Escape Doom - Software Testing Assignment 2

Kambal, Nowak, Perov, Selbach, Winter

June 7, 2025

Contents

1 Quality Requirements

Quality Overview Table

Functionality	**Requirements:** • Instructor login and room management • Student interaction with rooms • Code editor syntax highlighting • Real-time leaderboard updates **Scenarios:** • Instructors log in securely to manage rooms • Leaderboards update after solving riddles
Performance	**Requirements:** • Handle 200 concurrent users • Code execution within 10 seconds **Scenarios:** • High concurrency in a single room • Fast result feedback for code
Usability	**Requirements:** • Join room in under one minute • Visible hints on all screen sizes **Scenarios:** • Fast room joining • Hints visible on various devices • Keyboard navigation for all actions
Reliability	**Requirements:** • Persistent progress after browser cache is cleared • Handle malicious code without crashing • Keep core features during partial outage **Scenarios:** • Progress restoration after cache clear • System stability on bad input • Maintain basic features during failure
Security	**Requirements:** • Prevent unauthorized access • Sandbox for code execution • Encrypted communication **Scenarios:** • Block access on login failure • Run code in isolation • Use HTTPS for all communication
Data Integrity	**Requirements:** • Secure storage of results • Backup and recovery **Scenarios:** • Restore leaderboard after crash • Recover from backup in 10 minutes
Accessibility	**Requirements:** • WCAG 2.1 compliance • Support for color blindness **Scenarios:** • —

2 Relevant Product Risks

This chapter identifies the primary product-level risks associated with the Escape Doom system, based on its context, architectural strategy, and intended use. These risks are relevant to ensuring the platform delivers a reliable and maintainable experience for both students and lecturers of FH Campus Wien.

Context-Based Risk Identification

The Escape Doom system enables lecturers to manage escape rooms and students to participate in them through a shared browser-based interface. Key software and systems involved include:

- **Frontend (Next.js):** Unified interface for students and lectors.
- **Backend (Spring Boot Microservices):** Business logic separated by domain (e.g., game management, user sessions).
- **Code Executor Service:** Executes user-submitted code (externalized via API).
- **Image Hosting Service:** Provides visual assets used in room puzzles.
- **Authentication Provider:** Handles identity and access control.

These components interact across runtime boundaries, which introduces integration and reliability challenges that must be mitigated by effective testing.

Risks Derived from the Solution Strategy

Risk Category	Description
Deployment Configuration Errors	Kubernetes-based deployment introduces risk in service discovery, routing, and persistence, especially when deploying updates to stateless services.
Data Loss or Corruption	Mismanagement of externalized state (e.g., Redis, PostgreSQL) may lead to lost game progress or broken leaderboard data.
Security Misconfigurations	Improper gateway configuration or missing authentication checks could expose APIs or user data, especially since third-party services like Code Executor are integrated.
Frontend Failure or Desync	Since a single Next.js application serves both user groups, any frontend bug can affect all flows. Moreover, inconsistency between client state and backend responses may lead to corrupted game flows.
Performance Degradation Under Load	Scalability is a stated goal, but it comes with the risk that system components (especially custom services) may not scale equally, leading to degraded UX.
Code Execution Instability	As the Code Executor is an externalized subsystem, failures or incorrect behavior in this service can block entire escape room flows.
Inter-Service Communication Failures	The system relies heavily on service-to-service calls (e.g., Spring Cloud Gateway). Misrouting, timeouts, or schema mismatches can disrupt game logic.

Test-Relevant Risk Dimensions

Each risk will be further analyzed along three key dimensions during test design:

- **Likelihood:** How likely is this risk to occur given current implementation and system context?
- **Impact:** What would be the consequence to user experience or data integrity if the risk occurred?
- **Detectability:** Can this issue be easily discovered during development or testing?

These will help prioritize test effort and define test depth and scope for each risk area.

Implications for System Quality

The product risks identified above point to areas where the system's quality could be compromised if not properly addressed during testing. Several architectural decisions such as splitting the backend into microservices, using external services for code execution, and relying on runtime communication—introduce specific challenges that could affect both user experience and system robustness.

For example, inter-service dependencies may lead to cascading failures, which makes resilience and fault tolerance essential. The integration of external components, like the Code Executor, creates additional attack surfaces and reliability concerns that must be mitigated through isolation and fallback mechanisms.

