Project: Web-based Manga Platform with OCR and Translation
Course: Introduction to Software Engineering (HCMUS) – Instructor: Mr. Truong Phuoc Loc
Meeting Date: November 20, 2025
Meeting Phase: Software Design Planning Meeting

Meeting Details

- **Time:** 14:00 – 16:30
- **Location:** HCMUS Campus
- **Attendees:** All group members (Group 07)
    - Le Anh Duy – 23127011
    - Tran Gia Huy – 23127199
    - Nguyen Thanh Luan – 23127296
    - Nguyen Thanh Tien – 23127539

Purpose: To initiate the Software Design phase by reviewing finalized requirements and planning the system's architecture and design. This meeting focuses on translating requirements (from the SRS completed in November) into a concrete design strategy: deciding on system architecture, drafting major design artifacts (ERD, class diagram, UI layout), and assigning design tasks. The outcome will guide the creation of the Software Design Document and prepare the team for the Implementation phase.

Agenda

1. **Review of Requirements & SRS Highlights** – Ensure everyone understands the final requirements baseline (FRs, NFRs, use cases) as we move into design.
2. **System Architecture Design** – Decide on the overall architectural pattern and layering of the system (e.g., 3-tier architecture, client-server interactions, APIs).
3. **Database Design (ERD)** – Identify all database entities and relationships; start drafting the Entity-Relationship Diagram for the manga platform.
4. **Application Design & Class Diagram** – Plan the high-level design of the software, including key classes, their responsibilities, and relationships (covering front-end components and back-end classes).
5. **UI/UX Design Approach** – Outline the user interface structure and confirm which screens and user flows will be designed (based on prototypes), and the style guidelines (using Tailwind CSS components, etc.).

6. **External Integration Strategy** – Design considerations for integrating OCR/Translation API and Payment gateway into our system (e.g., design of service modules or adapters for these).
7. **Design Deliverables & Documentation** – List all outputs required for this phase (design diagrams, descriptions) and set internal deadlines.
8. **Task Assignments** – Assign team members to specific design tasks (database schema design, class diagram, UI design, etc.) and plan collaboration.
9. **Preparation for Implementation** – Discuss any tools or environment setup needed in advance for coding, based on the design decisions (e.g., deciding on frameworks, libraries, or version control structure if not already).
10. **Next Steps** – Plan the next meeting or review session and integration of design work into the formal design document.

Discussion and Notes

1. Review of Requirements & SRS Highlights

The meeting began with a quick recap of the final Software Requirements Specification (SRS) delivered at the end of November. Each member had read the SRS, so this was a high-level review to ensure all design decisions align with the stated requirements. Key points reaffirmed:

- **Scope Check:** The functional requirements and use cases in the SRS will drive our design. We confirmed that no new requirements have been added beyond what was discussed in the previous phase. Therefore, the design will concentrate on implementing exactly those features (account management, manga reading with translation, content upload/approval, payments, comments, etc.) and meeting the specified non-functional criteria (performance, security, etc.).
- **Use Case Review:** We revisited the main use case list to mentally walk through how the system needs to behave. For example, we traced the flow for "Reader views a translated page" and "Uploader submits a chapter and Admin approves it" to identify components needed (e.g., a translation service module, a content approval interface). This exercise ensured the team had a shared vision of what parts of the system we must design.
- **Requirements Questions:** One question was clarified: the earlier notion of a "Director" user (for viewing statistics) will indeed be handled as an admin feature.

So, when designing, we won't create a separate module for a new user type; instead, we'll include a reporting component accessible to Admins.

- **Constraints:** We reminded ourselves of important constraints from the SRS that impact design: for instance, supporting multiple browsers and responsive design (influences our frontend approach with Tailwind CSS), and integrating third-party APIs (influences how we design our backend modules and external service interfaces).

With everyone on the same page regarding what needs to be built, we moved on to planning the system architecture.

