Quality Assurance Plan

Quality Assurance is crucial in ensuring that the software product we develop meets client's requirements and expectations. This quality assurance plan outlines how the project development process will be maintained. The RTE of each subteam is responsible for monitoring the execution and processes in the agile release train in order to ensure adherence to the quality assurance plan.

Individual development branches can only be merged to the master branch if they successfully passed the automated tests in the CI/CD pipeline to ensure they are properly tested and compatible with the existing codebase. Every feature developed needs to be tested by all team members before it could be deployed and followed by a final testing after deployment. Testing process will not only include internal testing but also user acceptance tests which involve external parties like the client or end users to test the software. User feedback can then be gathered to see if the product meets the desired requirements.

In order for the deliverable feature to be considered complete, a set of conditions or criteria known as the "Definition of Done" (DoD) have to be met. They can be used to check the different levels of completeness of the deliverable product such as user stories, features or epics. A few examples of DoD for the product may include meeting non-functional requirements, passing unit tests and functional tests, clean integration and meeting acceptance criteria of the client. Code artefacts refer to the outputs or deliverables produced during the development process and are important parts of reviewing, testing and documentation. These include the source code of the product, test code, test data, test reports and code reviews which help to identify issues or help with maintaining the code.

The proposed process flow within an agile team starts with backlog refinement, followed by sprint planning, software development, standup meetings, testing, deployment, sprint review and finally a sprint retrospective meeting. This entire process will be repeated in all Sprints.

CI/CD software and tools

CI/CD (Continuous Integration/Continuous Delivery) is a critical aspect of modern software development, enabling teams to automate and streamline the process of building, testing, and deploying software. There are various CI/CD software and tools available in the market, each with its own advantages and disadvantages.

Name of tool	Advantage	Disadvantage
GitHub Actions	• Native integration with GitHub repositories. • Supports a wide range of programming languages and platforms. • Offers generous free usage for open-source projects. • Easy to configure with YAML-based workflows.	• Relatively new, may lack certain advanced features. • Pricing can become expensive for large-scale private repositories.
Azure DevOps	• Integration with other Azure services. • Comprehensive features including version control, CI/CD, and project management. • Good for Microsoft-centric development. • Flexible deployment options including on-premises.	• Can be costly for large teams or enterprises. • Complex setup and configuration compared to simpler tools.
Travis CI	• Easy integration with GitHub repositories. • Provides a free tier for open-source projects. • Supports various programming languages and frameworks. • Fast setup and configuration.	• Limited concurrency and build minutes for free accounts. • Less suitable for complex, large-scale projects.
CircleCI	• Cloud-based and easy to set up. • Supports parallelism and distributed testing. • Good documentation and community support. • Integrates with popular version control systems.	• Pricing can be expensive for large-scale usage. • Limited flexibility compared to self-hosted solutions.
GitLab CI/CD	• Tight integration with GitLab version control. • Built-in CI/CD pipelines as part of GitLab. • Easy to set up and configure. • Supports auto-scaling and parallel execution.	• Limited support for other version control systems. • Continuous deployment features might be basic for complex scenarios.

The chosen CI/CD tool for the project is GitLab CI/CD due to easier integration as the code base will be held in a GitLab repository.

Non Functional Requirements

Maintainability

Maintainability is a generally implied trait in software systems. Hence, to achieve this, the following general but crucial software development practices will be implemented:

Code Documentation: Thoroughly document codebase, including comments, README files, and API documentation to facilitate understanding for future developers.

Modular Architecture: Adopt a modular architecture to compartmentalise functionalities, making it easier to update and maintain individual components without affecting the entire system.

Version Control: Utilise GitLab for version control, ensuring that changes are tracked, revertible, and collaborative, promoting seamless collaboration and rollback if needed.

Testing and Continuous Integration: Implement robust testing practices and set up continuous integration pipelines to catch bugs early and ensure new changes don't break existing functionalities.

Adherence to Best Practices: Enforce coding standards, design patterns, and best practices throughout development to maintain consistency and readability across the codebase.

Regular Refactoring: Schedule periodic code refactoring sessions to optimise performance, improve readability, and eliminate technical debt, keeping the codebase clean and efficient over time.