Ensuring smooth frontend-backend interaction is key to maintaining a consistent game flow, while performance bottlenecks during peak usage could directly impact the responsiveness expected by students and instructors.

These risks underline the importance of focusing testing efforts on aspects such as maintainability, availability, data integrity, security, and overall system responsiveness. Addressing these proactively in the test design will support the delivery of a stable, usable, and secure application.

Testing will therefore be closely aligned with these quality characteristics to reduce the likelihood and impact of such risks.

3 Quality Requirements

Quality Overview Table

Category	Requirement / Sub-Goal	Scenario
Functionality	• Instructor login and room management • Student interaction with rooms • Code editor syntax highlighting • Real-time leaderboard updates	• Scenario 1: Instructors log in securely to manage rooms • Scenario 2: Leaderboards update after solving riddles
Performance	• Handle 200 concurrent users • Code execution within 10 seconds	• Scenario 1: High concurrency in a single room • Scenario 2: Fast result feedback for code
Usability	• Join room in under one minute • Visible hints on all screen sizes	• Scenario 1: Fast room joining • Scenario 2: Hints visible on various devices • Scenario 3: Keyboard navigation for all actions
Reliability	• Persistent progress after browser cache is cleared • Handle malicious code without crashing • Keep core features during partial outage	• Scenario 1: Progress restoration after cache clear • Scenario 2: System stability on bad input • Scenario 3: Maintain basic features during failure
Security	• Prevent unauthorized access • Sandbox for code execution • Encrypted communication	• Scenario 1: Block access on login failure • Scenario 2: Run code in isolation • Scenario 3: Use HTTPS for all communication
Data Integrity	• Secure storage of results • Backup and recovery	• Scenario 1: Restore leaderboard after crash • Scenario 2: Recover from backup in 10 minutes
Accessibility	• WCAG 2.1 compliance • Support for color blindness	• —

4 Derived Test Objectives

Based on the identified product risks and defined quality requirements, the following test objectives have been established to guide test design and prioritization:

- **Verify system stability across microservices**, ensuring failures in one service do not propagate and that fallback or error handling is in place.
- **Test secure user authentication and role management**, validating that only authorized users can access protected features and user data is handled securely.
- **Validate persistent and correct game state management**, including user progress, session data, and leaderboard synchronization, even in failure scenarios.
- **Confirm code execution behavior is safe and reliable**, particularly in handling edge cases, incorrect input, or malicious submissions.
- **Assess frontend usability and responsiveness**, especially in terms of navigation, accessibility, and performance across devices and screen sizes.
- **Check system performance under expected concurrency**, ensuring smooth operation with up to 200 users per room and acceptable response times.
- **Ensure recoverability and data integrity**, with proper test coverage for backup, restore, and resilience mechanisms in the case of data loss or outages.
- **Verify CRUD operations in all repositories**, ensuring that each microservice managing a domain entity has complete test coverage for Create, Read, Update, and Delete functionality. This applies to all persistent components within the system.

These objectives reflect the test priorities and help define the focus areas for unit and system-level testing. Each objective will be mapped to specific test cases in later test design phases.

5 Unit Test Approach

General Approach

Unit tests are written following a Test-Driven Development (TDD) strategy where applicable, particularly during the early stages of service implementation. Tests are focused on public methods. Private methods are not tested directly, as their logic is implicitly validated through the public interface.

Each microservice follows a consistent folder structure to support maintainable and repeatable testing workflows. When upper layers (e.g., services or controllers) are introduced, previously granular tests may be reduced to improve build speed while still ensuring coverage through layered validation.

Test Structure by Domain Service

Data Service As a core service, the data service maintains the highest test coverage:

- **Data access layer:** 62% of classes, 74% of lines covered
- **Mapping layer:** 90% of classes, 53% of lines covered
- **Service layer:** 100% of classes, 83% of lines covered

Testing began with TDD to ensure all repositories implement and verify CRUD operations. Service-layer tests validate integration with data handlers and repositories. Flyway is also integrated to provide a stable schema for testing.

Leaderboard Service This service is not yet fully implemented. Testing is planned as follows:

- **Unit tests** will focus on transformation logic between session data and leaderboard models.
- **Edge cases** such as empty result sets and invalid session responses will be covered.
- **Risk-based focus:** Inaccurate or missing leaderboard data could break user trust, so correctness and reliability are critical.

