Requirements Document
Milestone 3

Testing plan:

Currently, the testing approach our team is using involves writing the code first and then testing to see if it works. That is the "manual testing" or "ad-hoc testing," where tests are performed informally without a predefined test plan or test cases. While this approach can be effective for quickly validating functionality, it may lead to oversight of edge cases and make it challenging to ensure comprehensive test coverage.

In the future, our team plans to adopt a "Test-Driven Development" (TDD) approach. Test-Driven Development is a software development methodology where tests are written before the code implementation. The TDD process typically follows these steps:

Write Test Cases: Developers write test cases based on the expected behavior of the code. These test cases serve as specifications for the code's functionality.

Run Failing Tests: Initially, the tests written in the previous step will fail because the corresponding code does not exist yet.

Write Code: Developers write code to fulfill the requirements specified by the test cases.

Run Tests: After writing the code, developers run the test suite. The goal is for all tests to pass, indicating that the code implementation meets the specified requirements.

Refactor Code: Once the tests pass, developers may refactor the code to improve its design, readability, or performance while ensuring that all tests continue to pass.

By adopting Test-Driven Development, our team aims to improve code quality, reduce the number of defects, and promote better design practices. TDD encourages developers to think critically about the desired behavior of the code upfront, leading to more robust and maintainable software in the long run. Additionally, having a comprehensive suite of automated tests ensures that future changes or refactoring efforts can be done with confidence, knowing that existing functionality remains intact.

Design Patterns:

1. **Model-View-Controller (MVC) Pattern:**

The MVC pattern is a widely used architectural design pattern that separates an application into three interconnected components: Model, View, and Controller. In the context of our online ticketing app:

- Model: Represents the data and business logic of the application. In the case of the ticketing app, this could include user information, event details, and transaction data.

- View: Displays the user interface and interacts with the users. In the ticketing app, the view would be the web page where users browse events, select seats, make purchases, and sell their unwanted tickets. As well as where event organizers create and manage their event information and tickets.

- Controller: Acts as an intermediary between the Model and View. It handles user input, processes requests, and updates the Model accordingly. For the ticketing app, the controller would manage actions like creating an event, processing a ticket purchase, and updating the user's transaction history

When implemented, the backend of the ticketing app can be implemented using a server-side framework that follows the MVC pattern, such as Django (hence why we chose it in our techstack)

2. **Mediator pattern:**

The mediator design pattern as its name suggests specializes in managing complex interactions between multiple independent components similar to the role of a control tower in an airport. In the case of our ticketing app, using the mediator pattern will allow us to manage the communication between different components without them being explicitly aware of each other. For example, the booking system, payment gateway, and user authentication module can communicate through a central mediator, reducing direct dependencies. Additionally the mediator pattern should simplify the addition of new components or functionalities without requiring changes to existing components.