Portability

NFR6 - Users can use the web app on phones, iPads, and PCs: Enabling flexibility of device to use the web app by creating a portable system that functions on and adapts to different device types

To ensure portability aligns with Puzzle Bluster's vision of engaging puzzle enthusiasts across various devices and platforms, we'll design a web-based application leveraging TypeScript for backend programming and React JS for frontend development. Utilising Bootstrap ensures responsive design, making the app accessible across tablets, PCs, laptops, and smartphones. Firebase as the database management system facilitates seamless synchronisation of user progress across devices, enabling a continuous gaming experience. Additionally, GitLab for repository management ensures version control and collaboration. This approach guarantees that Puzzle Bluster remains easily accessible and enjoyable for users across different devices and browsers, enhancing engagement and reach as envisioned in the team's vision statement.

Security

NFR2 - Secure data storage: Ensuring data integrity and confidentiality so that users' account details are protected against data loss and unauthorised access

Firebase is our chosen database to manage user data. Firebase is built-in with security measures and best practices for database management. There are several notable mechanisms which Firebase offers. Firstly, data encryption in both data storage and data transfer safeguard data from unauthorised third parties. Besides that, it offers real-time monitoring and logging which allow database administrators to identify potential threats by analysing database activities. Moreover, its adherence to various compliance standards and certifications guarantees that Firebase conforms to accepted security and privacy standards in the industry.

Overall usability

We plan to achieve usability through the following features/qualities:

- **NFR1 - Intuitive and Simplistic UI**: Easy to use interface that allows users to navigate between menus and puzzles intuitively
- **NFR3 - The user can log in as a guest without an account:** Providing users an additional option to use the app without having to register, thus simplifying the experience for lightweight users
- **NFR4 - Attractive animation on homepage:** Adding visually appealing animations to attract users and retain their interest by creating a pleasing user experience

Name	Scale	Meter	Target	Constraint
Effectiveness	Completion rate of tasks	**Test:** Users that are new to the system are given tasks (eg: to create a project) to complete in the system, the number of tasks successfully completed can be recorded as a percentage of the total tasks performed.	Target of > 80% completion rate	Completion rate should not fall < 60%
Effectiveness	Number of errors	**Test:** Users are given a set of tasks, the number of errors made are recorded as a percentage.	Error rate < 20%.	Error rate should never fall < 50%
Efficiency	Task completion Time	**Test:** Users are given a specific task (e.g. navigate to a specific page of the app). The mean time spent by the user on completing the task is recorded.	Mean time for completing tasks < 10s	Mean time should not exceed 20s
Efficiency	Response time	**Test:** Time taken for operations	Response time for all user actions < 1s	Response time should not > 5s

Engagingness	Positive feedback from users	**Test:** Collect feedback from a group of users to comment on the user interface and game experience	At Least 70% of the users should give positive feedback	At Least 50% of the users should give positive feedback
Engagingness	Number of visits over time	**Test:** Keep track of the number of users on the platform	The number of users should increase over time	The platform should atleast have 10 active users every 2 weeks
Error Tolerance	Error recovery rate	**Test:** If the test user encounters an error or mistake, the number of users who successfully recover from the error is recorded as a percentage.	Error recovery rate > 70%	Error recovery rate should not fall < 50%
Error Tolerance	Error severity	**Test:** Errors that occur in the system are assessed for their severity and how much impact it has on the user experience. It is rated on a scale of 1-5 (1 = minimal, 5 = severe). The mean of the severity is recorded.	Mean severity rating <= 2	Mean severity rating should not exceed 3.5
Ease of Learning	Time for proficiency	**Test:** The average time taken for users to learn and get used to interacting and performing basic tasks with the software.	Average time taken < 1 min	Average time taken should not exceed 5 min
Ease of Learning	Difficulty of tutorial	**Test:** Users are given a tutorial on how the system works. The tutorial is rated by them on a scale of 1-5 on the difficulty in understanding the system (1 = easy to understand, 5 = difficult to understand). The mean score is recorded.	Mean difficulty rating <= 2	Mean difficulty rating should not exceed 3.5

