Reviews, Inspections, and Automated Testing Process LO5

s2298559

January 2026

1 Selected Requirement for Review

1.1 Requirement Summary

FR3: The computed flight path shall never intersect any polygon defined as a no-fly zone.

This requirement is safety-critical and involves complex geometric reasoning, making it an appropriate candidate for review-based quality assurance.

1.2 Code Under Review

Key code locations implementing FR3 include:

- `DeliveryPathCalculator.calculatePath()`
- geometric intersection logic in helper methods
- path-generation loops enforcing movement constraints

2 Review and Inspection Criteria Applied to Code

2.1 Review Criteria Used

Based on standard inspection practice, the following criteria were applied during manual review:

2.2 Issues Identified Through Review

Several issues were identified that are difficult or impossible to detect through testing alone:

- **Implicit assumptions about polygon validity**: some methods assume polygons are closed without enforcing this at every call site.

Criterion	Rationale
Correctness	Does the logic fully enforce FR3 in all execution paths?
Completeness	Are all no-fly zone constraints implemented (enclosure, blocked endpoints)?
Robustness	How does the code behave under malformed or extreme inputs?
Readability	Is the geometric logic understandable and maintainable?
Defensive programming	Are assumptions documented or enforced by checks?

- **Hidden coupling between geometry and path planning**: intersection logic is embedded within control-flow loops.
- **Limited documentation of invariants**: constraints such as "once inside the Central Area, never leave" are enforced procedurally but not clearly documented.
- **Null-handling assumptions**: some controller methods rely on framework-level validation rather than explicit defensive checks.

3 Selected Measurable Attribute for Review

3.1 Attribute Summary

QR2: Coordinate calculations must respect distance tolerances and floating-point precision limits.

3.2 Testing Limitations Identified

- Tolerance-based assertions confirm correctness only at sampled points.
- Floating-point edge cases (near-collinear points, boundary grazing) remain under-explored.

3.3 Inspection Benefits

Manual inspection reveals:

- repeated floating-point comparisons without a centralised tolerance constant
- duplicated distance calculations that could diverge if modified inconsistently
- lack of documentation explaining numeric stability assumptions

These issues are not failures but represent long-term maintenance and confidence risks that inspections are designed to expose.

4 Implemented CI Pipeline

4.1 Pipeline Overview

A continuous integration pipeline has been implemented using GitHub Actions. The pipeline is deliberately split into:

- a **push-time pipeline** providing fast feedback on every commit, and
- a **nightly pipeline** executing heavier and longer-running tests.

This separation ensures rapid regression detection while still enabling deeper assurance checks without slowing development.

4.2 Push-Time Pipeline Stages

The push-time pipeline executes on every push and pull request and includes:

- Maven build with fail-fast compilation checks
- Static analysis using Checkstyle and SpotBugs
- Unit tests and selected lightweight integration tests

These stages complete quickly and prevent incorrect or low-quality changes from being merged.

4.3 Nightly Pipeline Stages

The nightly pipeline executes scheduled jobs that include:

- system-level HTTP tests
- property-based and invariant-driven geometric tests
- performance and scaling tests unsuitable for push-time execution

Test selection is controlled through Maven configuration to exclude expensive tests from push builds while ensuring regular execution.

5 Automation of Testing

5.1 What Is Automated

- Unit, integration, and system tests are fully automated using JUnit.
- REST dependencies are stubbed to ensure determinism.
- Static analysis and coverage tooling execute automatically in CI.

Aspect	Automation
Code style	Checkstyle
Bug patterns	SpotBugs
Coverage	JaCoCo
Regression	Performance thresholds
Review support	Automated warnings triggering inspection

5.2 Planned Automation Extensions

6 CI Pipeline Effectiveness

The implemented CI pipeline detects different fault classes at appropriate stages:

- logic and API regressions via unit and integration tests
- geometric rule violations via invariant-based tests
- performance regressions via timing thresholds
- maintainability risks via static analysis

Concrete execution evidence is available via GitHub Actions logs, showing successful and failing runs triggered by code changes. Early-stage failures prevent flawed changes from propagating, while later-stage tests provide higher-level behavioural assurance.

7 Evaluation of Automation and Review

Automation provides fast, repeatable regression detection but does not guarantee completeness. Invariant and property-based tests increase confidence but cannot prove correctness over continuous geometric spaces. Performance automation identifies trends rather than worst-case bounds.

For these reasons, automation is treated as a complement to manual review. Static analysis warnings and CI failures are used to trigger targeted inspections, focusing human effort where automated signals indicate increased risk. This combination provides stronger assurance than either approach alone.