Player Service Current coverage status:

- **Data access layer:** 33% of classes, 18% of lines
- **Service layer:** 84% of classes, 18% of lines

Further testing is required to reach at least 60% line coverage. Focus areas include data integrity, score calculation logic, and boundary conditions for user progress tracking.

Session Service Already has strong test coverage:

- **Data access layer:** 100% of classes, 80% of lines
- **Service layer:** 100% of classes, 36% of lines

Remaining work includes covering more complex logic branches in the service layer, particularly scenarios with invalid session states or timing constraints.

Gateway Service As this component mainly handles routing and security, its logic will be validated through frontend end-to-end and system integration tests rather than unit tests.

Test Tools

- `JUnit 5` for all Java service-level tests
- `Mockito` for mocking dependencies in isolation
- `Flyway` for consistent schema migration and rollback in test environments

6 System Test Approach

System testing plays a critical role in validating that all components of the Escape Doom platform work together as expected from an end-user perspective. While unit and integration tests focus on isolated components and internal logic, system tests verify real-world behavior across services, interfaces, and layers.

Test Basis

System tests are based on:

- Functional specifications
- Use cases for both student and lecturer roles
- Previously identified product risks and quality goals

Test Objects

- Complete microservice-based application stack (frontend, backend, gateway)
- End-to-end flows such as:
  - Joining and playing an escape room
  - Submitting and evaluating code
  - Viewing and updating the leaderboard
  - Session management by instructors

System Testing Strategy

The system tests are primarily conducted through **end-to-end (E2E) automation**, simulating realistic user behavior across the application stack. These tests aim to ensure that key user-facing functionality remains intact during development cycles.

- **E2E Tests with Playwright and Selenium:** Used to simulate browser-based interaction scenarios involving students and lecturers. This ensures that critical user workflows (e.g., login, code submission, viewing leaderboards) work across services and through the UI.
- **MockMVC Tests:** Used within backend services to simulate HTTP requests at the controller level. This allows us to test edge cases, error handling, and validation logic without requiring a full UI-driven scenario.

Test Tools

The team has agreed on the following tools to support system and controller-level testing:

- **Playwright / Selenium** – For simulating real user actions and full-stack testing.
- **MockMVC** – For controller tests, especially useful where UI coverage is not sufficient.
- **TestContainers** – To create realistic, isolated test environments with actual database instances.
- **AssertJ** – For fluent and human-readable assertions in all Java-based tests.

Why System Tests Are Important

Given our microservice architecture and external dependencies (e.g., Code Executor, Authentication Provider), integration and contract errors may not be caught at the unit level. System tests help verify:

- Cross-service communication and data flow
- User-visible behavior consistency
- Fault tolerance and error handling under real conditions
- Confidence in critical flows such as progress tracking and leaderboard updates

7 Test Infrastructure

This chapter describes the infrastructure required to execute automated and manual tests across the Escape Doom platform. A consistent and reliable test environment is critical for maintaining test quality and enabling reproducible results across services.

Test Environment

The system is built on a containerized microservice architecture. Therefore, all test environments replicate production-like conditions using:

- **Docker Compose / Kubernetes (Minikube):** Local environments replicate service orchestration, routing, and scaling behavior.
- **TestContainers:** Used in integration and system tests to spin up real dependencies like PostgreSQL and Redis on-demand.
- **Flyway:** Ensures consistent database migrations across test environments for schema alignment.

Test Data

- **Static test data:** Predefined users, sessions, and escape rooms for repeatable scenarios.
- **Dynamic test data:** Generated on the fly using factories in tests for coverage of edge cases.
- **Schema migration tools:** Flyway migrations are applied automatically to ensure a valid schema state before tests execute.

Tooling Overview

The following tools are used across all testing levels:

- **JUnit 5** – Test runner for unit and integration tests
- **AssertJ** – Fluent assertion library for readable test output
- **Mockito** – For mocking dependencies during isolated unit testing
- **MockMVC** – For controller-level testing without a real HTTP server
- **TestContainers** – For running real database containers during integration tests
- **Playwright / Selenium** – For system and end-to-end tests via browser simulation
- **Flyway** – For applying database migrations in test environments

CI Integration

- Tests are executed via a CI pipeline on each merge request to ensure no regressions.
- TestContainers and Flyway enable clean and isolated database states for every CI run.
- Test reports are generated and archived to identify failures early.

Test Stability and Repeatability

All tests are designed to be environment-independent. No reliance on shared mutable state or external live systems ensures that results are deterministic and failures are traceable to code changes, not environment noise.