Scalability

NFR5 - Support 35 online users simultaneously: Allowing a considerable number of online players to participate in the web app simultaneously

In order to make the system scalable and allow considerable concurrent users, we will use Firebase’s dynamic and auto-scaling capabilities. Firebase's real-time database capabilities enable efficient data synchronisation and handling of concurrent user connections. Additionally, Firebase Hosting allows for automatic scaling of server-side logic and dynamic content delivery. We will structure the database schema in a way to minimise the data retrieval time by organising data into logical collections and using appropriate data types. Thus, by optimising database queries and leveraging Firebase's scalable infrastructure, we can reliably accommodate 35 online users simultaneously while maintaining performance and responsiveness.

Accessibility

NFR7 - Special aids for users with disabilities: Incorporating special elements to make the app more inclusive and usable for people with disabilities

To make the design of the application more accessible and user-friendly, various accessibility features are taken into consideration. To make the design more accessible for users with colour blindness, the design may include clear and legible texts or symbols to represent the icons while utilising colours that are distinguishable by colour blind people. Interactive objects may also be clearly highlighted along with clear audio cues based on feedback from their actions for people with slight visual impairments. Clear animations or effects may be added alongside the audio cues to aid people with hearing impairments. The controls will also be simple and smooth, relying mostly on the mouse for controls to accommodate people with motor disabilities. The design will also refrain from implementing flashing images or effects to ensure that it is safe for people with epilepsy.

Git Repository and Branching Model

The Git Workflow Model, chosen by the team, is the GitLab Flow is similar to GitHub Flow but includes additional stages for review and deployment. It features a single main branch (typically master or main) with feature development occurring through feature branches. This was chosen over the Gitflow Workflow and GitHub Flow as it supports rapid iteration and continuous deployment while still allowing for code review and quality assurance, as well as, enabling a balance between agility and control, making it suitable for a wide range of projects, including ours. Additionally, GitLab has a wide range of features, including built-in CI/CD, code review, issue tracking, and more.

Furthermore, in the gitlab workflow, development work for new features, bug fixes or improvements takes place in feature branches which are created from the main branch and are dedicated to specific tasks or features. This ensures that the main branch always remains stable and deployable at all times and once the development work on the specific feature is done and passes all necessary checks, it is merged back into the main branch.

In our workflow, merge requests ensure rigorous code review and testing before integration into critical branches. Designated team members conduct comprehensive reviews to uphold coding standards. Code changes undergo rigorous testing via CI/CD pipelines, automating tests like unit and integration tests. Merge requests are labelled for context, with categories like "bug fix" or "feature enhancement," and may be tied to milestones for project phases. These measures ensure only high-quality, tested code is integrated, minimising the risk of introducing bugs.

Commits

It is crucial to standardise git usage among all team members to ensure consistency and efficiency throughout the entire software development process. First and foremost, commit messages should be well-written and descriptive enough for others to tell at a glance what was implemented in that particular commit. This eliminates the need for other developers to ask about the implementations behind the commit or having to spend time reading through the entire code just to know about what was committed to the branch. To allow easier reference, the user story number needs to be included as part of the commit message for future traceback. Moreover, frequent small commits are needed with each commit serving the code for a single feature instead of committing a big chunk of code incorporating multiple features. This not only provides a clear history of commits, but it also eases the reverting and rollback process that a particular commit can be entirely deleted without affecting other features implemented. The master branch is the stable version ideally without any bugs and hence no one should commit code directly to the master branch to avoid any potential bugs that may cause huge disruption to the existing working software. Lastly, team members should not commit unneeded files to ensure a clean structure of the team repository.

Example of commit message :
US01 - Implemented Saving and Resuming Game Progress

Branch

The team will utilise Feature Branch Workflow, with the core idea being that each feature development should be in its dedicated branch rather than the main branch. The main or master branch should be a locked branch which should not be committed onto and any changes should be done by creating a new branch and adding the new feature or changes to the new branch before sending a merge request. This is to ensure that only reviewed and approved changes are merged into critical branches. Each branch should have limited write access only for those working on the branch while every member is given read access to not disrupt the workflow and ensure members are working on their respective tasks.

