Requirements Document
Milestone 3

Current Testing Plan

Overview:
The current testing plan appears to follow a sequential development approach where:

- Development First: Code is written for the event ticketing website's functionalities without preliminary testing considerations or test-driven development practices.
- Post-Development Testing: Once a significant portion of the code or a complete functionality is developed, testing is conducted to identify and fix bugs.
- Limited Early Feedback: This approach might limit early feedback that could prevent bugs or design flaws, potentially leading to more significant changes late in the development cycle if issues are discovered.
- Testing Scope: Likely focused on functional testing, with performance, usability, and compatibility testing following the development.

Future Approach: Integrated Testing Strategy

To enhance efficiency, reduce bugs, and improve overall product quality from the early stages of development, consider an integrated testing strategy. This strategy involves continuous integration, continuous testing, and Agile methodologies.

Agile Methodology: Adopt an Agile development process, where development and testing are concurrent and iterative. This approach allows for continuous feedback and adjustments throughout the development cycle.

- Sprints: Break down the development into short sprints, allowing for specific features to be developed, tested, and reviewed in cycles.
- User Stories and Acceptance Criteria: Define user stories with clear acceptance criteria, guiding both development and testing efforts.

Test-Driven Development (TDD): Implement TDD where tests are written before the code. Writing tests first ensures that the code meets the required functionality from the outset.

- Unit Testing: Focus on writing tests for small units of code, ensuring each part functions correctly before integrating.
- Integration Testing: After unit tests, ensure that different parts of the application work together as expected.

Continuous Integration (CI) and Continuous Deployment (CD): Automate the integration of code changes from multiple contributors into a single software project.

- Automated Testing: Incorporate automated testing in the CI/CD pipeline to run tests automatically every time a new code commit is made, ensuring immediate feedback on the impact of changes.
- Regular Deployment: Use CD to automatically deploy changes after they pass tests, allowing for regular feedback from end-users and stakeholders.

Enhanced Testing Scope: Broaden the testing scope early in the development cycle.

- Performance, Usability, and Compatibility Testing: Integrate these testing types earlier in the development process to identify and address potential issues sooner.

Collaboration and Communication: Foster a culture of open communication and collaboration between developers, testers, and stakeholders to ensure alignment and address issues promptly.

Transition Plan

To shift from the current to the future approach:

- Training and Mindset Change: Train the team on Agile practices, TDD, and the importance of continuous integration and testing.
- Tooling: Invest in tools and platforms that support automated testing, CI/CD, and agile project management.
- Iterative Implementation: Start small with one project or team and gradually expand the approach as the organization adapts to the new methodology.

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.