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

5.1 – Identify and Apply Review Criteria to Selected Parts of the Code and Identify Issues in the Code:

The code review process was conducted to identify structural, logical, and maintainability issues across the system. Automated tools such as IntelliJ IDEA were used to detect syntax errors, unused variables, formatting issues, and potential code smells, while manual inspection focused on critical modules like order validation, route planning, and no-fly zone enforcement. Detailed observations, including detected dead code, duplicated validation logic, and high cyclomatic complexity in core methods, are documented in **('docs/ci/code_review')**. These findings directly informed following improvements, such as simplifying conditional chains, refactoring repeated logic, and enhancing documentation. By referring to this file, all issues, analysis, and corrective actions are transparently recorded and can be traced back to specific modules and lines of code.

Use of Automated Review Tools

Automated checks were carried out using IntelliJ IDEA to identify syntax errors, unused variables, formatting issues, and potential code smells. This helped maintain consistent coding standards and reduced basic implementation errors early in development. The automated analysis identified various issues across several categories which included unused declarations and code duplication where validation logic was repeated across service classes. These tools were effective for identifying structural problems but were limited in detecting whether the system's logic fully matched functional and safety requirements.

Manual Logical Inspection

To address this limitation, manual reviews focused on key system components such as order validation, route planning, and no-fly zone enforcement. This process examined how decisions were made within the code and whether safety and correctness rules were applied in the correct order and under the right conditions. A significant finding was the identification of dead code through coverage analysis, the isNearBoundary method in RestrictedAreaService showed 0% execution across all test runs, confirming it as unused rather than untested. The review also identified excessive cyclomatic complexity in core pathfinding methods (isInRestrictedArea had complexity of 22), which was subsequently reduced through refactoring. Traditional imperative loops were identified as candidates for functional programming patterns.

Standards and Code Organization

The project followed Java naming conventions, which improved readability and maintainability (camelCase for methods and PascalCase for classes). The codebase was organized into functional packages, separating concerns such as data fetching, validation, and route calculation. This structure made it easier to locate faults and extend functionality. The package structure was refined from an initial flat organisation in a single root package into a layered architecture with dedicated controller, service, dto, model, and config packages. This reorganisation enforced proper separation of concerns and followed Spring Boot best practices. Method coverage improved as a direct result of the review

process identifying and removing three unused methods (isNearBoundary, validateCache, formatTimestamp) that coverage analysis confirmed had zero execution.

Identified Improvements

Some parts of the code contained repeated logic and long conditional chains. These were simplified by using clearer control structures and modern Java features such as streams. Nested if else logic in the priority calculation was replaced with switch expressions to make all cases easier to follow and fully covered. Duplicate validation that appeared in both the controller and service layers was moved into a single, shared location using Spring Boot annotations to remove repetition.

While comments existed in complex sections, documentation linking tests to specific modules was limited. To address this, 34 public methods were updated with full Javadoc comments, describing parameters, return values, exceptions, and the underlying algorithms, including the A* pathfinding logic.

Summary of Review

The systematic review process yielded measurable quality improvements. Method coverage increased from 87% to 100%, 47 total issues were identified and resolved. Code organisation improvements through package restructuring enhanced maintainability and navigability throughout the codebase.

5.2 – Construct an Appropriate CI Pipeline for the Software:

Purpose

The CI pipeline was designed to integrate automated quality checks throughout the development lifecycle. Every push to main or develop, as well as every pull request, triggers the pipeline, ensuring that new code is compiled, tested, and verified automatically. This reduces the risk of introducing errors, enforces coding standards, and allows faster feedback to developers.

Pipeline Workflow

Code Checkout and Environment Setup

- The pipeline begins by checking out the project repository from GitHub, ensuring the latest code is used for each run.
- Java 21 is installed via the pipeline configuration to provide a consistent runtime across all jobs.
- Maven is configured to manage dependencies and cache previously downloaded artifacts, speeding up repeated builds.

Build and Compilation

- Source code in the backend directory is compiled using Maven, creating executable classes while resolving all dependencies.
- Build artifacts are packaged for testing and deployment.
- Build logs are preserved to trace compilation issues or dependency problems efficiently.

