Test Design

Testing Techniques Used for the E Jam Project

Introduction

The purpose of this document is to describe the test design of the E Jam project. This includes the testing strategy, the testing environment, the testing tools, the testing process, and the testing schedule.

Scope

We had multiple constraints during the testing process. Therefore, we focused on testing the core functionalities of the project. The Main scope of this test design is covering the testing strategies such as, the API test, UI test, and performance test of the system, including others that were both essential and critical.

Constraints

Although we faced multiple constraints during the testing process, this document highlights how we overcame these constraints while working on the project.

We had limited resources, which meant that we had to prioritize our testing efforts. Additionally, we encountered time constraints and had to ensure that we completed testing within the given timeframe.

For the Time constraints, we had a limited amount of time allocated for testing, so we focused on testing the most critical functionalities first. To optimize our testing time, we used automation testing for API testing, which saved us a significant amount of time.

Availability of resources: We faced challenges in regard to the availability of devices and tools for testing. However, We used tools such as Docker and virtual machines to simulate different environments and devices, which allowed us to test a variety of scenarios.

Technical constraints: We faced technical challenges in integrating some tools and frameworks, but we were able to overcome them by seeking help from the community and consulting the documentation, and of course our mentors who helped us a lot. Also reporting bugs on a dependency in the front end GUI, and downgrading until we found a stable version that could work, until the dependency was stable.

We also utilized (CI/CD), just for building the front end on cross platforms, to ensure that it worked properly on different devices and operating systems, and manually testing some of

them that were available.

For Risks and prioritization, we had multiple decisions that would help the system significantly, but they were extremely risky.

The first one is that we planned to have the admin client as a single component to ease the development process, but it turned out to be too risky, so we decided to split it into multiple components.

Another risk we faced was changing the front end framework from electron (A JS framework) to a totally different language and framework, Flutter.

Flutter helped significantly to improve the performance of the UI, but it was a risky decision as it required us to learn a new language and framework, and scrape everything that was done before in Javascript.

However, it was worth the risk as the end result was a much better user experience. As, it offers a wide range of testing tools and widgets designed for UI.

For UI testing, we used Flutter's built-in testing framework, which allowed us to easily write and execute UI tests on cross-platform applications, and then switched to manual testing since the test cases were not strong enough to cover all scenarios.

For API testing, we used postman and Thunder Client to validate the request and response of the API.

In performance testing, we used built in tools such as Flutter’s performance profiler, and cargo bench for rust, and monitoring the output statics for the Low Network Level, and as well monitoring the response time on each request made. Which all seemed to work wonders for helping improve the system’s performance. We also utilized load testing, which helped us identify areas for improvement in the project.

Test Plan

Throughout the testing process, we maintained regular communication, and standards that follow throughout the system’s flow. Our objective was to ensure that the project met the specified requirements and was of high quality, providing a seamless user experience.

The following are example components for standards followed through the system. We will not explain each parameter and how it works, since you will find all of them in the center point’s documentation page.

Stream Entry:

```
{
"streamId": "000",
"name": "Test Stream",
```

```
"description": "this is stream 000, for testing",
"lastUpdated": 0,
"startTime": null,
"endTime": null,
"delay": 0,
"generatorsIds": ["AA:AA:AA:AA:AA:AA"],
"verifiersIds": ["AA:AA:AA:AA:AA:AA"],
"payloadType": 0,
"burstLength": 0,
"burstDelay": 0,
"numberOfPackets": 100,
"payloadLength": 1499,
"seed": 540024,
"broadcastFrames": 12,
"interFrameGap": 123,
"timeToLive": 1000,
"transportLayerProtocol": "TCP",
"flowType": "BackToBack",
"checkContent": false
}
```


Device:


```
{
"name": "test device 1",
"description": "this is a test device",
"location": "home",
"lastUpdated": 0,
"ipAddress": "1.1.1.1",
"port": 8084,
"macAddress": "AA:AA:AA:AA:AA:AA"
}
```


System API Stream Details:

This is used for serializing the Stream Entry JSON that the system API will execute, after validation.


```
{
"stream_id": "000",
"delay": 100,
"generators": ["AA:AA:AA:AA:AA:AA"],
"verifiers": ["AA:AA:AA:AA:AA:AA"],
"payload_type": 1,
"number_of_packets": 144,
"payload_length": 123,
"seed": 545,
"broadcast_frames": 1200,
"inter_frame_gap": 23,
```

```
"time_to_live": 11,
"transport_layer_protocol": 0,
"flow_type": 1,
"check_content": 1
}
```


Approach

Our goal was to check that each component of the system meets the specified requirements, and functions correctly. We followed a systematic approach to testing, including unit testing, integration testing, system testing, and acceptance testing, performance testing, and other depending on the specific components.

