TOBB ETÜ

University of Economics & Technology

Quality Assurance Plan

EYAY

Ali Türkücü

Esra Alparslan

Muhammed Yusuf Kartal

Yağız Can Akay

Contents

1. Quality Assurance Strategy

1.1 Overview

The QA strategy for HAYAI ET focuses on ensuring a safe, child-friendly, and functional environment. Given the target audience, our primary focus is not just code functionality, but also content safety and usability.

We adopt a "Shift-Left" testing approach, where testing is integrated early into the development phases.

Release Criteria (Definition of Done) To consider a version ready for release, the following criteria must be met:

- **Critical Bugs:** 0 open Critical (P0) defects.
- **Test Coverage:** 100% pass rate for Automated Integration Tests.
- **Performance:** Average AI generation time is within the acceptable threshold (< 60s).
- **Safety:** Moderation filters successfully block 100% of known harmful test datasets.

1.2 Testing Workflow & CI/CD Integration

To ensure code stability, we have integrated testing into our deployment pipeline. Testing is not a manual afterthought but a gatekeeper for merging code.

- **Local Development:** Developers run Unit Tests (pytest/Jest) locally before committing.
- **Continuous Integration (CI):** Upon pushing code to GitHub, an automated workflow triggers
    - **Commit Level:** Runs all Unit Tests. Failure blocks the commit.
    - **Pull Request (PR) Level:** Runs Integration Tests and Static Analysis. Code cannot be merged into the main branch unless these pass
- **Staging:** End-to-End (E2E) manual tests are performed on the Staging environment before final deployment.

1.3 Testing Methodologies

We will employ the following methodologies to verify the requirements defined in the Requirements Document:

- **Unit Testing:** Used to test individual components in isolation.
    - *Backend:* Testing FastAPI endpoints using pytest.
    - *Frontend:* Testing React components to ensure UI logic works as expected.
- **Integration Testing:** Focuses on the interaction between modules.
    - Verifying the data flow between React Frontend, FastAPI Backend, and MongoDB.
    - Validating the connection with external APIs (OpenAI).

- **Usability Testing:** Ensuring the interface is intuitive for children.
    - Verifying that icons are clear and fonts are readable.
    - Ensuring the "Predefined Comment" system works without confusion.
- **Security & Safety Testing:** Critical for COPPA/GDPR-K compliance.
    - Testing the "Parental Verification" mechanism during registration.
    - Verifying that the content moderation system blocks inappropriate image uploads.

