools, mocking failures, and complex logic areas). It was not a full formal inspection of every class. The approach was pragmatic and aligned with the limited project time. The review's effort was focused on high-risk components and boundary points.

5.b. Construct an appropriate CI pipeline for the software

An appropriate CI pipeline for this medical dispatch planning system would follow a staged process similar to a DevOps workflow, where each stage acts as a quality gate before code progresses.

1. Code Commit
   - Developers commit changes to the GitHub repository.
   - A .github-ci.yml file defines the pipeline stages, runners, and scripts.
   - Every push/merge request automatically starts the pipeline.
2. Build
   - The backend is built using Maven.
   - This stage compiles the project and validates that the codebase is syntactically and structurally correct.
   - This outputs compiled classes and (optionally) a packaged JAR artefact.
   - Failure here prevents later stages from running.
3. Test
   - Executes automated backend tests:
     - unit tests (geometry/validation/query logic)
     - integration tests (ILP retrieval/mapping boundary tests)
     - system-level API tests (MockMvc planner endpoints)
   - This stage is the main correctness check. If tests fail, the pipeline fails immediately.
4. Coverage/Reporting
   - Generates IntelliJ coverage reports after tests.
   - Coverage results are published as pipeline artefacts (HTML/XML) so they can be inspected.
   - Enforce minimum coverage thresholds (See 4.b.).
5. Lint/Static Analysis
   - Runs style and static analysis checks to catch maintainability issues and potential defects early.
   - This helps catch issues that tests may miss (unused code, bad patterns, potential null risks).
6. Deploy
   - If all quality checks pass, the pipeline can deploy to a staging environment.
   - In this context, deploy simply means publishing the built JAR as a pipeline artefact or building a Docker image from the included Dockerfile.
7. Release (Optional / Production)
   - On tagged commits (e.g., v1.0.0), the pipeline can create a release artefact.
   - This stage represents a controlled, ready-for-production step.

If any stage fails, developers fix the issue and re-commit the pipeline, then reruns from the beginning, enforcing the cycle of:
commit -> build -> test -> lint -> deploy -> release.

5.c. Automate some aspects of the testing

Automation is integrated into multiple stages to ensure fast and repeatable feedback:

- Build stage automation
  - Automatically compile the backend to detect errors early (fast feedback).
  - Prevents wasting time running tests when the project doesn't compile.
- Test stage automation
  - Automatically runs all JUnit tests using Maven Surefire:
    - unit tests (e.g., distance/nextPosition/isInRegion)
    - integration tests (mocked ILP boundary)
    - system tests (MockMvc planner endpoints)
  - These collectively check functional correctness at multiple levels.

- Coverage/reporting automation
    - Automatically generates IntelliJ reports.
    - Coverage results are stored as CI artefacts for inspection.
    - Enforce minimum coverage targets.
- Lint/static analysis automation
    - Automatically checks code quality rules (naming, complexity warnings, bug patterns).

This automation ensures that every commit is validated by the same repeatable process, making regressions and accidental breakages much less likely.

5.d. Demonstrate the CI pipeline functions as expected

The CI pipeline is functioning as expected when it reliably fails on real issues and provides clear, traceable artefacts. The kinds of issues it would detect include:

Build-stage failures

- Java compilation errors, missing imports, and incorrect refactors.
- This would be evident in the Maven compilation output in CI logs.

Test-stage failures

- Behaviour regressions (e.g., planner totals change unexpectedly; GeoJSON breaks format).
- Interface/schema mismatches.
- Evidence:
    - Surefire test summary + stack trace in CI logs
    - target/surefire-reports/* attached as artefacts

Coverage gate failures

- Un-tested new logic introduced without adding tests (coverage drops).
- Evidence:
    - IntelliJ summary in the pipeline logs
    - IntelliJ coverage report artefacts showing line/branch drops

Lint/static analysis failures

- Style violations or bug patterns that tests didn't catch.
- Evidence:
    - tool output logs (Checkstyle/SpotBugs/ESLint reports)

Deploy checks

- Ensures only green builds produce deployable programs.
- Evidence:
    - produced JAR/Docker file in pipeline artefacts