The names of the branches should also be descriptive, indicating which sub-team or individual is working on the branch and the feature being implemented through the branch. Tags may also be included in the branch name to indicate their purpose and make them more clear such as “/UIUX”, “/database”, “/feature” etc. Names of the branches should be consistent to ensure that each member can easily identify their purpose at a glance and make it neater to go through. For example, the branch names should all be lowercase, with underscores representing spaces and the slash symbol to separate tags relating to the branch such as “team_1/database/database_configuration”.

Branches should also be created and managed based on the type of feature implemented and the depth of the branch. For shallow branches that have fewer commits and differ slightly from the main branch, they should be deleted after merging to reduce clutter and improve repository performance. For deeper branches that have longer commit histories and the main branch has deviated a lot, rebasing should be utilised to ensure that the feature branches are updated with other features that were added to the main branch. This also allows the team to resolve

any conflicts before merging. Deeper branches should be cleaned after rebasing to reduce complexity and also be deleted after merging.

Merge Requests

In our workflow, merge requests play a crucial role in ensuring that code changes are thoroughly reviewed and tested before being integrated into critical branches. Only designated team members with sufficient expertise in the respective areas will be responsible for approving merge requests. These members will conduct thorough code reviews to ensure adherence to coding standards, best practices, and overall quality.

Before merging, all code changes will undergo rigorous testing using continuous integration and continuous deployment (CI/CD) pipelines. These pipelines will automate various tests, including unit tests, integration tests, and possibly end-to-end tests, to validate the functionality and stability of the codebase across different environments.

To streamline the process and provide clear context, merge requests will be labelled or tagged appropriately. Labels may include categories such as "bug fix," "feature enhancement," or "documentation," allowing team members to quickly identify the nature of the changes. Additionally, milestones may be assigned to merge requests to indicate their significance or tie them to specific project phases or deadlines.

By following these guidelines, we ensure that only high-quality, thoroughly tested code is integrated into critical branches, minimising the risk of introducing bugs or regressions into the main codebase.

Rules to Follow

Summarise the description of rules above in the table below. You can use this as the reference to raise concerning problems.

Code	Subject	Description	If violated
M1.1	Commits	Descriptive commit messages for readability	RTE should advise team member to provide clearer context and understanding in the commit messages
M1.2	Commits	Frequent small commits	RTE should advise team member to commit code more frequently
M1.3	Commits	User story number should be included as part of the commit message	RTE should ask team member to include user story number in commit messages in the future commits or edit commit message if possible
M1.4	Commits	Not needed files should not be committed	RTE should advise team member to remove the files from the commit before a final merge to the master
M2.1	Branch	New branch should be created for each feature	RTE should advise team member to migrate their commits to a new branch

M2.2	Branch	Branch should only be written and edited by subteam responsible	RTE should advise team members to work on their own branch.
M2.3	Branch	Descriptive branch names for readability	RTE should ask team member to rename the branch to be clearer so that other members can easily differentiate each branch from one another
M2.4	Branch	Rebase deep branches with main to catch up with changes	RTE should advise team members to rebase the main branch into their own branch to resolve conflicts and check the compatibility of their code.
M3.1	Merge Requests	Delete the source branch after merging.	RTE contacts that team member to delete the branch.
M3.2	Merge Requests	All merge requests should undergo code review by at least one other team member	Merge Request should be denied,RTE should advise team members responsible for the merge request to have his/her code reviewed by another team member.
M3.3	Merge Requests	Clear Description of the changes included in the merge request.	RTE should advise team member to provide clearer context and understanding in the merge request
M3.4	Merge Requests	Single Responsibility merge request	RTE should remind all team members that merge requests should ideally address a single issue or feature at a given time

M3.5	Merge Requests	Codes testing before merge	Merge Request is denied,RTE should advise team member to do some sort of testing of his branch to ensure there are no errors and no other functionality changes/breaks.
M3.6	Merge Requests	Merge Conflict Resolution	RTE should work with team member in resolving the issue or restoring the branch to its working version.