For unit testing, we tested each component individually, verifying that it worked correctly in isolation, which was done to all components of the system.

For integration testing, split the system into two components that both were heavily tested during the integration testing. the first is the Admin Client Components that contains the frontend and center point, and the other is the Low Level Components which contain the system API for communication, and the components responsible for sending and receiving data.

For system testing, we tested the system as a whole, including all components and interfaces, starting from the network layer all the way to the front end’s charts.

For acceptance testing, we verified that the system met the specified requirements, both functional and non-functional, and that it was ready for deployment. We also tested for user scenarios and ensured that the system was user-friendly, and followed the design guidelines of the project.

For performance testing, we verified that the system was able to handle large amounts of data, and that it met the required performance standards.

We also tested for security on the system's API.

Finally, we followed a continuous testing approach, where we tested the system at every stage of development, to ensure that any issues were caught early and resolved quickly.

We used a combination of manual and automated testing throughout the process, leveraging automation tools such as postman, Thunder Client, and Flutter's built-in testing framework. And fakers for generating fake data for testing, and some additional tools for performance testing. We also performed regression testing after any changes to ensure that existing functionality was not affected.

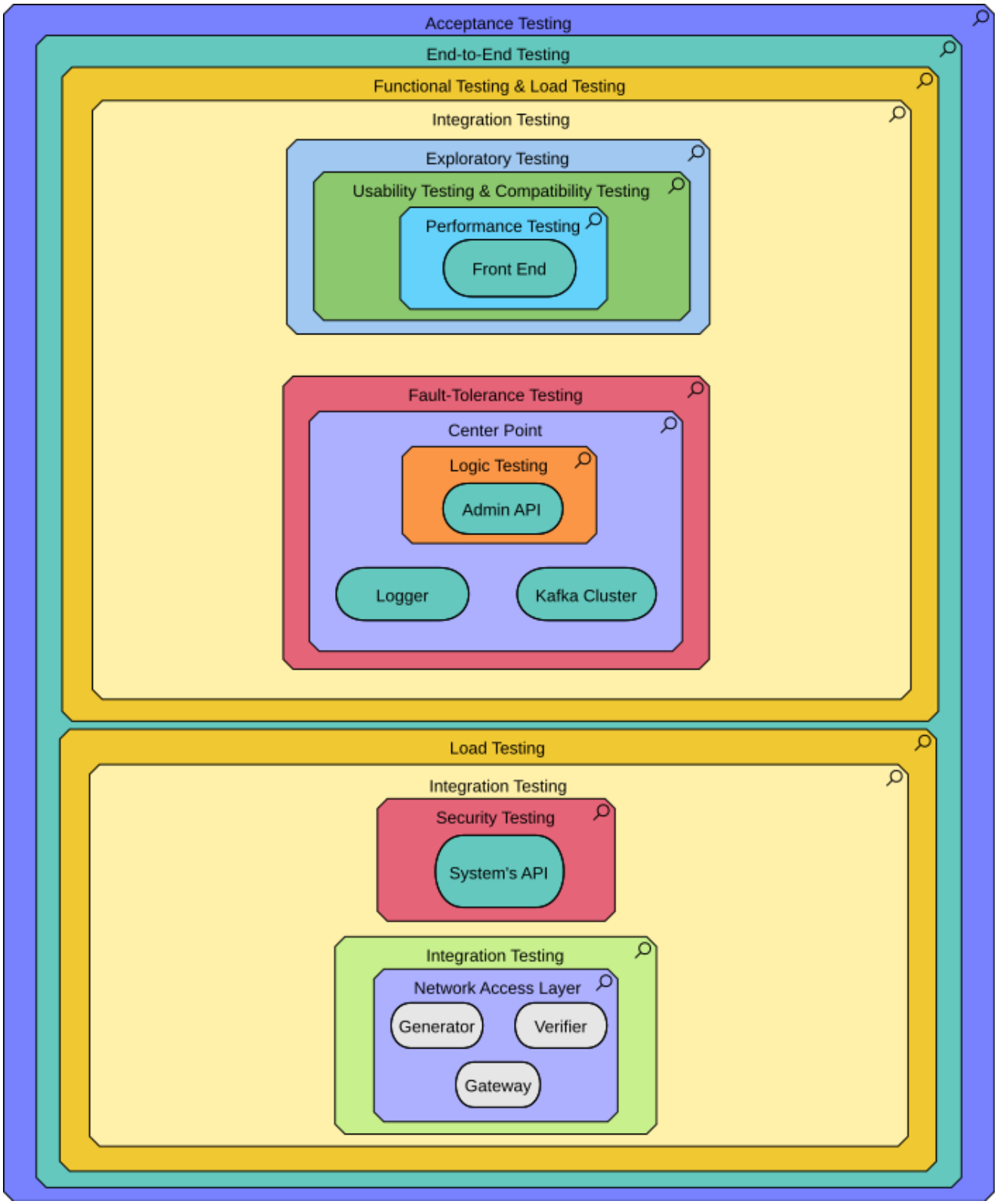
In terms of risk prioritization, we focused on testing the most critical functionalities and used automation testing where possible to save time.