2. System Architecture Design

The team discussed and finalized the **architecture** of the system. We agreed on adopting a **three-tier architectural pattern** which was suggested in our proposal:

- **Presentation Layer (Frontend)** – This will be the Next.js web application running in the browser, responsible for the user interface and client-side logic. It will include pages and components for the reader, uploader, and admin views. The frontend communicates with the backend via HTTP/HTTPS requests (RESTful API calls).
- **Application Logic Layer (Backend Services)** – This is the Node.js/Express server which contains the business logic and API endpoints. It will handle tasks such as authentication, handling form submissions (uploads), processing requests for translations (by calling external APIs), enforcing business rules (like ensuring only admins can approve content), etc. We decided on a modular service structure on the backend:
    - **Auth Service** (manages login, registration, JWT token issuance, etc.)
    - **Manga Service** (handles manga and chapter CRUD operations, reading progress, etc.)
    - **Translation Service** (module that interacts with the OCR/translation API – likely implemented as an external API integration wrapper)
    - **Payment Service** (handles payment transactions and verifying purchase access)
    - **User Management Service** (handles user roles, profile management, etc.) We will use controllers in Express to expose these services via endpoints (e.g., POST /api/uploadChapter, GET /api/getTranslations, etc.), following a REST design.

- **Data Layer (Database and External Services)** – This includes the PostgreSQL database which will store all persistent data (user accounts, manga metadata, chapters info, comments, transactions, etc.), and also any external services the system relies on:
    - The **OCR & Translation API** (external): Our system will act as a client to this service. We determined that when a user requests a translation, the backend will send the image to this API and get text results. The design must account for network latency and possible failures (so maybe implement a retry or asynchronous processing if needed). We will likely call a cloud API (e.g., Google Cloud Vision + Translate) and not host our own model.
    - The **Payment gateway** (external): For payments, we might simulate or integrate with a sandbox of a payment service (e.g., a fake payment processor or something like PayPal sandbox or MoMo/ZaloPay sandbox). In design, we will include a Payment API integration point in our architecture.
    - **Cloud Storage/CDN** (optional external): We discussed that storing images (manga pages) might require a storage solution. For now, we plan to store images on disk or a simple storage since the scale is small, but we keep in mind that in a real system, images might go to a cloud storage (like Cloudinary or S3). We won't detail this in our design beyond noting where images are stored.

This 3-layer separation adheres to best practices and was explicitly intended in our development plan. It allows the front-end and back-end to be developed and scaled independently. We sketched a high-level **architecture diagram** to visualize these components and their interactions. The diagram shows the user's browser (client) communicating with an Express.js server, which in turn connects to the PostgreSQL database and external APIs (OCR/Translation, Payment). We also indicated the separation of the front-end into public pages (for guests) and protected pages (for logged-in users with roles), as well as separate admin routes.

Additionally, we deliberated on **design patterns** to use within this architecture:

- On the client side, we will use Next.js's page/router structure and possibly React context or Redux for state management of user data (like login state, cart for purchases, etc.).
- On the server side, we will follow an MVC-ish structure: Express routers/controllers for each module (Auth, Manga, User, etc.), service classes for business logic, and data access objects (or use an ORM like Prisma/Knex) for database interactions.

This layered approach improves maintainability. We will also incorporate authentication middleware (for verifying JWTs on protected routes) and maybe use design patterns such as Repository (for data access abstraction) or Adapter (for the translation API integration).

- For the translation integration, the **Adapter pattern** was suggested: we create our own interface in the backend (e.g., a function translatePage(image) in our TranslationService). Under the hood it calls the external API. This way, if the API changes or if we switch providers, the rest of the system is not affected – we just update that adapter.

The team unanimously agreed on this architecture approach. It aligns with our initial plan and satisfies the requirement of separating concerns and supporting scalability.

