***ILP Inspection and Review***

# **Review criteria**

One of review techniques used as part of the code review process is ***checklist review***. Using this technique, we create a checklist of all the things the program should be able to do; in our case, this was simply our list of requirements from the requirements specification document. The code was ran multiple times to check that the requirements at system level were met, and the test suite served to cover the requirements at unit and integration level. Additionally, manual testing also helped with verifying some of the requirements, particularly the command line input validation. This review served as a final ***functionality check*** before deployment. Some of the issues identified during this review were that there is a small delay between when the system finishes execution, and the output files appearing in the appropriate directory. However, this was a very minor delay (around 5 seconds) and does not hinder our system performance (the performance requirement is still met).

Another review technique used as part of the code review process is ***ad-hoc review***. This technique states that there is no pre-defined plan as to what should be done, but rather just an inspection of the overall software. This technique was used as part of the ***code quality review*** to ensure ***readability***, ***appropriate code documentation***, and ***check for any missing Maven dependencies/ other issues that may arise***. During this review, we ensured the code has appropriate Javadoc documentation for all classes, as well as all public methods in each class, and that the naming conventions (such as using camelCase for naming variables) were followed. Some of the issues identified during this review were that some Maven dependencies were missing, which was causing the creation of the JAR file to fail. This was, however, corrected, and the software functions as expected.

# **Construct the pipeline**

The following steps showcase how I would go about constructing a Continuous Integration (CI) pipeline for my chosen project.

1. **Set up the version control system**

Firstly, I would set up a version control system (VCS) to keep track all changes made to my project. This can be done by creating a repository on a hosting platform like GitHub or GitLab. In my case, I have opted for GitHub.

1. **Choose a continuous integration (CI) tool**

After setting up the VCS, I would choose a CI tool to automatically build the code and run all test files whenever a change is made to my project. One option would be to use Jenkins, since it is a well-known, open-source CI tool that is easy to use. Additionally, it supports a wide range of plugins, is compatible with Windows, MacOS and Linux, and has a large community that provides support if needed.

1. **Install and set up Jenkins**

I would install and configure Jenkins on my own device and set up users and permissions.

1. **Create and configure a Jenkins job**

After setting everything up, I would create a Jenkins job for my project. This would be configured to build when changes are pushed to my version control system repository. The configuration would specify the necessary build steps in the Jenkins job configuration, which include checking code from the VCS repository, running the ‘mvn clean install’ command to compile, package and test the code, and archiving the build artifacts.

1. **Configure the test suite.**

The Jenkins job configuration would contain all the necessary steps, such as running the unit and integration tests using the ‘mvn test’ and ‘mvn verify’ commands and running functional plus end-to-end tests using a separate set of commands.

1. **Deploy the staging environment**

Once the build is successful and all tests pass, I would deploy the software to a staging environment (for further testing) using the Jenkins Deploy. The further testing would involve additional tests such as performance tests among others. For these performance tests, one potential tool that could be used would be Apache JMeter.

1. **Release and monitor**

Once the tests pass, the Jenkins job would release the code changes to the production environment. The production environment would then be monitored for any issues. If issues do arise, alerts (such as emails) would be set up to notify the developer.

# **Demonstrate the pipeline works as expected**

Each time a change is made, the build server will pull the latest version of the code from the VCS and run all the tests on the pipeline. The tests vary in level, from unit tests all the way to system tests, and also include some tests for measurable qualities such as performance. If any of the tests fail, or some other issue occurs, the build will fail, and the developers will be notified that a failure occurred. If all the tests pass, then the server will deploy the code to a staging environment (for further testing) and the developers will be notified of the deployment.

Having considered how the pipeline operates, I will now further discuss the issues that could arise as part of the development process (i.e., what might cause the build server of our CI pipeline to fail).

Below is a list of some of the types of issues that our proposed CI pipeline would discover:

* **Failed tests:** Here the failure(s) will be identified by the individual tests themselves (unit, integration, system, etc.). Since the CI pipeline is configured so that a successful build occurs only when all tests have passed, a failed test will cause the build to fail.
* **Syntax errors:** These are errors that prevent the code from being compiled and would be immediately identified by the CI pipeline when trying to build.
* **Deployment failures:** These failures could arise during the deployment stage and would essentially consist in preventing the software from getting successfully deployed to the staging environment. One way to mitigate this would be done through the use of the Jenkins Deploy Plugin, which will fail the build if any issues occur during the deployment process.
* **Misconfigured/incompatible dependencies:** Some code changes could introduce new bugs that cause conflicts with the current dependencies of the project. These issues will be identified by Maven and can be found inside the POM file. Therefore, issues in this file will prevent the code from successfully compiling, so the pipeline would fail the build.
* **Performance issues:** Some of the changes in the code might cause the system to perform poorly (and potentially fail to meet our performance requirement). Here, the additional tools (such as Apache JMeter) integrated in the CI pipeline would help in identifying the issue.
* **Code style violations:** Some code changes could contain issues such as inconsistent formatting, not following naming conventions, etc. These issues may not prevent the code from working as expected, but they do impact readability. Hence, to identify them, we could use tools such as Checkstyle or Black.
* **Configuration errors:** Some changes to the code could introduce errors in the configuration files. Of course, this would then introduce new errors that could prevent our software from working as expected. To avoid this, we will integrate tools such as Ansible or Puppet in our CI pipeline; these tools can be used to check the configuration files for errors, and in case errors are found, force the pipeline to fail the build.

In all the cases mentioned above, the pipeline will identify the issue and fail the build (which also notified the developer). The developer would then be expected to fix the reported issue, upload the changes to the VCS, and attempt rebuilding to ensure that the issue was corrected.