Ho Chi Minh University of Science
Department of Information Technology

Course: Introduction to Software Engineering
Instructor: Mr. Truong Phuoc Loc
Class: 23CLC03
Group: 07

Team Members:

- Le Anh Duy – 23127011
- Tran Gia Huy – 23127199
- Nguyen Thanh Luan – 23127296
- Nguyen Thanh Tien – 23127539

Ho Chi Minh City, December 2025

1. Introduction

Purpose: This Test Plan outlines the strategy and approach for testing the *Manga Auto-Translation Platform*. It defines the scope of testing, including which features and modules will be tested and which will not, the testing methods to be used, required resources, schedule, and responsibilities. The goal is to ensure that the platform meets its requirements for functionality, reliability, security, and usability before release. This document will guide the project team in conducting systematic testing and tracking test progress, and serves as a reference for all stakeholders on how testing will be performed.

References: This plan is based on the project's requirements and design specifications. Key reference documents include:

- **Software Requirements Specification** (Group 07, November 2025) – detailing all functional and non-functional requirements for the manga platform.
- **Software Design Document** (Group 07, December 2025) – describing the system architecture and module design.

These documents provide the basis for identifying test items, features, and acceptance criteria. The Test Plan will also align with any relevant standards or guidelines provided in the course or by the instructor.

2. Test Items

The items (software components and modules) that will be tested under this plan are listed below. Each corresponds to major subsystems or features of the Manga Auto-Translation Platform:

- **Reader Subsystem:** The end-user facing module for readers, including the web reader interface and associated back-end services. This covers all functionality for regular users reading and interacting with manga content.
- **Uploader Subsystem:** The module enabling content contributors to upload manga chapters. It includes the uploader's web interface and back-end logic for submitting and managing content.
- **Admin Subsystem:** The administrator module for managing the platform. This includes the admin dashboard and back-end for user management, content moderation/approval, and configuration of platform settings.
- **Translation Module (OCR & Translation Integration):** The components that handle optical character recognition of manga images and machine translation of extracted text. This is an external API integration used by the platform, which will be tested via our system's ability to call the service and handle responses or errors.
- **Payment/Coin Module:** The functionality for in-platform purchases (coin wallet and payment integration). This covers the process of users purchasing coins (via payment gateway like ZaloPay or credit card) and spending coins to unlock premium chapters.

Note: All front-end (client) and back-end (server) parts of the above items are in scope. The specific versions are the initial development versions as of this project (there are no prior versions in production). Any updates or patches during testing will be tracked.

3. Features to Be Tested

The following key features and functionalities of the Manga Auto-Translation Platform will be tested. These are derived from the requirements and use cases for each module:

- **Reader Features:**
    - **User Registration and Login:** New users can create accounts (including via third-party OAuth like Google/Facebook), and existing users can log in. We will test validation (e.g. password rules) and security (encrypted passwords, HTTPS) for these flows.
    - **Manga Browsing and Search:** Users can browse the catalog of manga titles and use search by keywords or filters (genre, author, etc.). The search

functionality will be tested for accuracy and performance (e.g. results returned under a couple of seconds for common queries).

  - **Reading Manga Chapters:** Users can open a manga and read through its pages sequentially. Testing will cover page navigation (next/previous page), loading of images, and tracking of reading progress (the system should remember the last page read and update progress).
  - **On-the-fly Translation:** A core feature where the user can request translation of a manga page in real time. We will test that when "Translate" is triggered, the system sends the page image to the OCR/translation service, receives translated text, and overlays or displays the translated text on the page correctly. This includes testing multiple target languages if applicable, and ensuring the translation appears in the right location on the page.
  - **Comments and Ratings:** Readers can comment on chapters and give ratings. We will test adding a comment, viewing comments, and the rating system (e.g. one user one rating, average rating calculation). This ensures comments/ratings are stored and retrieved properly.
  - **Content Sharing:** The ability for users to share manga or chapters via external platforms (e.g. sharing a link on social media). We will test that the share buttons or links generate the correct URL or interface (where applicable) for external sharing. (Note: The actual posting on external sites might not be fully automated, but the link generation will be verified.)
  - **Payment & Access Control:** If a chapter is premium (requires purchase), we will test the coin purchase flow and chapter unlocking. This includes depositing money to convert to in-app currency (coins) through the payment gateway (in test mode), verifying that the coin balance updates, and then spending coins to unlock a chapter. After unlocking, the chapter should be readable. We'll test scenarios like successful payment, payment cancellation, insufficient coin balance for purchase, etc., to ensure proper handling.
- **Uploader Features:**
  - **Request Uploader Role:** A regular user can request to become an uploader. We will test the submission of an uploader request and ensure it is recorded and visible to admins for approval.

