LO5 Supporting Document

5.1 Identify and apply review criteria to selected parts of the code and identify issues in the code

It is worth noting that in general it is a good idea to perform peer-review code in order for the reviewer to have a bigger picture of the code. From our own personal review, we see that documentation would have to be added, both for the system, and for the tests. We see that variables have meaningful names which makes the program easier to read and understand, along with a properly formatted code.

We see however some improvements can be made, such as trying to have less “magic variables”. We could indeed store the message responses (such as “Bad Request”) in a “constants.js”, so we can reuse them in various places, and after a potential modification, these changes would be made everywhere. There also can be some work done regarding the exception handling in the “/orders” route. Indeed, there is a simple try-catch block, but it would be better to perform more detailed exception handling, which would allow the developers to understand better what is happening (when debugging), and would allow to return appropriate error messages to the user.

In order for another developer to perform review on our code, we can provide them with a review checklist, including criteria such as:

- Having meaningful function/class names and variables.
- Correct folder structure following specified conventions, along with any other coding conventions
- Making sure that the functions that we test for our requirements have unit tests with great code coverage.
- Error messages are returned as expected in order to guide developers and users, along with appropriate status codes.
- Wide variety of inputs are used to perform testing

5.2 Construct an appropriate CI pipeline for the software

Here is an example of an appropriate CI pipeline using GitLab, each of which represents a different step in the development and deployment process. The pipeline would be configured in the `.gitlab-ci.yml` file, and would be triggered on each commit to the repository, ensuring that the application is tested and deployed in a consistent and automated manner.

1. **Build**: In this stage, the pipeline would use GitLab's built-in runner to build the application using the provided `Dockerfile`. This would include tasks such as installing dependencies and transpiling code.
2. **Test**: In this stage, the pipeline would use the same runner to execute tests, using the `jest.config.js` and `TestDockerfile`. This would include both unit and integration tests, as well as any performance tests configured in `performance-test.yml`
3. **Deploy**: In this stage, assuming tests have passed, the pipeline would deploy the application to a staging environment. This could be done using GitLab's built-in Kubernetes integration, or by using a separate tool such as Helm to manage the deployment.
4. **Release**: In this stage, the pipeline would create a new release of the application, tagging the git commit and updating the version in the `package.json`
5. **Monitoring**: Finally, the pipeline would include monitoring and metrics collection, using the tool `Artillery` to check application performance, and sending the results to a monitoring tool such as Grafana.

5.3 Automate some aspects of the testing

Automating aspects of testing is an important part of an efficient CI pipeline. In terms of embedding testing in the pipeline, one way to automate testing is to include it as part of the pipeline itself, so that tests are run automatically whenever code is committed. This can be done by configuring GitLab's built-in runner to execute the tests as part of the pipeline's `test` stage.

Regarding the required level of testing for the CI environment, it is important to have a comprehensive set of tests that cover a wide range of scenarios. Hence we would include both unit and integration tests, as well as performance testing.

Thus, the application is thoroughly tested and any issues are identified and fixed as early as possible in the development process.

5.4 Demonstrate the CI pipeline functions as expected

The CI pipeline could identify various issues through the development and deployment process. For example, build issues, occurring during the build process, such as missing dependencies or errors in the DockerFile, which would be identified by the runner when it attempts to build the application, and would fail the pipeline. There are also test failures during the testing process, such as failing unit or integration tests. Deployment failures can also occur during the deployment process, such as errors in the configuration of the staging environment or issues with the deployment itself. Performance issues such as slow response times or high errors rates, which could be identified by the “Artillery” when it runs the performance tests and sends the results to a monitoring tool.

Build issues could be identified with error. message indicating missing dependencies or invalid code syntax. Test failures can be identified with messages indicating that tests have failed. Deployment failures could be identified with error messages. Performance issues can be identified with error message indicating that the results of the performance tests show high response times or high error rates. Furthermore, the pipeline could be configured to send notifications to the dev team when any of these issues occur.