Test Document

Version 1.0
DEC 12, 2025

SyncBridge

Group 6
CHEN Tianyu 1155210962
GUO Mingkang 1155211019
HUA Zifan 1155211089
Li Molin 1155191354
Qi Zihan 1155191644

Submitted in partial fulfillment
Of the requirements of
CSCI 3100 Software Engineering

1. Test Strategy & Approach

1.1 Development and Testing Workflow

This project follows a **feature-first development with supplemental automated testing** approach (non-strict TDD):

- Core business logic is implemented first for both frontend and backend, including authentication, role management, requirement submission, discussion, and status workflows.
- Automated tests are added post-implementation to cover critical user paths and backend APIs.
- Manual exploratory testing complements automated tests to validate edge cases and UI correctness.

1.2 Testing Types

The project employs the following testing types:

1. **Frontend Integration Tests**
    - Focus on App components and simulate full user interactions: "User Action → Mocked Network Request → UI Update".
    - Validate login flows, role-based UI differences, and form submission interfaces.
2. **Frontend Manual Testing**
    - Exploratory testing of core workflows:
        - Registration/Login
        - Role switching
        - Requirement creation, viewing, and discussion
    - Ensures that the visual presentation and user experience match expectations.
3. **Backend Automated Tests**
    - Validate API endpoints, database persistence, business logic, and permission enforcement.
    - Include tests for forms, functions/nonfunctions, messages, file handling, and WebSocket interactions.

1.3 Testing Tools and Frameworks

Frontend

- **Test runner:** Vitest
- **Test environment:** jsdom

- **UI testing library:** React Testing Library
- **Assertion library:** @testing-library/jest-dom

Key practices:

- render(<App />) — mount the component in jsdom
- screen.getByText / getByPlaceholderText / getByLabelText — query elements
- fireEvent.change / fireEvent.click — simulate input and clicks
- waitFor — wait for asynchronous UI updates
- vi.fn() and global.fetch — mock backend APIs
- localStorage — check token persistence

Backend

- **Test runner:** Pytest
- **Framework:** FastAPI TestClient
- **Database:** MySQL (test database instance)
- **Mocking:** unittest.mock for WebSocket manager and permission functions

2. Test Coverage

2.1 Frontend Coverage

- All automated frontend tests are in src/App.test.jsx.
- Core features covered:

1. **Login/Registration UI** — validates presence of input fields, buttons, and role selection.
2. **Role switching (client ⇄ developer)** — ensures radio buttons update state correctly.
3. **Login flow & requirement list** — mocks backend responses, verifies UI updates and token persistence.
4. **Role-specific UI** — client sees submission form; developer sees status update guidance only.

2.2 Backend Coverage

- Key components tested with Pytest and TestClient:

Component	Endpoints / Methods	Coverage
Forms	/form, /form/{id}, /form/{id}/status, /form/{id}/subform	CRUD operations, status transitions, permission enforcement
Functions / NonFunctions	/function, /function/{id}, /nonfunction, /nonfunction/{id}	Creation, update, deletion, role-based access control
Messages	/message, /messages, /message/{id}	CRUD operations, WebSocket broadcast simulation
Files	/file, /file/{id}	Upload, size validation, deletion, disk & DB consistency
WebSocket	/ws	Connection, authentication, ping/pong, presence notifications

3. Representative Test Cases

3.1 Forms

Test Case 1: Create Main Form

- **Objective:** Verify that a client can successfully create a main form.
- **Input:** Form data including title, budget, expected_time.

- **Expected Outcome:** Form persisted in MySQL with correct user_id and initial status.
- **Rationale:** Main forms are central; creation must succeed and associate with the correct user.
- **Testing Approach:** POST /form with valid payload; assert database entry exists.

Test Case 2: Update Form Status

- **Objective:** Ensure only authorized roles can update status according to workflow.
- **Input:** Status change request (e.g., preview -> available) by client.
- **Expected Outcome:** Status updated if permitted; unauthorized changes return HTTP 403.
- **Rationale:** Enforces business logic and permissions.
- **Testing Approach:** PUT /form/{id}/status; simulate client and developer roles; assert DB updates or forbidden responses.

3.2 Functions / NonFunctions

Test Case 3: Create Function

- **Objective:** Ensure functions can be added to a form by authorized users.
- **Input:** Payload with form_id, name, choice.
- **Expected Outcome:** Function record created; permission checks enforced.
- **Rationale:** Functions represent work items; misassignment must be prevented.
- **Testing Approach:** POST /function; assert database contains new function.

Test Case 4: Update NonFunction

- **Objective:** Validate editing of non-functional requirements.
- **Input:** Update payload (name, description).
- **Expected Outcome:** Only authorized users can update; changes reflected in MySQL.
- **Rationale:** Maintains integrity of non-functional requirements.

- **Testing Approach:** PUT /nonfunction/{id}; assert updated fields.

3.3 Messages

Test Case 5: Post and Broadcast Message

- **Objective:** Validate message creation and WebSocket broadcast.
- **Input:** Message payload (form_id, text_content).
- **Expected Outcome:** Message persisted in MySQL; broadcast called with correct payload.
- **Rationale:** Messages are real-time notifications; WebSocket must function reliably.
- **Testing Approach:** Mock WebSocket manager; POST /message; assert broadcast invoked.

Test Case 6: Delete Message

- **Objective:** Ensure messages can be deleted and WebSocket notified.
- **Input:** Message ID.
- **Expected Outcome:** Message removed from DB; broadcast delete event sent.
- **Testing Approach:** DELETE /message/{id}; assert message gone from MySQL and broadcast called.