    - **Manga Chapter Upload:** Uploaders can upload new manga chapters. This involves entering metadata (title, genre, summary, language) and uploading image files for pages. We will test the entire upload form, including validation (required fields, acceptable image formats/sizes) and the uploading process of multiple page images.
    - **Draft Management:** After uploading, the chapter may be saved as a draft. We will verify that an uploader can save a draft and later resume editing it, edit metadata, reorder or replace page images, etc., before submitting.
    - **Submit for Admin Approval:** We will test the process of the uploader submitting a completed chapter for review. Once submitted, the chapter's status should change to "Pending" (or similar) and become visible in the admin's queue. We will ensure the uploader cannot further edit after submission unless rejected.
    - **View Submission Status and Stats:** Uploaders should be able to see the status of their uploaded chapters (e.g. Draft, Submitted, Approved/Published, or Rejected). We will test status updates (for example, when an admin approves or rejects a chapter, the uploader sees the updated status). If the platform provides any simple analytics to uploaders (like view counts or ratings on their chapters), we will verify those are displayed correctly.
- **Admin Features:**
    - **User Management:** Admins can manage user accounts. We will test functionalities such as approving uploader requests (promoting a user to uploader role), demoting or changing roles, and locking or suspending user accounts that violate rules. This includes verifying that suspended users can no longer log in or perform certain actions.
    - **Content Moderation & Approval:** A central admin duty is reviewing new chapter submissions. We will test the admin's ability to view a list of pending chapters, open and inspect a chapter's content (pages and metadata), and then approve or reject it. On approval, the chapter should become live/public; on rejection, it should send feedback to the uploader (if the system provides a rejection reason mechanism). We will also verify that admin can edit or remove any content if needed (e.g., delete a published chapter or fix metadata).

    - **Featured Content Management:** Admins can mark certain manga or chapters as "featured" or highlighted (for example, to promote on the homepage). We will test adding a chapter to a featured list, setting the duration it should stay featured (and that it auto-expires or can be manually unfeatured). This ensures promotional content is handled correctly.
    - **Platform Configuration (Integration Settings):** Where applicable, we will test admin controls over integrated services. For instance, if the admin interface allows configuring the translation service API keys or toggling certain translation features, we will verify those settings. Similarly, if admins can adjust payment settings or view transaction logs, we will include tests for those as needed. (These configuration aspects are minor, and many might be one-time setup, but any exposed to admin UI will be tested.)
    - **Reporting/Analytics (if implemented):** In case the admin dashboard provides any summary statistics (e.g., number of readers, popular genres, revenue from coins), we will do basic verification of those displays. (If such "director" or management reports are limited or not in this phase, this can be minimal. Any implemented analytics or logs visible to admin will be spot-checked for reasonable data.)

Additionally, **Error and Edge Case Handling** will be tested across features. For example, we will simulate what happens if the OCR/translation service is down or returns an error while a user requests a translation (the system should handle it gracefully, e.g., show an error message but not crash). Similarly, we will test behavior when an uploader uploads an extremely large image, or when a payment transaction fails, to ensure the system responds correctly. The focus is on verifying **functional requirements**: that each use case and user story performs as expected under normal and typical error conditions.

4. Features Not to Be Tested

The following features or aspects are **out of scope** for this test plan (they will *not* be formally tested in this phase):

- **Translation Quality and Accuracy:** We will not evaluate the linguistic correctness or contextual accuracy of the machine-generated translations themselves. The translation engine's performance (in terms of producing a perfect translation) is considered outside our scope. Our tests will check that translations are invoked and displayed, but not whether the translation is "good" or perfectly accurate in meaning.

- **Internal Mechanics of External Services:** We will not test the internals of third-party services such as the OCR/translation API or payment gateway beyond our integration points. For example, we won't test the payment provider's own fraud detection or the OCR engine's internal algorithms. We will only verify that our system properly calls these services and handles their responses or failures (timeouts, errors), not the correctness of the services themselves.
- **Non-Functional UI/UX Polishing:** Detailed user interface aesthetics and minor usability preferences will not be part of formal testing. While we will note obvious UI bugs that hinder functionality, we will not formally test cosmetic details (such as exact pixel alignments, font choices, or minor layout issues) or conduct user experience surveys. Similarly, aspects like support for older browsers or very specific responsive design nuances are not a priority in this test cycle unless they critically affect usage.
- **Performance and Load Testing:** Rigorous performance testing (e.g. measuring response time under high concurrent load, stress testing the system with thousands of users or large volumes of data) is not included in this test plan. We do have performance goals (e.g. page translation in under ~5 seconds and search results in under 2 seconds as per requirements), and we will observe performance informally during testing. However, dedicated load testing or benchmarking is not scheduled due to time constraints. Any obvious performance bottlenecks encountered with normal usage will be noted, but no formal load test scripts will be executed.
- **Future or Unimplemented Features:** Any features planned for future implementation (beyond the current project scope) will not be tested now. For instance, if an advanced "Director" analytics dashboard or additional social features were discussed but not developed in this phase, they are out of scope. We focus only on what has been implemented for this project milestone.

