

Learning Outcome 5: CI/CD Pipeline

5.1 Reviewing Code:

Constant code review was of utmost importance during the creation of this project. There were multiple ways I performed it:

- **IntelliJ's built-in code review tools**; highlighting, syntactical error detection, warnings and variable type checking.
- **RubberDucks development**: Every time I finished major functionality; I would walk through the code and describe it out-loud to a rubber duck (a Lego figurine actually). I would attempt to explain the rational and expected behaviour of each section. This helped me identify any logical bugs.
- **Pair Programming**: When I was stuck on a particular bug, I would ask to Pair Program with my flatmate, a superior coder to me. He would outline what we had to do, and I would write the code. This helped me write higher quality code, as it was more intentional and well-considered.

The code itself was good, high-quality code:

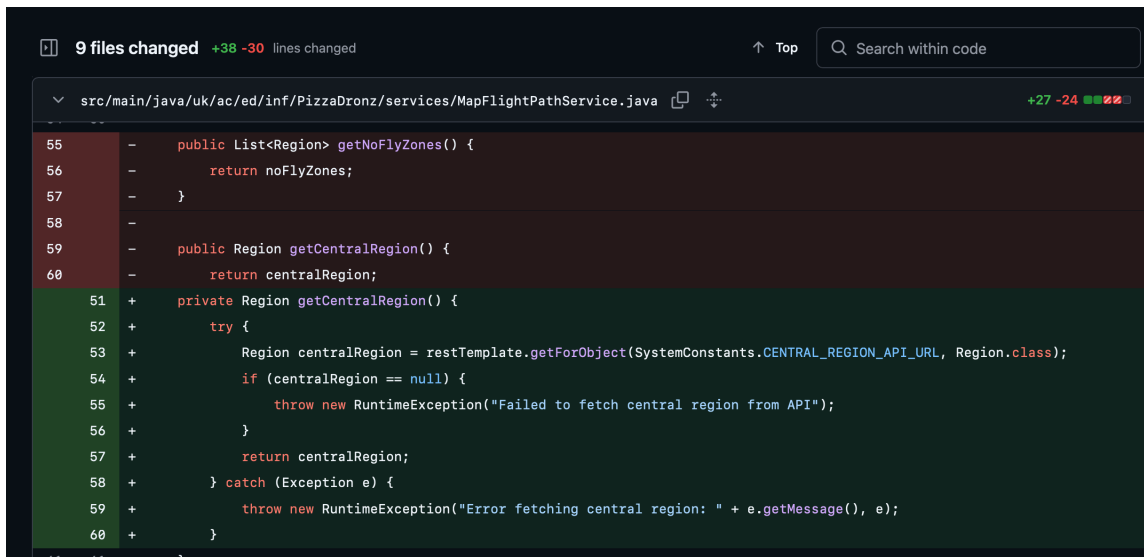
- **Documentation**: It was well commented, with doc strings explaining functions where appropriate.
 - o That said, the tests lacked comments.
- **Naming**: Variable and function naming was clear and high-entropy. They clearly described what the purpose of it was, with little room for confusion.
- **Structure**: The code was well structured. By splitting it up into relevant subfolders (as is industry standard), development was quicker and easier.
- **Java Conventions**: The code strictly follows Java's formatting and naming conventions (camelCase for variable names and functions, PascalCase for Class names etc).
- **OOP Adherence**: The code strictly adheres to SOLID and OOP principals. Basic data types, like Lng and Lat, as well as certain request bodies (e.g. the body of the inRegionRequest) received their own class. Validation of any type was always done at the class level, which allowed for reusing logic in a clean and maintainable way.
- **Complicated A***: More effort could have been placed in explaining the A* algorithm in the documentation, as well as the optimisations taken.

5.2 CI/CD Pipeline:

Due to the size and complexity of this project, a robust CI/CD pipeline was required. The components were:

- a) **Version Control**: A central repository to store, manage and track changes in a codebase.
 - I used GitHub for version control. When I inevitably made changes that broke the entire project, I was able to just roll back to a previous working commit. The branches

functionality was also heavily utilised. When I was working on new functionality, I would create a new branch. When the functionality was complete, I would merge that branch into main. This helped ensure exclusively full, proper working code was on the main branch. Here is an example of “code diffs”, quickly checking for when likely bugs were implemented.



```
9 files changed +38 -30 lines changed
src/main/java/uk/ac/ed/inf/PizzaDronz/services/MapFlightPathService.java +27 -24

55 - public List<Region> getNoFlyZones() {
56 -     return noFlyZones;
57 - }
58 -
59 - public Region getCentralRegion() {
60 -     return centralRegion;
61 - }
51 + private Region getCentralRegion() {
52 +     try {
53 +         Region centralRegion = restTemplate.getForObject(SystemConstants.CENTRAL_REGION_API_URL, Region.class);
54 +         if (centralRegion == null) {
55 +             throw new RuntimeException("Failed to fetch central region from API");
56 +         }
57 +         return centralRegion;
58 +     } catch (Exception e) {
59 +         throw new RuntimeException("Error fetching central region: " + e.getMessage(), e);
60 +     }
61 + }
```

- b) **Build:** This involved building the Java code into an executable (using Docker). This was immediately automated using GitHub actions and workflows (explained below).
- c) **Tests:** Initially, after code edits, I would manually run tests. This became cumbersome, so I automated this as well. Now, upon any code changes and commits, all of the tests automatically run (explained below).
- d) **Deployment:** The program only needed to be deployed locally. I utilised Docker to quickly and easily containerise the application. This meant that it could be packaged up and run on a wide variety of hardware.

