ENG1 Team Project
Assessment 2

Deliverable 5: Continuous Integration

Team 13 – Team Unlucky

LILLY BROWN
ROSCOE GILLATT
ADAM JOHNSON
YUMIS ZYUTYU
BRANDON WEST
AYMAN ZAHIR

a)
We chose Github as our version control system due to the team's familiarity with the software. Additionally, the existing code base was already hosted in GitHub, so it made sense to continue using the same platform rather than migrating the codebase to a different platform. Due to that, we chose Github Actions as our CI tool, which contributed to a seamless, integrated solution. Other CI software was considered, such as Jenkins; however, the main advantage of GitHub was that it includes both VCM and CI, whilst Jenkins has CI tools purely. We could have integrated it further with VCM tools (like GitHub), but it would have required an additional plugin and thus more time to setup.
Github actions has less scope for customisation via plugins than other software. Given the small scale of the project and our lack of experience with using CI tools, GitHub was the best choice as it covers all the necessities of continuous integration, making configuration and management faster as it required less effort.
By using GitHub actions, we can access everything through a single unified web UI while avoiding the overhead of additional setup.

GitHub actions provide preset workflows that enable continuous deployment of the project, such as checking code quality, code coverage and deployment, which was beneficial since CI tools are relatively new for our team. It also streamlines the release process whenever we create and tag different project versions rather than manually releasing it every time.

To indicate all the steps of the workflow have passed, GitHub Actions' page will have a green tick beside the most recent commit that triggered the workflow and a red cross otherwise indicating the current code base has errors relating to the code or the tests. Pressing on top of each job within the workflow will show what the error is and refactor the code until the workflow passes. This workflow ensures that whenever integration events occur across the repository, the workflow is triggered to build and test each new change and make developers aware of whether the code passes the tests.

Before integrating changes, the workflow checks if the correct code standards are maintained and fails otherwise. Developers will be made aware of whether the changes pass all the workflows before committing the changes, thus leading to cleaner code being pushed every time. For example if the lint code base job fails, developers will be notified and the console output will show where exactly the error is (including file name, line number, error type and measures on how to fix it) so that it can be fixed.
The changes are pushed, and the workflow rechecks the changes made. The workflow automates most of the code review process and developers no longer have to manually go through every line of code and are also informed on the best way to fix the linting errors via superlinter without looking it up.

Our methodology involved creating a custom YAML workflow file within the existing repository, using the preset workflows as reference. That reference listens to push and pull requests across all branches, checks out the latest version of the code, builds the Gradle project and then runs the tests on the project using pre-written GitHub actions. This workflow is automated using GitHub actions and can easily be modified to include automation for checking code quality and coverage. Github actions also allows us to build artefacts such as test reports that can be easily accessed through the Github actions UI as downloadable zip files. Additionally, Github actions also enable us to automate releases of our software.

b)
We used Github Actions as our CI infrastructure as it provides a seamless integration under a unified UI. The CI is configured using YAML files under GitHub/workflows in the repository. Each YAML file is a specific workflow. We currently have 3 workflows:

- **Java CI with GradleFinal** under gradleBuild.yml listens for events such as code integrations (every time any branch is pushed/pulled) and consequently runs the **build** job, **across the latest ubuntu, windows and macOS agents**, and performs multiple steps such as checking out the latest version of the code using the checkout@v2 action, setting up the java environment using setup-java@v2 with personalised arguments such as java version 11 and enabling Gradle caching. The **Build with Gradle** step involves automating building and testing the Gradle file using a command type **Gradle build**, which runs the build task using the Gradle wrapper. **Archive test report** step uploads a test report via upload-artifact@v2 action by gathering the **test report css and html** files generated after testing. This is followed by a secondary job **publish_test_report** which is dependent on the previous job's success, as we don't want to waste resources if the build fails. The job has 2 steps, both fetching and publishing different formats of the test report within the GitHub actions section of the respective commit for visibility and access to test reports after successful commits.

- **Lint Code Base,** under linter.yml, utilises the Super Linter action to automate ensuring code quality is maintained regarding java checkstyles and javadocs. It listens for integration events and consequently runs the build job on the latest ubuntu agent which involves steps such as **Checkout Code** which uses the action checkout@v3 with an argument: **fetch depth = 0** (Full git history is needed to get a proper list of changed files within `super-linter). The next step **Lint Code Base** runs the linter against the code base using super-linter@v4. We set **VALIDATE_ALL_CODEBASE = false** so that it checks out only the latest additions made to the code base rather than checking the entire code base which makes for a more efficient workflow. This was used for our convenience as the linter finds errors outside of our control such as the auto-generated html reports and hence code was still pushed under a failed linter workflow. However, it was convenient to use as we locating errors became simpler via automation.

- **Automate Releases On Tagged Commit**, under releaseTag.yml, listens for tagged commits and consequently runs the build job which checks out the code, builds the Gradle project and automates a release by fetching the latest version tag. This is done via step **Get the version** which gets the version tag under a variable **VERSION** then the jar file is renamed using **TAG** which is defined as the most recent version using the **VERSION** tag. Once the jar is renamed under the correct version the final step **Automatically release,** automates a release using a third party action **softprops/action-gh-release@v0.1.14** and the renamed jar file is released as an asset. The line **fail_on_unmatched_files** ensures the action fails if the correct files aren't matched prior to release as a fail-safe to avoid releasing wrong files

We have also included status badges of the workflows in the readme file informing developers of code quality, latest version and build status. Github Actions also sends automated emails to the developer that integrates the changes, informing them if the workflow passed or failed. The workflow status is also integrated with Github Issues so the status of each commit is visible seamlessly throughout the UI.

Hence, these workflows ensure that cleaner, properly functioning code is pushed or merged every time while also informing developers exactly where any errors are for easy fixing and finally streamlines releases by automating it instead of manually releasing every time.