By clearly defining out-of-scope items, we ensure the testing effort is concentrated on the agreed project deliverables. Any excluded areas can be revisited in future testing cycles if needed.

5. Test Strategy

Our overall test approach is **black-box, functional testing** focused on the end-to-end behavior of the application. The testing will be conducted from a user's perspective without access to the source code, ensuring we validate the system against its requirements and use cases.

Testing Levels: We will employ the following levels of testing:

- *Unit Testing:* Developers on the team will perform informal unit tests on their own modules during implementation (e.g., checking individual functions like text extraction utility or database queries in isolation). This is not the focus of this plan but serves to catch low-level bugs early.
- *Integration Testing:* As components (front-end, back-end, external APIs) are integrated, we will test interfaces between them. For example, testing the flow from the front-end action through the API to the database and back. Integration testing will ensure that subsystems (Reader-Uploader-Admin with the server, server with OCR API, server with payment API, etc.) work together correctly.
- *System Testing:* This is the core of our strategy and where formal test cases will be executed. System testing treats the entire platform as a black-box and verifies all the features (as listed in section 3) in a fully integrated environment. We will simulate real user scenarios (end-to-end use cases), covering normal usage as well as edge cases and error conditions.

All testing will be **functional** in nature, verifying that each function yields expected outputs given certain inputs or user actions. Since we are focusing on black-box testing, we will not be doing code review-based testing or white-box tests like branch/path coverage; instead, we derive our test cases from the requirements and design. Each test case will have defined expected results (from the SRS or UI specifications), and the actual results observed will be compared against them.

Test Case Design: Test cases are derived from use cases and requirements. For each feature, we create test scenarios including: typical valid cases, boundary or edge cases (e.g., extremely long input in a form, maximum allowed pages in an upload, etc.), and error situations (such as network loss during an upload, or invalid credentials during login). We will ensure traceability by mapping test cases to specific requirement IDs or user stories to confirm coverage. Each test case will outline test data, steps, and expected outcomes. The team will prioritize test cases so that critical functionality (like login, reading, uploading, approvals, payments) is tested first, ensuring any high-impact bugs are found early.

Testing Methodology: We plan to execute most tests manually through the web UI and APIs, mimicking user actions:

- For UI-centric scenarios (like reading, commenting, uploading via the website), testers will follow written steps in a browser and verify the results on-screen.
- For certain back-end validations or where precise control is needed (e.g., verifying API responses, injecting error conditions), we will use tools like Postman or custom scripts to directly call the API endpoints with various inputs.

- We will also create a few **smoke tests** to run whenever a new build is deployed (ensuring basic functionality like site up, login, and main page loading works). After any major code update, a quick smoke test pass will verify that no critical regression (showstopper) has occurred before proceeding with detailed testing.

Defect Handling: Whenever an actual result deviates from the expected result, a defect will be logged (in our chosen tracking tool or document). The development team will then address the issue, after which the test will be re-run to verify the fix. We will categorize defects by severity (Critical, Major, Minor) to help prioritize fixes. Our strategy emphasizes fixing critical and major issues before release; minor cosmetic issues may be scheduled for later unless time permits.

Regression Testing: As fixes are applied, we will re-test not only the specific fixes but also perform regression testing on related features to ensure no new bugs have been introduced elsewhere. Given the project's short timeline, regression will be targeted to areas of code that changed – for instance, if the upload module code is altered, we will re-run all upload and maybe some reader tests (since new chapters might affect reading) to be safe.

In summary, our test strategy is to **start testing early** in parallel with development (via unit/integration tests), and then conduct a thorough system test cycle once the application is feature-complete. The approach is **manual, black-box, requirement-driven** testing. We will concentrate on verifying that each user-facing requirement is met and that the system is robust against common error conditions. Non-functional attributes (security, basic performance, etc.) will be observed and any serious issues noted, but the primary focus is to validate functionality. Security will be considered by attempting some common improper actions (e.g., ensure a user cannot access admin pages, test form inputs for things like SQL injection or script injection on a small scale). Any findings will be reported, though a full security audit is not in scope.