5.3, 5.4 Automation & Examples:

The CI/CD Pipeline was as follows:

1. **GitHub:** The code was hosted on GitHub. I created a GitHub Workflow using Yaml.
2. **Automatic Testing:** Whenever new code was committed, GitHub would automatically spin up an instance to run the tests.
3. **Automatic Reporting:** If there was an error with the tests, I received an automatic email notification, as well as GitHub app notification, alerting me.
4. **Automatic Verification:** When the tests all passed, it was reported in the app.

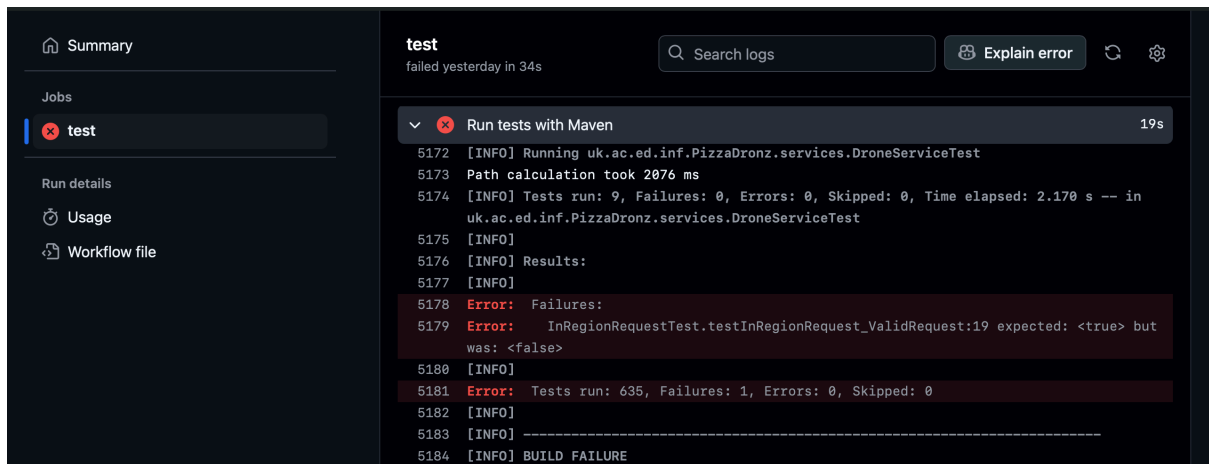


Figure 1. GitHub Action with Failed Tests

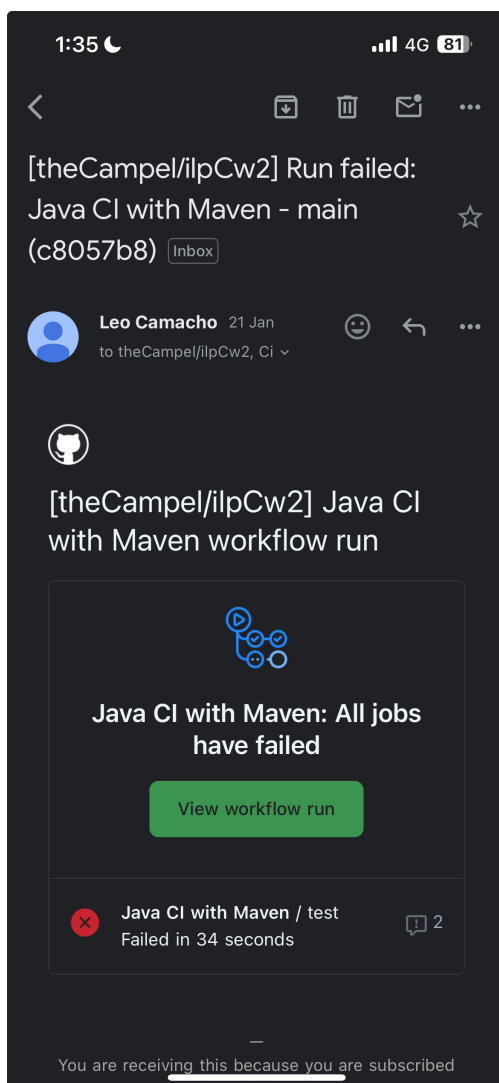


Figure 2. Email Notification of Failed Test