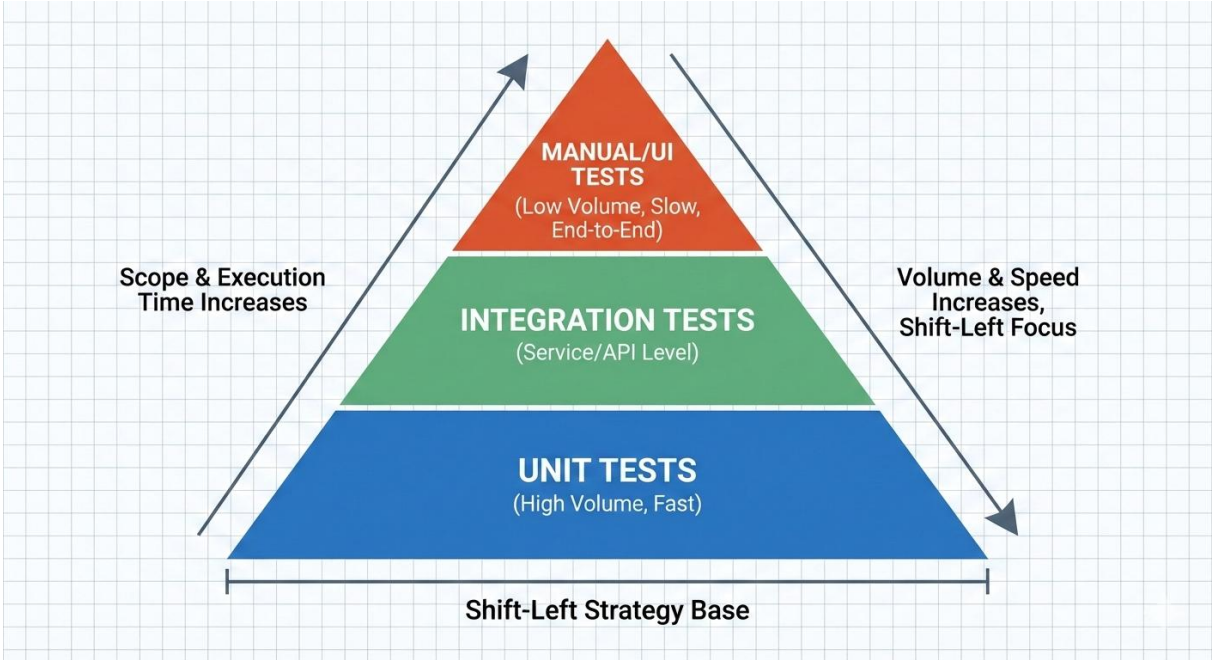

Figure 1 – Test Pyramid

Feature	Testing Type	Description
User Authentication	Automated	Verifying login credentials, token generation, and access denial for invalid users.
API Endpoints & Constraints	Automated	Validating HTTP response codes and enforcing constraints like File Size Limit (10MB) and Input Validation.
Safety & Moderation	Automated	Integration tests using Mock APIs to ensure inappropriate content is blocked as per the safety policy.
Input Restrictions	Automated	Unit tests ensuring UI components do not accept free-text input (Constraint SWC-002).
AI Image Quality	Manual	Visually reviewing AI-generated images to ensure they match the requested "Concept".
UI/UX Experience	Manual	Navigating the app as a child user to detect confusing layouts or navigation issues.

Table 1 – Test Types

1.4 QA Tools & Environments

- **Test Runner:** pytest (Backend), Jest (Frontend).
- **API Testing:** Postman & Swagger UI.
- **Browser Environment:**
    - Testing primarily on Chrome (Desktop)
    - Tablet/Mobile Web : iPad (Safari) and Android Tablet (Chrome)
- **Defect Tracking:** GitHub Issues & Projects.

2. Quality Factors & Metrics

The following quality factors are derived from the Non-Functional Requirements (NFR) and Project Success Criteria.

Quality Factor	Description	Measurement Metric
Safety	The system's ability to protect child identity and filter harmful content.	Percentage of inappropriate uploads successfully blocked by the moderation system.
Performance	The speed at which the system responds to user actions, specifically AI generation.	Time taken from "Upload" to "AI Result Display".
Usability	The ease with which a child can use the platform without adult assistance.	Percentage of test users (or proxy users) who can upload a drawing without help.
Reliability	The system's ability to function without failure under normal load.	Percentage of successful image uploads and generations without system crashes.

Table 2 – Quality Factors

3. Test Plan

This section details the test cases strictly aligned with the 4 Use Cases defined in the *Design Document* and the constraints defined in the *Constraint Tracking Form*. Each test case is assigned a priority level (Critical, High, Medium) based on its impact on the core system functionality and user safety.

3.1 Authentication & User Management

This section verifies that the entry points to the system are secure and compliant with the parental verification requirement (FR-1).

Test Case 1: Secure Registration & Login Flow

- **Associated Use Case:** Login & Sign-up
- **Priority:** Critical (P0)
- **Test Type:** Integration Testing
- **Technique:** Blackbox & Database Inspection
- **Tested Component:** auth_service.py (Endpoints: /register, /login)
- **Objective:** Verify that a user can register, their password is securely hashed, and a valid JWT is issued upon login (Checking Main Flow steps 1-3).
- **Pre-conditions:**
    - Database is clean (User child_01 does not exist).
- **Test Data:**
    - Username: child_01
    - Password: SecurePass123!
- **Steps:**
    1. **Sign-up:** Send POST /api/auth/register with username/password.
    2. **Hash Check:** Query the database users collection for child_01. Inspect the password field.
    3. **Login:** Send POST /api/auth/login with the same credentials.
    4. **Token Check:** Inspect the API response headers/body.
- **Expected Result:**
    - Step 1 returns HTTP 201 Created.
    - Step 2 confirms the password is **NOT** plain text (it should start with $argon2 or similar hash prefix).
    - Step 3 returns HTTP 200 OK.
    - Step 4 contains a valid access_token (JWT) and refresh_token.