This strategy ensures that by the end of the testing phase, we have confidence that the platform behaves correctly for end users and administrators in all the primary scenarios, and that we have identified and addressed the most important defects.

6. Test Deliverables

The testing process will produce several deliverables (artifacts) to document what was tested and the outcomes. The following are the key test deliverables for this project:

- **Test Plan Document:** (this document) Outlines the test scope, approach, resources, and schedule. It is a reference for the team and stakeholders on how testing will be conducted.

- **Test Case Specifications:** A detailed set of test cases with step-by-step procedures, input data, and expected results for each feature in scope. This may be in the form of a separate document or spreadsheet. Each test case will be uniquely identified (e.g., TC-01 Login Success, TC-02 Login Wrong Password, etc.) and traceable to requirements.
- **Test Data Sets:** Any specialized data needed for testing, such as dummy user accounts (test usernames/passwords), sample manga content (images and text for upload), and test credit card numbers or payment accounts (in sandbox mode). These will be prepared ahead of execution and documented (for example, a list of test user accounts with their roles).
- **Test Execution Log / Results Report:** During test execution, we will record the outcome of each test case (Pass/Fail and any observations). This might be done in the test case document itself or a separate log. At the end of the cycle, we will produce a summary report of test results, including how many test cases passed, failed, or were blocked, and a list of open defects.
- **Defect/Bug Reports:** For each defect found, a bug report will be created. This includes a description of the issue, steps to reproduce, severity, and status. All bug reports will be tracked (for example, in a bug tracking system or a shared spreadsheet). The collection of these reports (the *bug log*) is a deliverable that shows what issues were identified and how they were resolved (including which were fixed and re-tested, or any known issues deferred).
- **Test Summary and Closure Report:** At the conclusion of testing, we will prepare a brief summary report that recaps the testing activities and outcomes. It will highlight the major testing milestones, number of tests executed, number of defects found and fixed, any remaining known issues, and an assessment of whether the software meets the quality criteria for release. This document serves as a record that testing was completed and, if applicable, includes a sign-off indicating that the product is ready for deployment or final evaluation.

Additionally, supporting deliverables include **meeting notes or status reports** during the testing period (to communicate progress and any blockers to the team), and the **approved requirements and design documents** (used for test case creation) which have been listed as references. All deliverables will be stored in the project's documentation repository for internal access and review.

7. Testing Tools

We will utilize several tools to facilitate testing, both for execution and for tracking:

- **Web Browser & Developer Tools:** The primary interface for testing will be modern web browsers (e.g., Google Chrome, Firefox). We will use built-in developer tools for actions like inspecting network calls, simulating different screen sizes (for responsive layout checks), and checking console logs for errors during test execution. This helps in diagnosing front-end issues on the fly.
- **Postman (or similar API client):** Postman will be used to directly test the backend API endpoints. This is useful for testing scenarios like authentication, manga upload, or translation calls in isolation, and for sending edge-case requests that might be hard to simulate via the UI. Postman also allows us to automate sequences of API calls (for example, login then upload then translation) and check responses, which complements manual UI testing.
- **Database Viewer/Client:** While not for traditional end-user testing, we will use a database client (such as pgAdmin for PostgreSQL or a command-line psql) in a read-only manner to verify data changes if needed. For instance, after a test we might query the database to ensure data was correctly inserted (e.g., a new user record after registration, or coin balance updated after payment). This helps verify back-end effects of front-end actions.
- **Version Control and CI Pipeline:** The project uses Git for version control (GitHub repository). We have a Continuous Integration (CI) pipeline (e.g., GitHub Actions) that runs on each commit. The CI is configured to run any automated test scripts (if we add some unit tests or lint checks) and to deploy the latest build to our test environment. The CI pipeline will help ensure that the build is successful and potentially run a small suite of sanity tests. We will monitor CI results for any failing build or test.
- **Issue Tracker:** To manage test progress and bugs, we will use an issue tracking tool (for example, GitHub Issues or a simple Trello/Jira board). This will record all identified defects and their fix status, as well as tasks related to testing (like "prepare test data" or "execute test suite for Admin module"). This ensures nothing falls through the cracks and team members have visibility on what's being addressed.
- **Communication Tools:** (For completeness) We will continue to use our team communication channels (e.g., Discord or email) to discuss testing findings quickly. While not a testing tool per se, prompt communication will aid the testing process, especially when clarifying expected behavior or seeking quick fixes.