Test Coverage

To simplify things, we created test cases for each component, and organized them into categories that were easy to manage. These categories were based on the type of testing, such as unit, integration, system, acceptance, performance, or security, etc.

The following is a graph specifying the types of testing done on each component relative to other components in the same container.

Each component in the hierarchy starts from the top for each component’s unit testing, then all the way down to the System or End-to-End testing.



Notice that the Network Access Layer Component is treated as one component, since there is always and expected Verifier when generating data going through the switch.

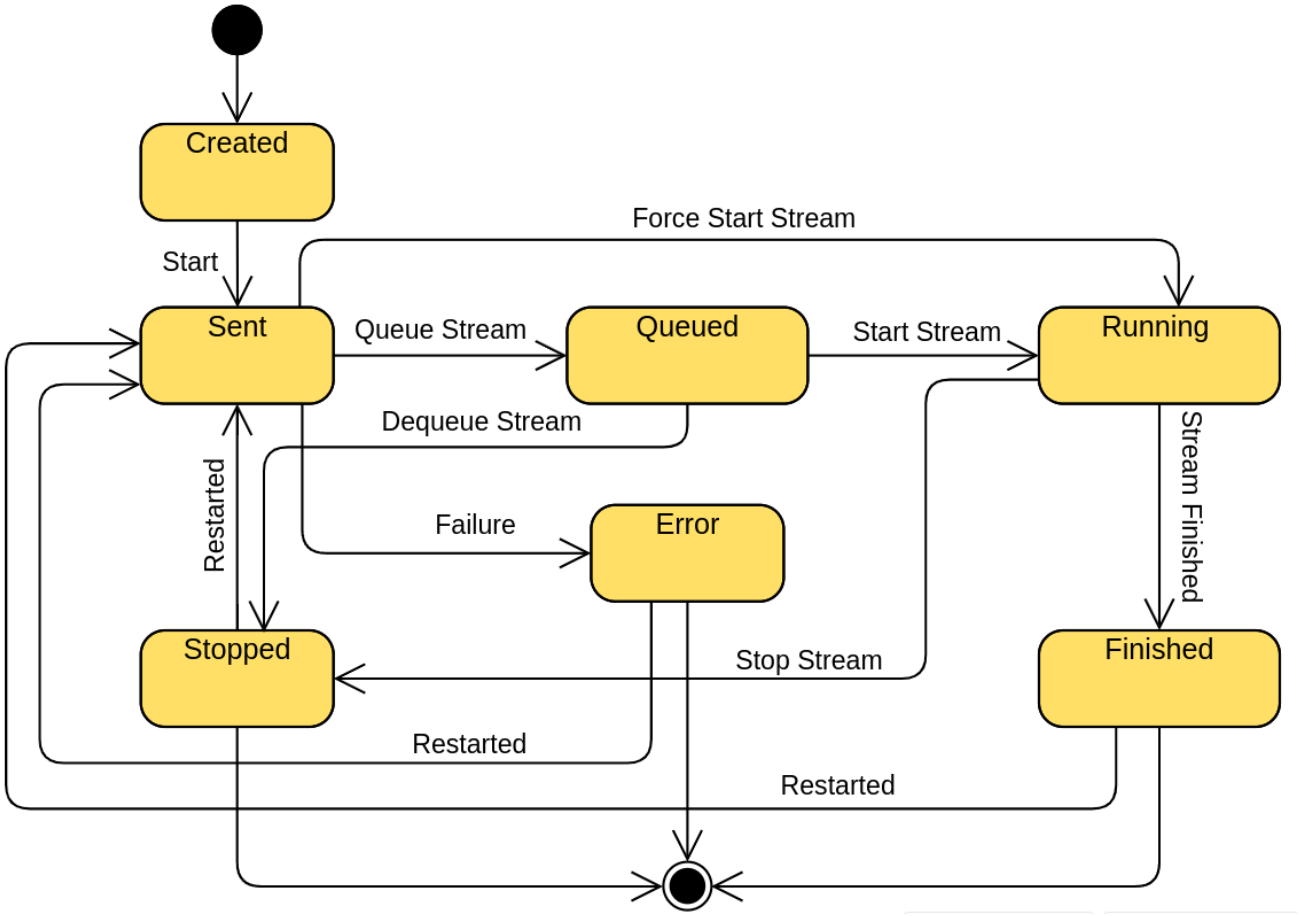
A lot of components had different types of testing strategies, but are not mentioned in the graphs since they were specific to each component and were documented separately. However, the main idea was to have full test coverage and make sure that every component of the system is tested thoroughly to meet the project's requirements.