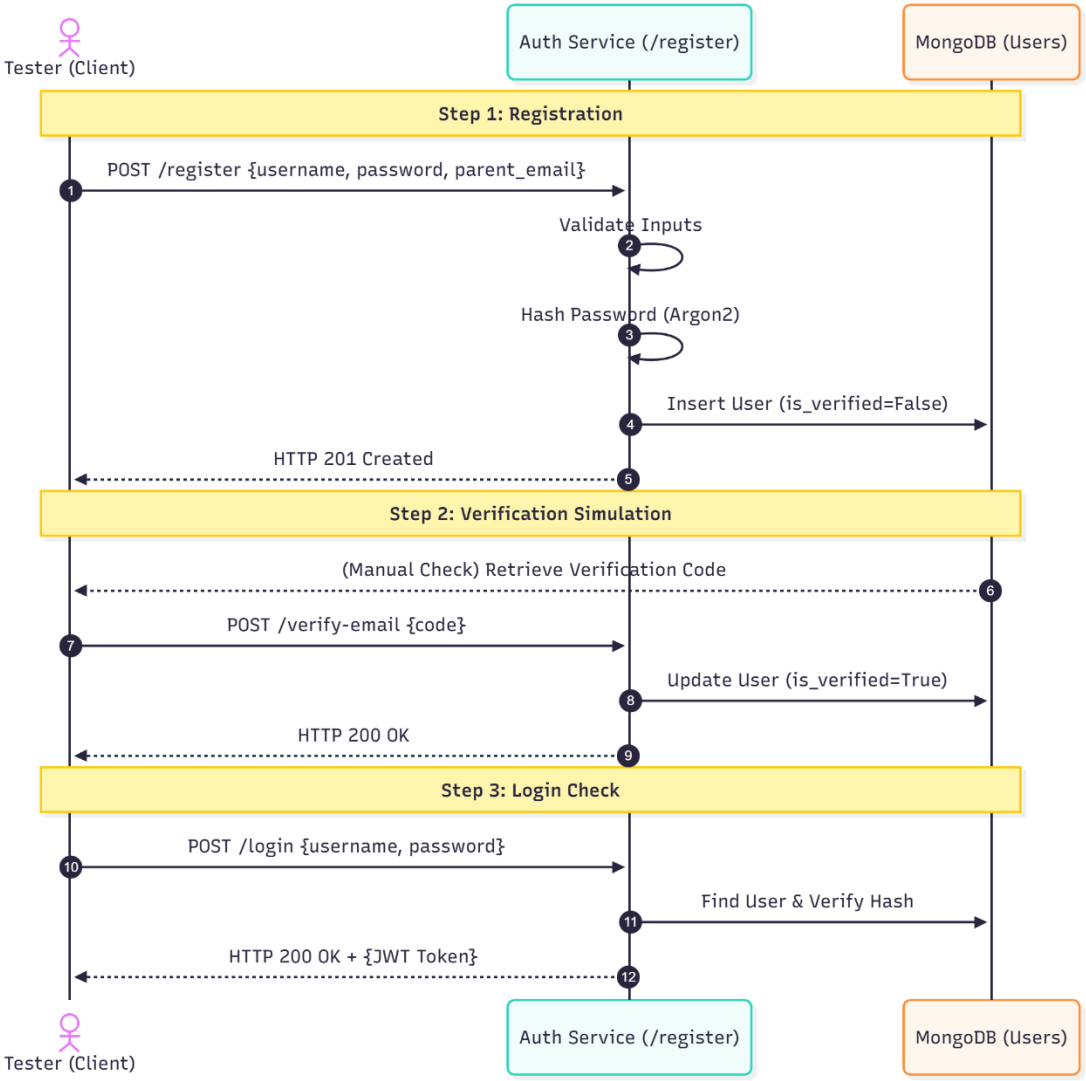

Figure 2 – Sequence Diagram of TC1

3.2 Core System Functionality

These tests validate the primary value proposition of the HAYAI ET platform: Uploading drawings, AI transformation, and viewing galleries (Use Cases 2 & 3).

Test Case 2: Upload & AI Transformation

- **Associated Use Case:** Upload Drawings & Transform with AI
- **Priority:** Critical (P0)
- **Test Type:** System Testing (End-to-End)
- **Technique:** Blackbox Testing

- **Tested Component:** /api/uploads, /api/ai/transform, AI Orchestrator
- **Objective:** Verify the complete flow: uploading a valid drawing, triggering the AI transformation, and receiving the finalized post.
- **Pre-conditions:**
    - Authenticated child user.
    - Test file drawing.jpg exists (Size < 10 MB).
- **Steps:**
    1. **Upload:** Send POST /api/uploads with multipart/form-data (drawing.jpg).
    2. **Verify Storage:** Check if response returns imageId and DB creates Image record with kind='original' and safe=false.
    3. **Transform:** Send POST /api/ai/transform with { imageId, theme: "space" }.
    4. **Processing:** Wait for AI Orchestrator to complete (Mock AI delay or poll status).
    5. **Final Check:** Retrieve the Post details.
- **Expected Result:**
    - Step 1 returns HTTP 201 Created.
    - Step 4/5 confirms that a new Image record (kind='ai') is created.
    - The Post status updates to approved (assuming moderation passes), and both original and AI images are linked.
    - Total process time is recorded in E2E logs (Target ≤ 120s).