If time permits and the project complexity warrants, we may also write a few **automated test scripts** (for example, using Selenium or a JavaScript end-to-end testing library like Cypress) for regression on critical flows (like logging in and reading a chapter). However, given the timeline, our plan primarily relies on manual testing and the tools listed above for assistance.

In summary, the chosen tools will help us thoroughly exercise the application (through UI and API), verify outcomes, and track any issues that arise in an organized manner. The combination of browser dev tools and Postman covers most of our needs for functional verification, while the CI/Issue tracker ensures we maintain quality and organization.

8. Environment

Testing will be carried out in an environment that closely mirrors the intended production setup of the manga platform. Below are the details of the test environment, including the tech stack and any specific configurations or mock services:

- **Frontend Technology:** The front-end is a web application built with Next.js (a React-based framework). For testing, we will deploy the latest build of the Next.js app on a local machine and a staging server as needed. The application will be tested primarily on Google Chrome (latest version) on Windows 10, which is our baseline. We will also do basic sanity checks on another major browser (Firefox) and ensure responsiveness by resizing the browser or testing on a mobile device (Android Chrome), as the platform is supposed to be mobile-friendly. The front-end communicates with the backend via RESTful API calls. We will host the front-end locally at a test URL (e.g., http://localhost:3000 or an internally hosted URL) during testing.
- **Backend Technology:** The backend is implemented in Node.js (for example, using Express.js or NestJS framework) and provides REST API endpoints for all functionalities (authentication, manga retrieval, uploading, commenting, etc.). It connects to a **PostgreSQL database** for data storage. For testing, we will either use the backend running locally (http://localhost:<port> for API) or on a dedicated test server. The **test database** will be a separate PostgreSQL database instance/schema initialized with sample data (so as not to interfere with any development or future production data). We will set up initial test data including some sample manga entries, user accounts (with roles Reader, Uploader, Admin), and perhaps pre-loaded translations or coin balances for specific scenarios. This controlled data will allow consistent test results.

- **External Services:** The platform relies on third-party services for OCR and machine translation, as well as an external payment gateway:
  - *OCR & Translation API:* We intend to use a cloud service (for example, Google Cloud Vision for OCR and Google Translate API, or a similar provider) for text extraction and translation. In the test environment, we will use our API keys and treat this as a black-box external dependency. We must ensure the test environment can access the service (internet connectivity and valid credentials). There is no true "mock" for these services provided by the vendor, so we will perform tests live but be mindful of usage limits. For certain error-condition tests, we might simulate the service being unavailable by using incorrect credentials or disabling network calls (to see how our system handles failures). The response times of these services in testing will be observed, but as noted, we won't stress-test them.
  - *Payment Gateway:* The platform uses a payment gateway (e.g., **ZaloPay**) for coin purchases. We will use the gateway's **sandbox mode** or test API if available. Typically, such gateways provide test merchant IDs and allow using fake card numbers or QR codes. Our test environment will be configured with the sandbox credentials so that no real financial transactions occur. We will verify payment flows using this test setup. For example, when testing coin purchase, we expect to be redirected to a ZaloPay sandbox page and then simulate a success callback to our site. If the payment provider doesn't have a sandbox, we will simulate the callback with a test script or consider this scenario carefully (but ZaloPay and similar usually have testing options).
- **Deployment and Access:** We plan to deploy the application in a staging environment that the team can access. This might be a cloud VM or just running on one team member's machine and accessible via LAN. The environment will have all necessary environment variables configured (API keys for OCR/Translate, keys for payment test mode, etc.). We will maintain a .env.test file with these configurations to ensure consistency. The environment will be reset/redeployed as needed (especially if the database needs to be restored to a baseline state for regression testing).
- **Test Data Management:** As mentioned, a controlled set of test data will be used. We will document initial states (e.g., "Admin account exists: admin@example.com, password X; Uploader test account exists and has Y coins; Sample Manga ABC with 3 chapters loaded for search tests," etc.). After certain tests (like an upload), the data in the environment will change. We might need to restore the database or have

scripts to add/remove data to reuse the environment for multiple test runs. If possible, we will utilize database transactions or cleanup scripts to return the system to a known state between test case groups (for example, remove a test chapter that was uploaded or reset a user's coin balance after a payment test).

- **Tools and Simulated Components:** No full mock server is planned for the API – we will test on the real API. However, for some specific tests (like simulating the translation service being down), we might use a small stub or toggle in our backend if one exists (for instance, a configuration to use a "mock translation mode" that returns a fixed error). If such a feature isn't built-in, testers will manually simulate by disconnecting network or altering API keys temporarily for that test, then restoring them. We do not anticipate needing separate hardware or special devices beyond standard PCs and smartphones for mobile view testing.