Automated Testing

- **Unit Tests:** Validate the correctness of individual classes and methods, such as DistanceService calculations and OrderValidator logic.
- **Integration Tests:** Verify that services interact correctly, including order handling, route planning, and no-fly zone enforcement.
- **System Tests:** Run end to end validations like PathValiditySystemTest and PerformanceSystemTest to ensure correct behaviour under realistic conditions.
- **Coverage Checks:** JaCoCo generates code coverage reports, and minimum thresholds are enforced automatically.

Performance Validation

- A separate job runs performance-specific tests, including PathfindingPerformanceTest and PositioningAccuracyTest, simulating high load conditions to evaluate efficiency and timing of critical algorithms.

Deployment Preparation

- Once all tests pass, the backend is packaged, and a Docker image is built for deployment.
- Containerisation ensures consistent execution across different environments and supports portability and scalability.
- The Docker image is tested to verify that it launches correctly

The CI pipeline is organised into parallel jobs for build/testing, code quality, and performance, reflecting the modular structure of the project. Unit and integration tests provide rapid feedback, while system and performance tests validate real-world behaviour. Static analysis ensures code maintainability, and containerisation guarantees environment consistency for deployment. By embedding these steps directly into the development workflow, the pipeline enforces quality, reduces manual testing effort, and provides actionable feedback to developers early and continuously. The full workflow was then implemented into **'.github/workflows/ci.yml'**.

5.3 – Automate Some Aspects of the Testing:

Purpose and Approach

Automated testing was implemented to improve reliability, reduce manual effort, and ensure that every code change could be quickly validated. GitHub Actions was used to create the workflow, integrating multiple test types and automatically generating coverage and performance metrics. This approach allows testing to scale with the project while maintaining consistent quality checks. Evidence of this can be found within **'.github/workflows/ci.yml'**.

Test Execution and Coverage

The pipeline automatically executes a variety of tests across the software:

- **Unit and Component Validation**: Core modules, such as DeliveryPlannerService and DistanceService, are tested in isolation to verify individual behaviors and boundary conditions. Parameterised tests were used to efficiently evaluate multiple input scenarios, including edge cases and invalid inputs.

- **Integration and Scenario Testing**: Interactions between services, such as order intake, route calculation, and restricted area validation, are validated through integration tests. Dynamic testing was introduced, allowing realistic operational scenarios to be evaluated without relying on static test data.
- **Performance and System Behaviour**: System tests measure timing, scalability, and reliability. For example, pathfinding and order validation are tested against large datasets to confirm response times remain within expected thresholds. These tests also provide early warning of potential bottlenecks or failures under load.

Dynamic and Repeatable Testing Features

- **Dynamic Data**: Using real or simulated datasets ensures that tests reflect realistic conditions, uncovering edge cases that might be missed with manually created data.
- **Reusable Test Fixtures**: Shared setup methods allow tests to be executed consistently, reducing redundancy.
- **Detailed Assertions**: Assertions include meaningful error messages to make failure diagnosis faster and easier.
- **Automated Reporting**: Coverage reports (JaCoCo) are generated automatically and enforce minimum thresholds, providing ongoing insight into testing effectiveness.

5.4 – Demonstrate the CI Pipeline Functions as Expected:

The CI pipeline ensures that each stage of development is automatically verified, from code submission to deployment. Evidence can be found in **'docs/ci/pipeline.pdf'**. All of the pipelines behaves as expected as demonstrated in pipeline.pdf with the following:

Observing Pipeline Behaviour

- **Code Integration:** Pull requests trigger automated checks, ensuring that changes meet coding standards and pass initial tests before merging.
- **Build Verification:** Each push triggers a Maven build to compile the code and package it into a Docker image, catching compilation errors or missing dependencies early.
- **Automated Testing:** Unit, integration, system, and performance tests run in sequence. Failures are reported immediately, preventing problematic changes from progressing further.
- **Deployment Check:** Docker images are built and tested to confirm that the software runs in a clean, isolated environment. This guarantees consistency between development, testing, and production setups.

Identifying and Addressing Issues

- Failures in any stage, such as test errors or build issues, are logged and reported. This allows rapid identification and resolution of problems before they affect other developers or the production environment.
- Static code analysis identifies potential quality issues such as unused variables or duplicated logic, which can then be refactored.

Automating the pipeline reduces manual effort, improves repeatability, and maintains high confidence in system stability. Developers can focus on implementing features, knowing that integration, testing, and deployment are continuously monitored and validated.