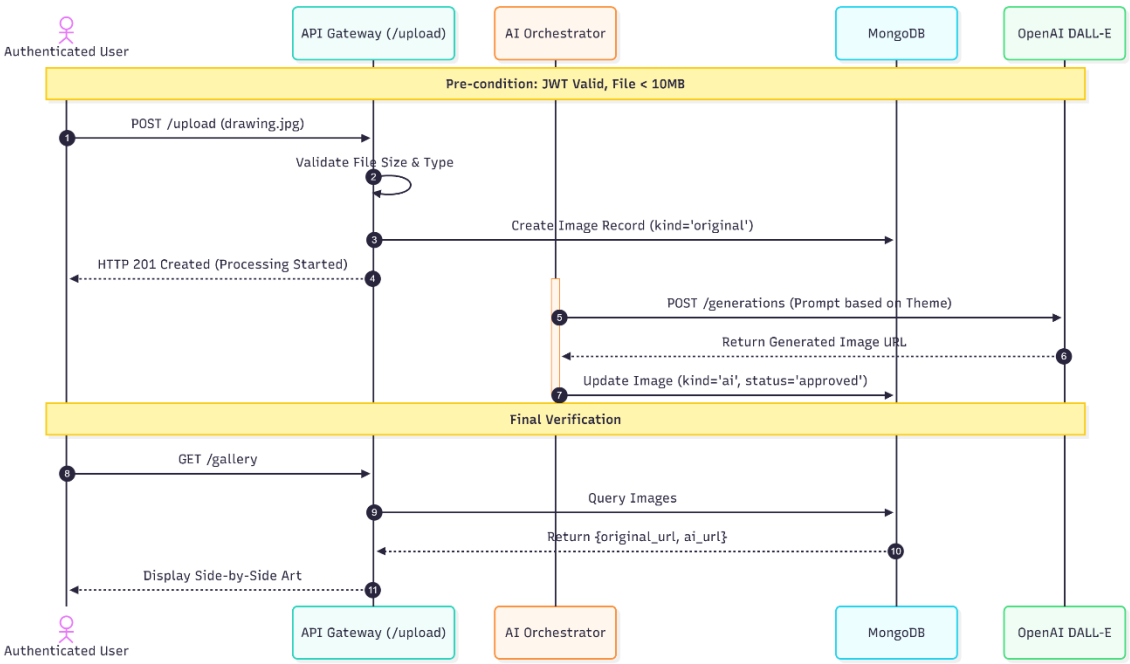

Figure 3 – Sequence Diagram of TC2

Test Case 3: Search & Public Gallery Visibility

- **Associated Use Case:** Search Friends & View Their Libraries
- **Priority:** Medium (P2)
- **Test Type:** Functional / Integration Testing
- **Technique:** Blackbox Testing (Data Filtering Verification)
- **Tested Component:** search_service.py, GET /api/users/search, GET /api/users/{id}/posts
- **Objective:** Verify that the search function works via prefix matching and that the gallery endpoint **strictly filters out** private and unapproved (rejected) posts.
- **Pre-conditions:**
    - User A (Searcher) is authenticated.
    - User B (Target) exists with username "AliVeli".
    - **Data Setup for User B:**
        - **Post 1:** Status Approved, Visibility Public.
        - **Post 2:** Status Approved, Visibility Private.
        - **Post 3:** Status Rejected (Moderation failed).
- **Steps:**

1. **Search:** Send GET /api/users/search?q=Ali (Testing prefix match).
2. **Verify Result:** Check if "AliVeli" appears in the JSON list with username and avatar.
3. **View Library:** Send GET /api/users/{user_b_id}/posts?visibility=public.
4. **Inspect Response:** Count the number of posts returned in the JSON array.

- **Expected Result:**
  - Step 2 returns HTTP 200 and includes "AliVeli".
  - Step 4 returns **exactly 1 post** (Post 1).
  - **Post 2** (Private) is excluded because the viewer is not the owner.
  - **Post 3** (Rejected) is excluded because it failed moderation.