In summary, the test environment consists of the Next.js front-end and Node.js/PostgreSQL back-end deployed in a controlled setting, using real integrations with external services in their test modes. This environment is as close to production-intent as possible so that test results are valid. All testing will be done on this environment. Prior to starting formal testing, the team will do a smoke check of the environment to verify everything is correctly configured (e.g., the front-end can communicate with back-end, database connections work, API keys are set, etc.). Any differences between the test environment and expected production (like using sandbox APIs or a smaller server instance) will be noted, but we ensure that core functionality remains the same.

9. Roles & Responsibilities

The testing effort will be a team activity, with specific roles assigned to ensure clarity in responsibilities:

- **Test Lead – Le Anh Duy:** The Test Lead is responsible for overall test planning and coordination. He prepared this Test Plan and will also oversee the creation of test cases. During execution, the Test Lead will monitor progress, ensure that testers have what they need (environment access, accounts, data), and coordinate with developers for quick defect resolution. He will also consolidate test results and communicate status to the team and instructor. In essence, Le Anh Duy acts as the quality coordinator, making sure the testing phase runs smoothly and on schedule.
- **Test Engineers/Executors – Tran Gia Huy & Nguyen Thanh Luan:** These team members will take primary responsibility for designing and executing test cases across the application. They will each focus on certain areas: for example, one might focus on Reader and Uploader feature tests while the other focuses on Admin

and integration (this division can rotate as needed for cross-checking). Their duties include preparing test data for their scenarios, running the tests step-by-step, logging defects when encountered, and re-testing after fixes. They will also review each other's test cases and results to ensure nothing is missed (peer review of testing). In practice, since our team is small, all members contribute to testing, but these two will dedicate significant time to thorough testing of the system's functionalities.

- **Developers (for Unit Fixes & Support) – Nguyen Thanh Tien and all team members:** Each team member is a developer for some part of the system (front-end or back-end). Developers are responsible for performing initial unit tests on the code they write and fixing any bugs discovered either in their own unit tests or reported by the Test Engineers. For example, if a bug is found in the upload feature, the team member who implemented that feature will take lead in diagnosing and correcting the issue. Developers will also assist in setting up the test environment (deploying latest builds, ensuring DB is seeded with data) and might pair with testers to reproduce tricky issues. While the dedicated testers execute the bulk of test cases, developers may also execute some tests, especially in areas where they have expertise or when doing quick smoke tests after a new deployment. Essentially, all four members will be involved in the testing phase at various points, but the developers ensure rapid turnaround on fixes and technical guidance on expected behavior.
- **Reviewer/Approver – (Project Supervisor / Instructor):** While not part of the student team, the instructor (Mr. Truong Phuoc Loc) or a designated project supervisor will serve as a review authority for the test plan and results. The Test Lead will submit this Test Plan for approval to ensure it meets the course's expectations. The instructor or TA may review test cases or observe demonstrations of certain tests (like during a project evaluation). They effectively act as a client or quality auditor. It is their role to ultimately approve that the testing is sufficient and that the system is ready (this corresponds to the "Approval" section at the end of this document). Internally, the team may also treat the Team Leader or another senior member as a reviewer of test artifacts to double-check quality before seeking instructor approval.

In addition to the above roles, **all team members share responsibility** for quality. This means anyone who spots an issue (even outside their assigned test cases) should log it, and everyone should collaborate to resolve it. The division of roles ensures coverage and efficiency, but communication lines remain open—regular syncs will be held during the testing period so that testers can update developers on new defects and developers can update when fixes are ready.

This clear delineation helps prevent gaps (for example, ensuring someone is focused on coordinating, while others are executing and others fixing) and avoids duplication of effort. It also aligns with academic project roles where everyone gets experience in both development and testing aspects, with one person taking the lead in organization.

10. Schedule & Milestones

Testing activities will be integrated into the overall project schedule. Below is a timeline with major milestones for testing, aligned with the project's development milestones:

- **Test Plan Completion – (Target: Dec 10, 2025):** By this date, the Test Plan (this document) is written and reviewed by the team, then submitted to the instructor for approval. Having the plan early ensures the team has a clear roadmap for testing as development wraps up.
- **Test Case Development – (Dec 5 – Dec 15, 2025):** As development is in its final stages, the team will start writing detailed test cases. We aim to have the first draft of all test cases by mid-December. This overlaps with coding so that we can begin test execution on completed features without delay. Milestone: *Dec 15* – All test cases prepared and reviewed internally.
- **Unit & Component Testing (Ongoing, by Developers):** Throughout development (Nov – Dec 2025), developers perform unit tests on their code. By *Dec 15*, we expect all major code to be implemented and unit-tested, and perhaps minor tweaks based on any initial issues. This isn't a single date milestone, but it's understood that by mid-December the codebase is feature-complete and stable enough to begin formal integration testing.
- **Code Freeze & Integration Completion – (Dec 20, 2025):** Target date for code freeze – all planned features are implemented, and only bug fixes will be applied beyond this point. This also implies the front-end and back-end are fully integrated, and a stable build is deployed to the test environment. Milestone: *Dec 20* – **Code Freeze** declared; the test environment is fully set up with the latest build and ready for full testing.
- **System Test Execution (Full Testing Cycle) – (Dec 21 – Dec 30, 2025):** During this period, the team will execute the test cases on the code-frozen build. We will follow the test case document and mark each case as Pass/Fail. This period includes time to log defects and for the developers to fix them. We anticipate an iterative process:

test, find bugs, fix, and re-test. We aim to complete an initial pass through all test cases by around *Dec 27*. The remaining days up to Dec 30 are for re-testing fixes and performing regression tests.

- **Interim Checkpoint – Progress Demo (around Dec 25, 2025):** Around the Christmas date (or a date set by course for a progress demo or alpha release), we expect to demonstrate the application with core functionality to instructors or evaluators. By this point, critical functionality should be mostly working (even if some bugs remain). This demo serves as a checkpoint where we can gather feedback and also ensure that our testing is focusing on the right areas. It's not a separate testing milestone, but it's an external milestone that our testing supports (we want a stable demo, so we do a smoke test before the demo to ensure no major issues).
- **Test Cycle Completion – (Dec 31, 2025):** By end of December, we expect to have executed all tests and resolved the majority of defects. Milestone: *Dec 31* – All test cases have been run at least once, and all critical/major bugs have been addressed. Any remaining minor issues are documented. At this stage, we prepare the Test Summary Report and get ready for final sign-off.
- **Buffer for Contingency – (early Jan 2026):** If needed, the first few days of January are reserved for final re-testing, performance tuning, or last-minute fixes. We anticipate the final project submission or presentation will be in the first week of January 2026. By *Jan 5, 2026*, we plan to have the final build ready with testing fully completed and approval obtained. This is effectively the **Release Candidate sign-off** date.
- **Final Presentation/Submission – (Early January 2026):** The project (with test evidence) will be delivered. As a milestone, the team will present the project and the testing results to the instructor/committee. Testing-related artifacts (test cases, results, bug logs, etc.) will be submitted as part of the project documentation.

Throughout the testing schedule, we will hold short daily meetings or sync-ups during the intensive test execution phase (Dec 21–30) to track progress and adjust priorities if needed. If at any milestone, things are not as expected (e.g., more bugs than anticipated by code freeze), we will revisit the plan – possibly de-scoping some low-priority tests or features to meet deadlines (after discussion with the instructor).

This schedule is designed to ensure testing is not an afterthought but an integral part of the project timeline. By planning adequate time for testing and bug fixing, we improve our chances of delivering a high-quality platform on time.

11. Risks & Mitigation

Despite careful planning, certain risks could impact the testing process or the overall quality of the product. We have identified key risks and our mitigation plans for each:

- **Risk: Tight Timeline and Last-Minute Changes** – With development and testing happening in a short timeframe, there's a risk that not all features will be thoroughly tested, especially if development runs late or features change near the deadline. *Mitigation:* We have started test planning early and will prioritize testing of critical features first. If time becomes limited, we will focus on core user journeys (e.g. reading and uploading) and defer less critical test cases. We also instituted a code freeze date (Dec 20) to prevent continuous changes; after that, only fixes are allowed. This gives a stable period for testing. Regular check-ins will help catch slippage early, and if a feature isn't ready to test by code freeze, we might consider dropping that feature or marking it beta, in agreement with the instructor, rather than rushing incomplete functionality.
- **Risk: External Service Dependency Issues** – The platform relies on external APIs (OCR/translation, payment). These services might have limitations: for example, the translation API might enforce rate limits or could experience downtime, and the payment gateway might have unexpected integration quirks. If these occur during testing, they could block our test cases or give false failures. *Mitigation:* We will use sandbox modes and have backup plans for simulating the services. For instance, if the translation API has a strict quota, we will constrain how many translation tests we run in a short time, and reuse results when possible. We've also planned some tests to simulate the service being down, so if it actually goes down, we'll at least know how our system reacts. We will schedule payment-related tests at times when the sandbox is known to be stable (perhaps morning hours, avoiding any maintenance windows). We also have contacts (documentation/forums) for those services should integration issues arise, and we can adjust our approach (like reducing concurrency of translation requests) if needed.
- **Risk: Incomplete or Ambiguous Requirements** – If any requirement in the SRS was misunderstood or not clearly implemented, testers might find behavior that seems like a defect but is actually a gap in requirement clarity. This can lead to last-minute design changes or confusion. *Mitigation:* The team has been involved in the requirements and design phases, so testers are familiar with expected behavior. We will clarify any ambiguous expected results before test execution (for example, confirm with the group what the expected behavior is if two people comment at the same time, etc.). We have also defined expected outcomes in our test cases

