Pipeline Design:

In my pipeline, there will be four stages[1]. These are:

- Source
- Build
- Test
- Deploy

First, the 'Source' stage refers to tracking changes made to the code base. Since I am using GitHub, the pipeline will be activated when I push a new commit, and subsequent phases will be executed.

The 'Build' stage deals with building the artifact and making sure that the correct plugins are implemented and that there are no build errors prior to test execution. For this task, I will use Jenkins as my tool for the following reasons: it is portable to all major platforms, errors are reported and can be fixed immediately, and "carries thousands of plugins which make CI/CD operations much simpler"[2]. My system uses Maven to build its tools, thus Jenkins can take advantage of this as it is able to integrate with Maven.

The 'Test' stage includes the unit, integration, and system level tests that I have laid out in previous documents. The tool that will be used here is JUnit, as Jenkins can integrate with it to run tests automatically. JUnit is appropriate for this pipeline because not only is it supported by almost all IDEs, but it also can efficiently detect errors as well as being easy to read.

Finally, the 'Deploy' stage signals that the altered codebase has successfully built its plugins, passed all the given tests, and is ready to be deployed into a real-life environment. A deployment strategy that could be useful here is the 'Blue-green deployment strategy'. This is a model used to transfer "traffic from a previous version to a new version"[3], and "helps to reduce downtimes and risks"[4]. I thought that this model was advantageous to the design as my system deals with streams of users making requests and the system producing responses to them, both of which are essential and should not be lost. Since the handover from a 'Blue' environment to a 'Green' one is seamless, users will not experience any downtime and even if there is an error in the new environment, it is easy to roll back to a previous version and work on a fix.

Issues with the pipeline:

Although Jenkins is widely used, it is not the easiest to maintain and requires expert knowledge to operate it. This falls under personnel risk, as if an employee who is tasked with maintaining the server leaves, then it will be difficult for anyone else to do their job. This could be remedied by

[1] How to set up a Continuous Integration & Delivery Pipeline, Shanika Wickramasinghe, bmc blogs, https://www.bmc.com/blogs/ci-cd-pipeline-setup/
[2] Maven vs Jenkins: Key differences, Shreya Bose, BrowserStack, https://www.browserstack.com/guide/maven-vs-jenkins
[3] Blue-Green Deployment: An introduction, Stephen Watts, bmc blogs, https://www.bmc.com/blogs/blue-green-deployment/
[4] Blue-Green Deployment: An introduction, Stephen Watts, bmc blogs, https://www.bmc.com/blogs/blue-green-deployment/

creating a document detailing the job in detail as well as having multiple employees undergo training to avoid an absence of talent.

A disadvantage to the Blue-Green deployment strategy is that due to the instant transfer from one environment to the other, user requests that occur at that exact time could be lost, which is something that should be prevented. One potential fix for this issue is to put the new environment into a read-only mode first to ensure all requests are safely read, and then switching to read-write.

Levels of testing:

For this pipeline, the levels of testing required are as follows:

- Unit Testing
  - LngLat
  - Card
- Integration Testing:
  - Polygon
  - Graph
  - OrderPizza
- Performance Testing
  - Visibility graph generation
  - A-Star search
- System Testing
  - App

In a hypothetical scenario where this system is pushed to a live environment, there are a couple of additional tests that the pipeline would perform. One example of this is security testing, and this would ensure that customer data such as credit card detail and user information do not get leaked and that the source code is hidden from outside view.

Another type of test that I would employ is spike testing. This form of testing is used to check the resilience of a system when there is a sudden spike in the number of users currently using the platform. Since my system is a pizza delivery service, it is expected that there will be surges during certain times – lunch, dinner, and perhaps during the late hours too. Thus, it will be imperative to carry out some sort of spike testing during the CI pipeline so that the system does not go down during times where it is most needed.

Furthermore, some tests should be executed before others, i.e., there should be dependencies between tests. ‘x -> y’ signifies that test suite x should be run before y. Here is a list of dependencies that the pipeline should obey to follow a bottom-up approach to testing my system.

- LngLat → Graph -> App
- LngLat → Polygon → App
- LngLat → PathFinder → App
- Card → Order → App
- Performance (Separate)

Issue Identification by the Pipeline:

During the build stage, one issue that Jenkins will identify is if the Java version is not compatible. Since Jenkins requires Java version 1.8 exactly, any other version will result in an error and thus the pipeline will stop this commit at this stage. A solution to this would be to set JAVA_HOME to version 1.8. Another issue could be a timeout error when pushing to GitHub, where Jenkins will not continue its tasks after fetching changes from GitHub. The pipeline is notified of this error by Jenkins, and a potential fix could be to either downgrade Jenkins to version 2.0.3, or also the timeout could be increased to allow the process to complete, even if it takes a long time.

After the build stage is passed, the next hurdle to overcome in the pipeline would be the test phase, where unit, integration, system, and performance tests are run. As written above, there is a certain hierarchy of tests such that some test suites must be run before the next ones are executed. For example, the tests for Order must be run after tests for Card since they rely on Card being sound and complete. Issues that arise from this part of the pipeline would be if the tests do not pass and the system behaves incorrectly. These would be caught by JUnit, and more specifically, the use of *assert* statements in my test suites. I used a combination of *assert, assertEquals* and *assertNull* to check that test outputs were as expected, and if these assertions fail, then JUnit gives a concise report on which test failed so that I can fix the bugs.