3.4 Files

Test Case 7: File Upload Size Validation

- **Objective:** Reject files exceeding MAX_FILE_SIZE.
- **Input:** File > 10MB.
- **Expected Outcome:** HTTP 400 with validation error.
- **Rationale:** Prevents server overload and storage issues.
- **Testing Approach:** POST /file with oversized file; assert response 400 and no database entry.

Test Case 8: Delete File

- **Objective:** Verify file removal from disk and DB.
- **Input:** File ID.
- **Expected Outcome:** File deleted on disk; record removed from MySQL.
- **Testing Approach:** DELETE /file/{id}; assert file path does not exist and record deleted.

3.5 WebSocket

Test Case 9: WebSocket Connect and Ping

- **Objective:** Validate client can connect to WebSocket and receive pong response.
- **Input:** Valid JWT token and form_id.
- **Expected Outcome:** Connection accepted; ping/pong works; presence join broadcast called.
- **Rationale:** Ensures real-time communication infrastructure functions correctly.
- **Testing Approach:** Mock WebSocket manager; connect via FastAPI TestClient; send "ping"; assert send_to called for pong and presence update.

4. How to Run Tests

4.1 Frontend

- npm install
- npm run test

- Executes Vitest tests in src/*.test.jsx.
- Runs in jsdom; backend calls are mocked.

4.2 Backend

- pip install -r requirements.txt
- pytest

- Runs Pytest against FastAPI TestClient and test MySQL database.
- Tests include API validation, database assertions, and WebSocket mock checks.

5. Functional and Non-Functional Requirements Coverage

5.1 Functional Requirements Covered

The following functional requirements are **explicitly covered** by automated and/or manual tests:

Authentication & Authorization

- User registration with license validation
- User login and JWT token generation
- Role-based access control (client vs developer)
- Permission enforcement on all protected endpoints

Form Management

- Create main forms
- Retrieve form details and lists
- Update form status according to workflow rules
- Create and manage subforms (one active subform per main form)

Functional / Non-Functional Requirements

- Create, update, and delete functional requirements
- Create, update, and delete non-functional requirements
- Enforce ownership and role-based edit permissions
- Validate is_changed logic for subform-related updates

Messaging & Discussion

- Post messages associated with a form
- Retrieve message lists
- Delete messages
- Trigger WebSocket broadcasts for message events

File Handling

- Upload files with size validation
- Associate files with forms/messages
- Delete files from both disk and database

WebSocket Communication

- WebSocket connection authentication
- Ping / pong heartbeat
- Presence join and leave notifications
- Message broadcast simulation

5.2 Functional Requirements Not Covered

The following functional aspects are **not fully covered by automated tests** and rely on manual or future testing:

- UI responsiveness across different screen sizes
- Browser compatibility (Chrome, Firefox, Safari, etc.)
- Concurrent multi-user editing conflict resolution
- Long-running WebSocket connections under real network conditions
- Error recovery from unexpected backend crashes

5.3 Non-Functional Requirements Covered

Reliability

- API correctness validated through repeatable automated tests
- Database consistency checks after create/update/delete operations

Security

- Authentication required for all protected endpoints
- Authorization checks enforced for role-specific actions
- Oversized file upload rejection

Maintainability

- Modular test structure (by feature and API domain)
- Clear separation between frontend and backend test suites

5.4 Non-Functional Requirements Not Covered

- Performance benchmarking under high concurrency
- Load and stress testing
- Penetration testing and advanced security audits
- Disaster recovery and backup validation
- Accessibility (WCAG) compliance testing

6. Test Design Conditions

Test cases are designed under the following conditions:

- **Known user roles**: client and developer

- **Valid and invalid JWT tokens**
- **Clean test database state** before each test run
- **Isolated test data** created via fixtures
- **Controlled backend responses** (frontend tests use mocked APIs)
- **Deterministic environment** (no reliance on external services)

Boundary and negative conditions include:

- Unauthorized access attempts
- Invalid status transitions
- Missing or malformed request payloads
- Oversized file uploads
- Deleting non-existent resources

7. Test Case Generation Methodology

Test cases are generated using a **requirements- and feature-driven approach**:

1. **API Contract Analysis**
    - Each REST endpoint is mapped to expected inputs, outputs, and side effects.
2. **Business Logic Decomposition**
    - Core workflows (form lifecycle, role permissions, messaging) are broken into discrete scenarios.
3. **Positive and Negative Path Coverage**
    - For each feature:
        - Valid use cases (happy paths)
        - Invalid permissions

        - Invalid inputs

4. **Regression-Oriented Design**
    - Tests are written to guard against future regressions when modifying business logic.
5. **Representative Test Selection**
    - Not all permutations are tested.
    - Focus is placed on:
        - High-risk features
        - Core business flows
        - Security-sensitive operations

8. Tools, Techniques, and Test Execution Frequency

8.1 Tools and Frameworks

Frontend

- Vitest — test runner
- React Testing Library — UI interaction testing
- jsdom — browser environment simulation
- Mocked fetch and localStorage — API and auth simulation

Backend

- Pytest — test runner
- FastAPI TestClient — API testing
- MySQL (test instance) — persistence validation
- unittest.mock — WebSocket and permission mocking

8.2 Testing Techniques

- Black-box testing (API behavior validation)
- White-box testing (permission and business logic checks)
- Integration testing (API + database)
- UI integration testing (user actions → UI updates)
- Manual exploratory testing for UX validation

8.3 Test Execution Frequency

- **During development**:
  Tests are run locally after completing each major feature.
- **Before submission / deployment**:
  Full frontend and backend test suites are executed.
- **Regression testing**:
  Automated tests are rerun after any change to:
  - Authentication logic
  - Permission rules
  - Form or requirement workflows