upfront, reviewing them against the SRS and UI design, to minimize confusion. If a discrepancy is found (behavior vs. spec), the team will quickly decide whether it's a bug to fix or an acceptable change (with instructor's input if significant). Keeping communication open with the development team (which is ourselves) means we can resolve these ambiguities quickly.

- **Risk: Environment Setup and Data Issues** – Setting up the test environment with all its components (especially external integrations, correct config, and seeded data) could take longer than expected or encounter issues. If the environment isn't ready, testing will be delayed. Also, if test data is not well prepared, we might waste time during testing creating accounts or content on the fly. *Mitigation:* We are allocating time before formal test execution to configure the environment. The CI/CD pipeline helps here by automatically deploying builds. We will do a dry run of deployment and data seeding before Dec 20 to iron out any problems (for example, making sure the database schema on the test DB matches latest migrations, etc.). For test data, we will prepare scripts or SQL dumps to load initial data easily. During testing, if data needs to be reset (e.g., reloading the DB), we have a plan for how to do that quickly. Each tester will have admin credentials and any necessary accounts ready so we don't lose time setting that up mid-test. Essentially, planning and rehearsal of environment setup act as mitigation.
- **Risk: High Bug Rate or Critical Bugs Late** – If testing uncovers many defects, especially critical ones (e.g., crashes, data corruption issues), fixing them all could be challenging within the time, or fixes might introduce new bugs. The presence of a showstopper bug late in the schedule could jeopardize the release. *Mitigation:* Our approach of testing in parallel with development means we hope to catch some bugs early (not all at the end). We've prioritized essential features in testing order to find any critical bugs in those first. The development team is prepared to work in quick iteration with testers (immediate bug fixing and patch releases during the test cycle). We also maintain good version control, so if a late fix threatens to break other things, we can decide to roll back that particular change and maybe live with a known issue rather than destabilize the system right before deadline. Additionally, by categorizing bugs by severity, we'll focus on resolving all critical ones; if needed, we might leave a minor cosmetic bug unresolved in the final delivery (with documentation) rather than risk a risky fix under time pressure.
- **Risk: Team Availability and Coordination** – Since this is an academic project, issues like exam schedules, holidays (late Dec) or personal constraints might reduce the effective time team members can contribute to testing and bug fixing. If

a key member is unavailable at a crucial time, testing or fixing could stall. *Mitigation:* We have planned the schedule with some buffer (especially early January) to account for any small delays. The team has agreed on a period of focus for testing right after Christmas when we'll all dedicate time. In case one member is unavailable, tasks are documented so another can pick it up (e.g., test cases are written so anyone can execute them, not just the author). We also share credentials and knowledge among the team to avoid single points of failure (for instance, more than one person knows how to deploy the system or how the payment integration works). Regular meetings ensure everyone is aware of progress and can step in if needed.

Listing these risks helps us remain vigilant. We will review the risk list during test execution to see if any are materializing and address them proactively. By having mitigation strategies, we aim to reduce the likelihood of these issues derailing the project and ensure a successful testing phase.

12. Approval

This Test Plan has been prepared by the project team and is submitted for approval to proceed. Signatures below indicate that the plan has been reviewed and agreed upon by the relevant stakeholders:

- **Prepared by:** Le Anh Duy – Test Lead (Group 07)
- **Reviewed by:** Tran Gia Huy – Developer/Tester (Group 07)
- **Reviewed by:** Nguyen Thanh Luan – Developer/Tester (Group 07)
- **Approved by:** Mr. Truong Phuoc Loc – Instructor (Project Supervisor)

Approval Date: ____________________

Upon approval, this Test Plan becomes the guiding document for all testing activities of the Manga Auto-Translation Platform. Any changes to the scope or strategy outlined here will be discussed with and re-approved by the stakeholders. Once approved, the team will execute the plan as scheduled and update the stakeholders on the testing progress and results.