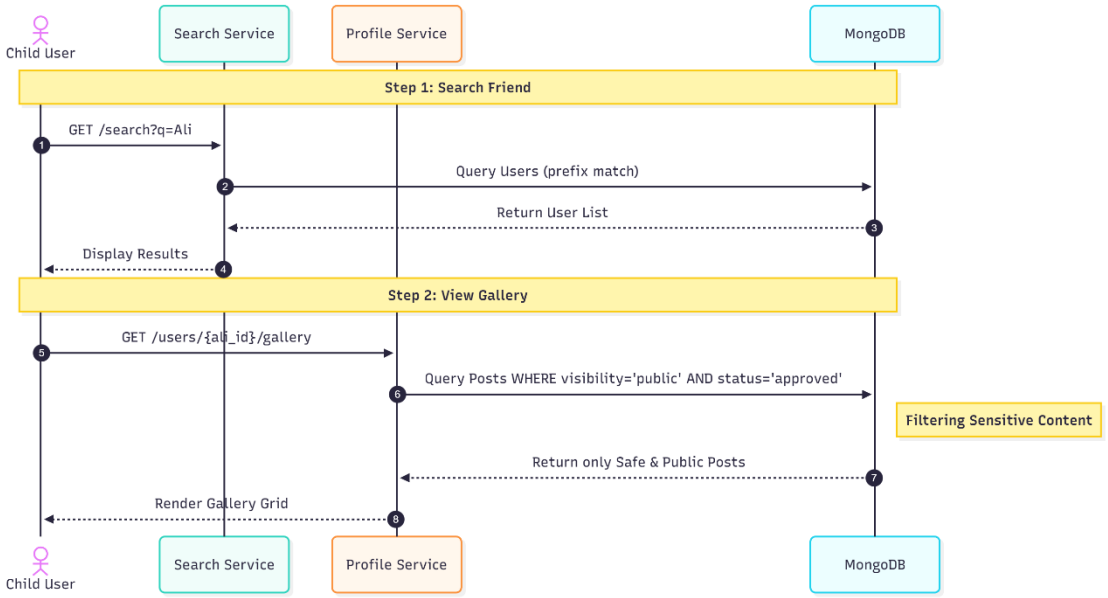

Figure 4 – Sequence Diagram of TC3

3.3 Safety, Security & Constraints

Given the target audience (children), this section specifically tests the system's resilience against misuse, strictly enforcing constraints SWC-002, SWC-004, and content moderation policies.

Test Case 4: Moderation Rejection

- **Associated Use Case:** Upload Drawings & Transform with AI
- **Priority:** Critical (P0)
- **Test Type:** Integration Testing
- **Technique:** Whitebox / Mocking
- **Tested Component:** Moderation Service, Post Service
- **Objective:** Verify that if the moderation service rejects an image, the post remains private and flagged (does not become visible).
- **Pre-conditions:**
    - User is authenticated.
    - Moderation Service is mocked to return { result: "rejected" } for the specific imageId.
- **Steps:**
    1. **Upload:** Send POST /api/uploads with a test image.
    2. **Transform:** Trigger POST /api/ai/transform.
    3. **Inspect DB:** Query the Post record associated with this image after processing.
    4. **Visibility Check:** Try to view this post from a *different* user's account (Public/Friend view).
- **Expected Result:**
    - The Post record exists but has a status of Needs review or Rejected.
    - The post visibility is restricted (Private).
    - Step 4 returns HTTP 403 Forbidden or the post is not listed in the feed (It is NOT deleted, but hidden, matching the UC-2 Exception: *"post stays private"*).

