CRANCSTORM
Software Testing Plan
March 20, 2026

Team Members:

Braydon Lamoureux

Kristopher Tomas

Noelia Canela

Ethan Ferguson

Project Sponsor: Dr. Steven Gerhke
Faculty Mentor: Dr. Ana Paula Chavez
Team Mentor: Bailey Hall

Table of Contents

1. Introduction

The CRANC application is a containerized web-based system designed to manage user data and interactions through a frontend UI, backend services, and a PostgreSQL database. The system supports core functionalities such as user creation, authentication, and data retrieval, which are derived directly from the project backlog.

From an architectural perspective, the system consists of three primary components:

- Frontend (UI layer)
- Backend API (business logic and data processing)
- Database (PostgreSQL)

Additionally, the system depends on:

- Containerization (Podman/Docker)
- Networking between services
- Authentication and user data handling

Scope of Testing

In Scope:

- Backend API logic and validation
- Database operations and persistence
- Frontend–backend communication
- Core user workflows (CRUD operations, authentication)
- Container deployment and service interaction\

These directly map to backlog features such as user registration, login/authentication, and data retrieval endpoints.

Out of Scope:

- Third-party libraries (assumed tested externally)
- Low-level UI styling (non-functional aesthetics)
- Platform-specific optimizations beyond development environment

This scope aligns directly with the project backlog, focusing on implemented features such as user data management, API endpoints, and containerized deployment.

Testing Strategy Overview

The CRANC project will implement the following testing types:

- **Unit Testing:** Validate individual backend functions and logic
- **Integration Testing:** Verify interaction between frontend, backend, and database
- **Usability Testing:** Evaluate user interaction and workflow efficiency
- **(Optional but included) Performance Testing:** Basic response time validation
- **(Optional but included) Security Testing:** Input validation and authentication checks

Testing Timeline:

- Unit tests: Continuous (every pull request / CI)
- Integration tests: After major feature completion and before demos
- Usability testing: After Alpha I and before Alpha II / Acceptance

- Performance/security checks: Before final demo

Rationale

The backend and database layers require the most rigorous testing due to their complexity and high risk of failure impacting all users. Integration testing is critical because the system relies heavily on containerized service communication. Usability testing is emphasized to ensure the system meets user expectations despite being technically complex.

The following sections describe each testing strategy in detail.

---

2. Unit Testing Plan

Unit testing focuses on verifying individual backend components in isolation to ensure correctness early in development.

Goals

- Validate business logic and computations
- Ensure proper input validation
- Prevent regressions during development
- Increase confidence in backend reliability

Tools & Frameworks

- Language-specific testing framework (e.g., pytest, JUnit, or similar depending on implementation)
- CI pipeline (GitHub Actions / GitLab CI)
- Code coverage tools (target ~70–80% for critical modules)

Execution

- Run locally during development

- Automatically executed on every pull request
- Must pass before merging code

Units Under Test

Key areas include:

- API request handlers
- Input validation functions
- Database query logic
- Authentication functions

Excluded:

- UI rendering logic
- Third-party libraries
- Simple data models without logic

Test Design Approach

Tests will include:

- Normal cases
- Boundary values
- Invalid inputs

Example Unit Tests

Unit Test Unit: createUser(userData)

Purpose: Validates and creates a new user record.

Test Case Categories:

- **Valid Inputs:**
  - Properly formatted user data
- **Boundary Cases:**
  - Empty optional fields
  - Maximum field lengths
- **Invalid Inputs:**
  - Missing required fields
  - Invalid email format
  - Duplicate users
- **Boundary Values:**
  - Username length: 1-50 characters
  - Email length: Up to 255 characters
  - Password Minimum length: 8 characters

Sample Tests:

- Valid users are successfully created and stored
- Missing username returns validation error
- Duplicate email is rejected
- Empty input returns error

---

3. Integration Testing Plan

Integration testing ensures that independently developed components function correctly together.

Overview

After Alpha I, multiple system components interact:

- Frontend ↔ Backend API
- Backend ↔ Database
- Container ↔ Container communication

Failures at this level can include:

- Data mismatches
- API contract violations
- Configuration errors

Key Integration Points

1. Frontend → Backend API
2. Backend → PostgreSQL Database
3. Authentication flow
4. Container networking (Podman setup)

Test Environment

- Containerized environment using Podman
- Dedicated test database

- Predefined test user accounts
- Reset scripts for consistent test data

Environment characteristics:

- Repeatable
- Isolated
- Close to production setup

Integration Scenarios

Scenario 1: User Creation Flow

Steps:

1. User submits form in frontend
2. Frontend sends POST request
3. Backend validates input
4. Data stored in database
5. Response returned
6. UI updates

Expected Results:

- API accepts request
- Data persists correctly
- UI reflects success

Failure Handling:

- Invalid input → validation error
- DB failure → error response

- Auth failure → access denied

Scenario 2: Data Retrieval

Steps:

1. User requests data
2. API fetches from DB
3. Data returned to frontend

Expected Results:

- Correct data retrieved
- UI displays data correctly

Scenario 3: Authentication Failure Handling

1. User submits invalid credentials
2. Backend validates and rejects request
3. Error response returned
4. UI displays error message

Expected Results:

- Unauthorized access is blocked
- Clear error message is displayed
- System remains stable

---

4. Usability Testing Plan

Usability testing evaluates how effectively users can interact with the system.

Goals

- Identify confusing workflows
- Improve navigation and UI clarity
- Reduce user errors
- Ensure system meets user expectations

Context & Assumptions

Target users:

- Developers or technical users
- Familiar with web applications
- Limited tolerance for unclear workflows

System complexity:

- Moderate (containerized backend + UI)
- Requires clear interaction design

Risk:

- Poor usability reduces adoption and efficiency

Testing Methods

Method 1: Task-Based Testing

- 2–3 users
- Realistic tasks:
    - Create user
    - Retrieve data
    - Navigate interface

Method 2: Client Feedback Sessions

- Informal demos
- Feedback collected verbally

Method 3: Expert Review

- Team member evaluates UI usability

Metrics

- Task completion rate
- Time on task
- Error frequency
- User feedback/comments

Development Cycle Integration

- After Alpha I → Initial usability testing
- Before Alpha II → Refinements
- Before Acceptance → Final validation

Feedback is documented and prioritized for future sprints.

5. Testing Workflow and Quality Controls

Defect Management

Bugs are tracked using an issue tracking system (e.g., GitHub Issues).

Each bug includes:

- Description
- Steps to reproduce
- Severity level
- Associated feature/user story

Severity Levels

- **Critical:** System failure, no workaround
- **High:** Major functionality broken
- **Medium:** Partial functionality issue
- **Low:** Minor UI or non-critical issue

Team Responsibilities

- **Developers:** Fix defects and write unit tests
- **QA/Testers:** Execute integration and usability tests
- **Project Manager:** Track progress and enforce quality gates

---

Development Cycle Criteria

To Start a Cycle:

- Requirements defined
- Tasks assigned

To Complete a Cycle:

- All unit tests pass
- Integration tests pass
- No critical/high bugs remain

Acceptable Issues

- Minor UI issues
- Low-priority bugs with known workarounds

---

6. Conclusion

This software testing plan outlines a structured and comprehensive approach to ensuring the CRANC system meets its functional and non-functional requirements. By combining unit, integration, and usability testing with clear workflows and quality controls, the team ensures early detection, strong system reliability, and a positive user experience.

The emphasis on continuous testing, realistic environments, and iterative feedback ensures that the final product will be stable, functional, and user-friendly. This plan provides a practical and effective roadmap for delivering a high-quality software system.