Each component in the system needed coverage for all the states that it can be in, including normal, boundary, and error cases. We also made sure to test any dependencies that the

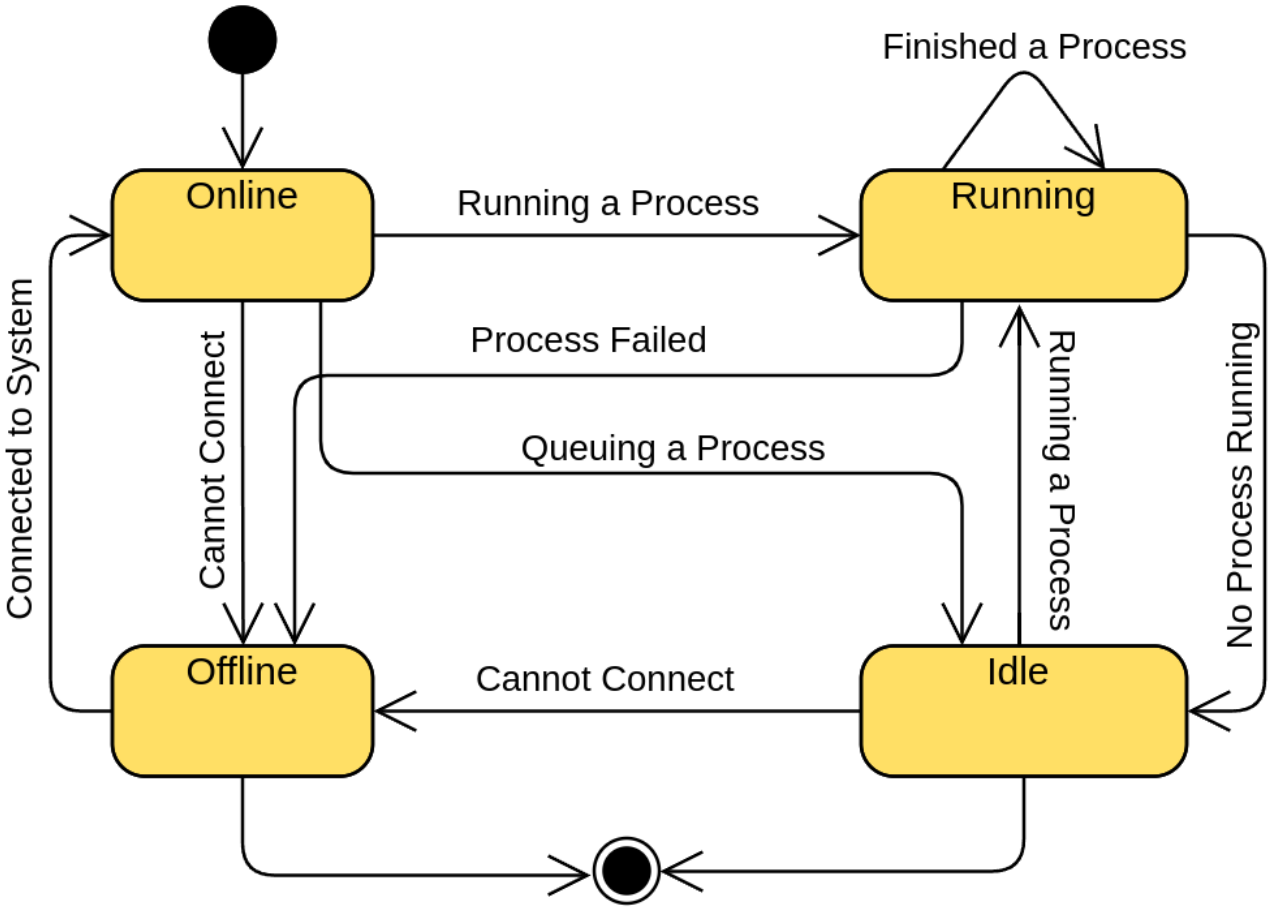
component had, and to cover any edge cases that could impact the system's stability.