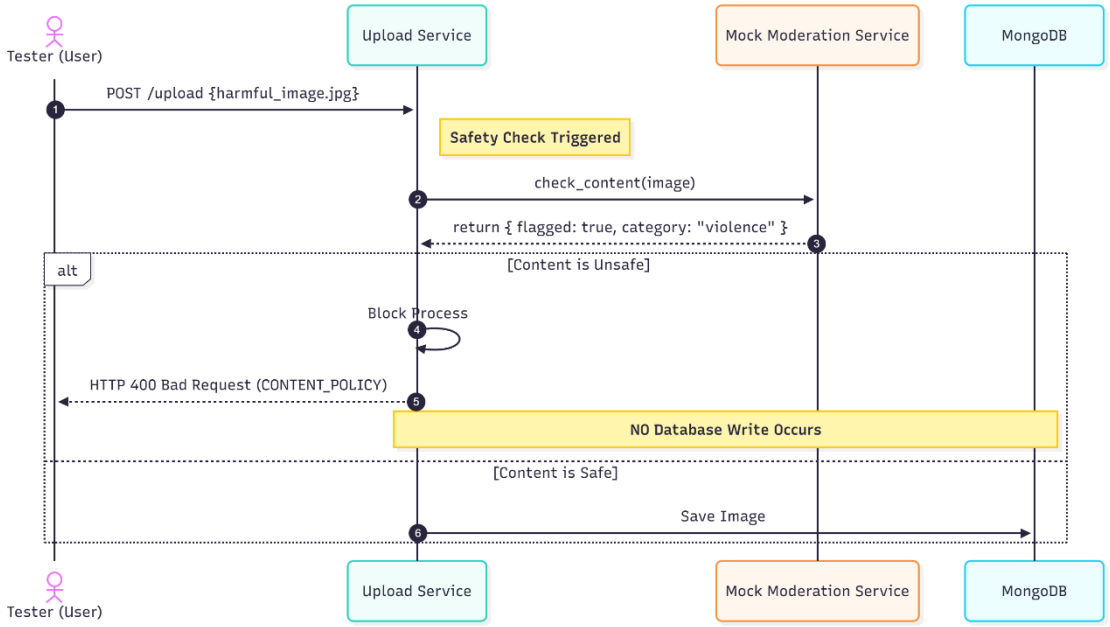

Figure 5 – Sequence Diagram of TC4

Test Case 5: File Size Restriction

- **Associated Use Case:** Upload Drawings & Transform with AI
- **Priority:** High (P1)
- **Test Type:** Unit / Input Validation Testing
- **Technique:** Whitebox Testing
- **Tested Component:** backend/utils/validators.py (Function: validate_file_size) & Middleware
- **Objective:** Verify that the system strictly enforces the 10 MB file size limit as defined in Constraint SWC-004 and returns the correct standard HTTP error code.
- **Pre-conditions:**
    - User is authenticated.
    - The backend service is running with default configuration.
- **Test Data:**
    - **File A (Boundary High):** limit_test.jpg (Size: 10.01 MB - Just above limit).
    - **File B (Boundary Low):** valid_test.jpg (Size: 9.99 MB - Just below limit).
- **Steps:**
    1. **Negative Test:** Send POST /api/uploads with **File A** (10.01 MB).

2. **Verify Response:** Check if the API returns HTTP 413 Payload Too Large.
3. **Verify Storage:** Query the images collection to ensure NO record was created for File A.
4. **Positive Test:** Send POST /api/uploads with **File B** (9.99 MB).
5. **Verify Success:** Check if API returns HTTP 201 Created.

- **Expected Result:**
  - Step 2 must return HTTP 413 (or 400 depending on config), confirming the request was blocked at the Middleware layer.
  - Step 3 must return 0 records for File A.
  - Step 5 must confirm the upload logic works for files within the limit.

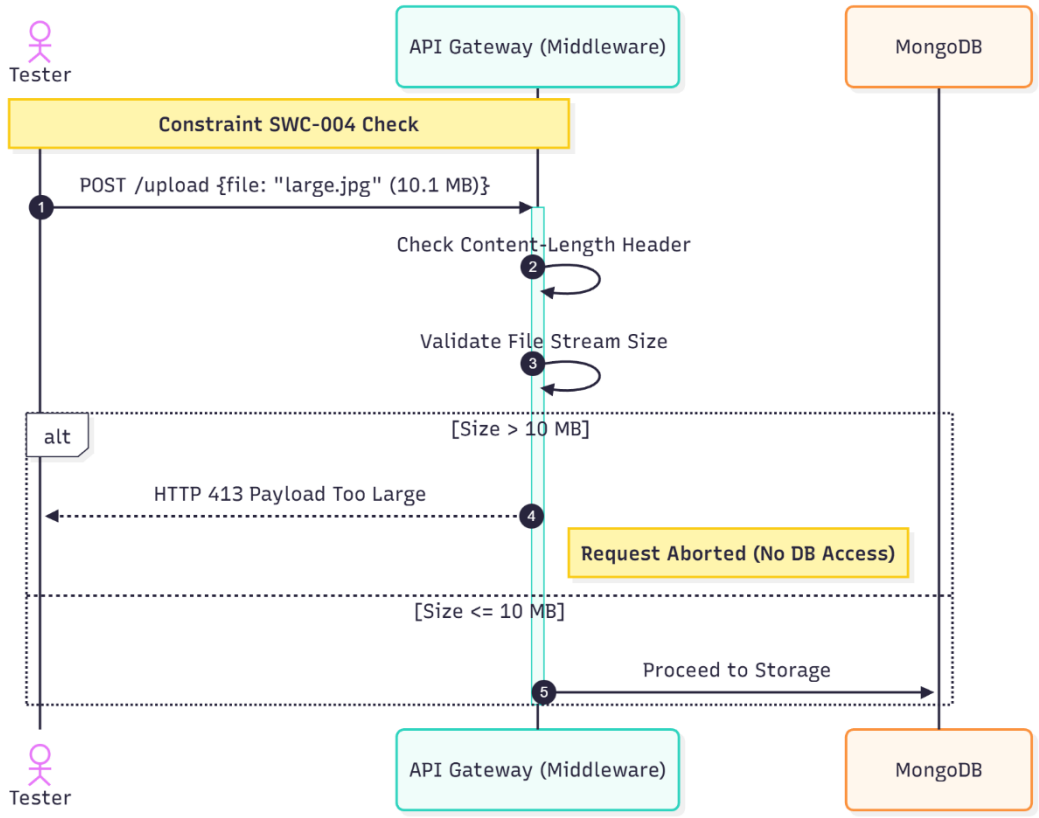

Figure 6 – Sequence Diagram of TC5

Test Case 6: Social Interaction Safety & Idempotency

- **Associated Use Case:** Like & Preset-Comment on Friends' Images
- **Priority:** High (P1)
- **Test Type:** Integration / Security Testing
- **Technique:** Blackbox Testing (API Fuzzing)
- **Tested Component:** interaction_service.py (Endpoints: /like, /comment)
- **Objective:** Verify that the "Like" action is idempotent (or toggles correctly) and that the backend **strictly rejects** any free-text payload sent via API tools (bypassing the UI), enforcing Constraint SWC-002.
- **Pre-conditions:**
    - User is authenticated.
    - Target Post ID post_123 exists and its status is approved.
    - User has not liked the post yet.
- **Steps:**
    1. **Like Action:** Send POST /posts/post_123/like.
    2. **Verify Like:** Check response status.
    3. **Valid Comment:** Send POST /posts/post_123/comment with payload { presetId: 5 }.
    4. **Security Attack (Free Text):** Attempt to bypass UI restrictions by sending POST /posts/post_123/comment with payload { content: "This is a bad text" } using an API client (e.g., Postman/cURL).
- **Expected Result:**
    - Step 2 returns HTTP 200 OK (or 201). Like count increments.
    - Step 3 returns HTTP 201 Created. Comment appears.
    - **Step 4 returns HTTP 400 Bad Request or 403 Forbidden.** The backend validation layer rejects the unknown field content or missing presetId. The text is **NOT** persisted in the database.