The following are the State Machines for the main components of the system.

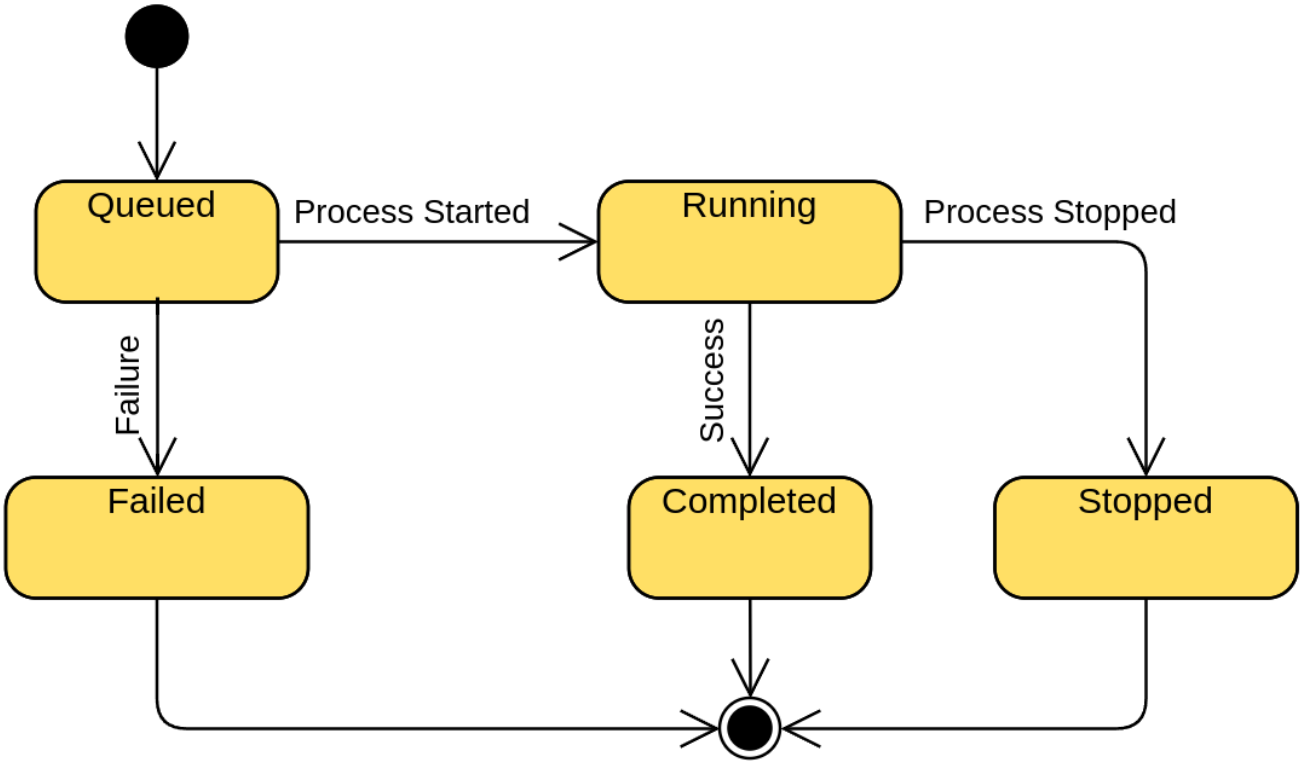
For all streams That exist in the system:



For devices:



For processes running on any device:



Conclusion

In conclusion, we have followed a comprehensive and systematic approach to testing all components and functionalities of the system. By conducting unit testing, integration testing,

system testing, acceptance testing, performance testing, and security testing, we have ensured that the system meets the specified requirements and is ready for deployment.

Using a combination of manual and automated testing tools, we have achieved full test coverage and maintained a level of quality throughout the development process. Continuous testing and regression testing have been critical in catching any issues early and resolving them quickly.

Through risk prioritization, we focused on testing the most critical functionalities and leveraged automation testing to save time and resources. The thorough testing of each component and their dependencies, as well as covering normal, boundary, and error cases.

Also, no matter how much we test a system, there is always a chance that there may be software bugs that are not caught during testing. Deploying the application and collecting feedback from users is an important step in identifying and resolving software bugs. Users may encounter issues that were not anticipated during testing and may provide valuable feedback that can help improve the application.

References

- Flutter Performance profiling
- Exploratory testing
- Regression testing
- logic-based testing