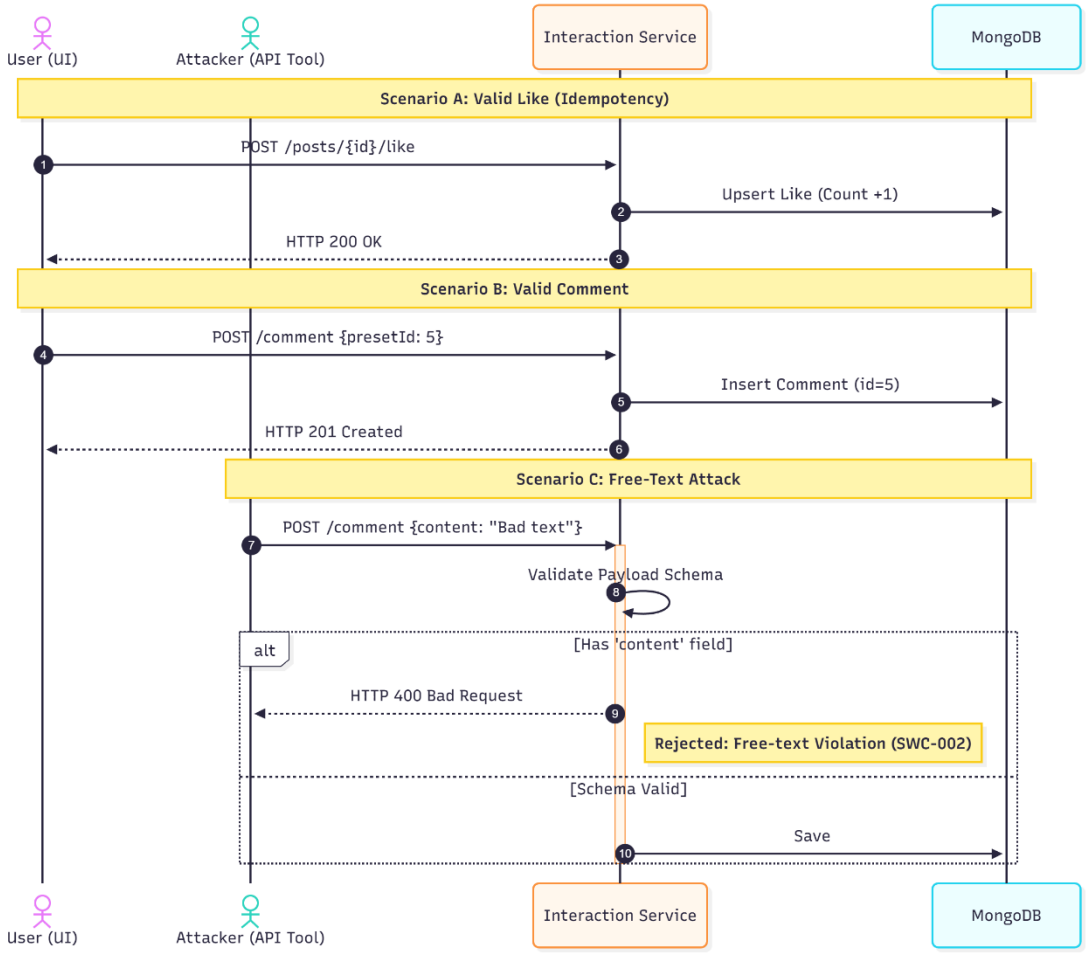

Figure 7– Sequence Diagram of TC6

4. Bug Tracking

As the project repository is hosted on GitHub, **GitHub Issues** serves as the primary centralized tool for tracking defects, ensuring full visibility and traceability throughout the development lifecycle.

4.1 Reporting Standards

The bug reporting process begins when a team member (acting as a tester) identifies a deviation from the expected behavior defined in the *Functional Requirements*. To ensure reproducibility and eliminate ambiguity, every reported issue must document the following three core elements:

1. **Steps to Reproduce:** A step-by-step guide to replicate the error.
2. **Expected Behavior:** What the system should have done according to the requirements.
3. **Actual Behavior:** What the system actually did (including error logs or screenshots).

4.2 Triage and Resolution Process

Once an issue is created, it is categorized using a labeling strategy to prioritize critical defects. Labels such as severity:critical are used for blockers (e.g., authentication failures), while component-specific labels like component:frontend or component:ai direct the issue to the responsible developer.

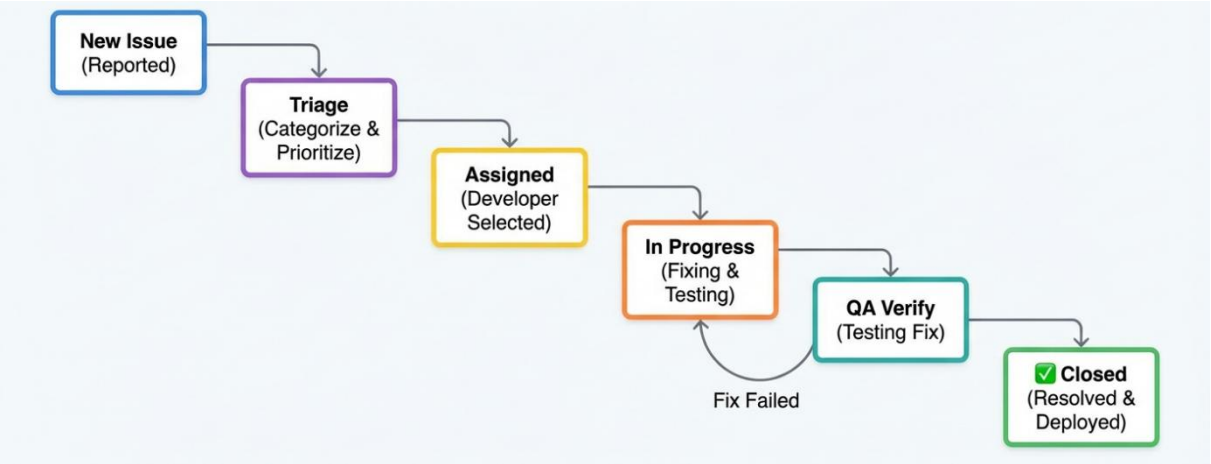

Figure 8 – Bug Lifecycle Flowchart

Bugs are resolved via a GitHub Pull Request (PR) that explicitly references the Issue ID, ensuring that every code change is inextricably linked to a specific defect.

5. Document-Specific Task Matrix

Document Task	Esra	Yağız	Yusuf	Ali
QA Strategy Definition				✅
Quality Factors & Metrics				✅
Test Plan				✅
Bug Tracking				✅
Formatting & Final Review	✅	✅	✅	✅

Table 3 – Document- Specific Task Matrix