3. Database Design (ERD)

We moved on to the database design, a crucial part of the system's foundation. Using the list of data entities gleaned from requirements, we identified the following primary entities (tables) to include in our **Entity-Relationship Diagram (ERD)**:

- **Users** – stores user account information (unique ID, username/email, password hash, role, etc.). We will have a field for role or a separate related table for roles if needed (depending on if a user can have multiple roles; for simplicity, we might use a single role attribute with values like "Reader", "Uploader", "Admin").
- **Manga** – represents a manga series or title. Contains fields like mangaID, title, author, genres, description/summary, cover image, etc. (Our project might not need a separate series vs chapter distinction if we treat each upload as a "chapter" linked to a manga title. We assume a manga can have multiple chapters.)
- **Chapters** – each chapter belongs to a Manga. Fields: chapterID, title, chapter number, language, upload date, status (pending/approved), uploaderID (who uploaded it). Approved chapters are visible to readers. We might combine Manga and Chapter into one if one upload = one chapter and treat "title" as part of chapter, but for clarity, we keep them separate so that multiple chapters can group under one manga title.
- **Pages** – this table will store references to each page image for each chapter. Fields: pageID, chapterID (foreign key), page number, image URL/path. This ensures we can retrieve all pages of a chapter in order. Alternatively, we could store page images as files named in order, but a table gives flexibility (and could store extracted text or translation if we wanted to cache it).

- **Comments** – stores user comments on chapters. Fields: commentID, chapterID, userID, content, timestamp. Possibly a rating field if we combine rating with comments or separate table for ratings. We decided to include a rating within comment or allow a comment to have a rating score, to simplify (one comment per user per chapter might carry a rating).
- **Favorites** – a join table between Users and Manga (or Chapters) to record which titles a user has favorited.
- **ReadingHistory** – to track reading progress: userID, chapterID (or mangaID), last page read, etc.. This can help resume reading and also possibly for analytics (but primarily for user convenience).
- **UploaderRequests** – since readers can request to become uploaders, we might have a table for those requests (userID, status of request, requested date, approved/rejected date). Alternatively, a simpler approach is just having a boolean field in Users for "isUploader" and when a user requests, set a flag and let admin manage it. We will likely implement the latter (e.g., user role field updated to "Pending Uploader" state) to reduce complexity. Given time, we noted this but left exact implementation detail flexible.
- **Transactions/Purchases** – to log payments. Fields might include transactionID, userID, chapterID (or a purchase pack ID), amount, date, status. This ensures that if a user has purchased a chapter, we know to grant them access. We might call this table **PurchasedChapters** linking users and chapters that are bought, and another table for **DepositTransactions** if users load credits or a wallet (the design doc index shows something about coin history, deposit transactions, suggesting the team had the idea of an in-app currency). In our discussion, we simplified: perhaps readers pay per chapter or buy coins; we decided on direct pay-per-chapter to not overcomplicate. So a PurchasedChapters table could suffice to track access rights.
- **AdminActions/Logs** – optional, to record admin activities (like approvals, suspensions). We acknowledged it but decided it might be beyond our immediate scope. Instead, the status fields (like chapter status, user status) can carry enough info, and we won't design a whole audit log unless time permits.
- **Role/Permissions** – since roles are few and mostly fixed (and a user has one role at a time), we might not need a separate roles table; roles can be an enum (e.g., 0=Reader, 1=Uploader, 2=Admin). For clarity in design docs, though, we can include a **Roles** table referencing user roles.

We sketched an ERD on the whiteboard with these entities and drew relationships:

- One User can upload many Chapters (one-to-many, User → Chapters).
- One Manga can have many Chapters; a Chapter belongs to one Manga (one-to-many, Manga → Chapters).
- A Chapter has many Pages (one-to-many, Chapter → Pages).
- A User can have many Comments, and a Comment is by one User on one Chapter (so Comments links User and Chapter in a many-to-one on each side).
- A User can favorite many Manga, and a Manga can be favorited by many users (many-to-many via Favorites).
- A User can purchase many Chapters (many-to-many via PurchasedChapters, or via a Transaction log).
- If using UploaderRequests table, it would link to User (one-to-one or one-to-many if a user could re-request after rejection).
- The relationships were straightforward and we ensured to include all required data for our features.

Le Anh Duy started drafting a formal ERD diagram, which will be polished after the meeting. We cited some table name references from the design document outline to double-check we hadn't missed an entity (the design template listed things like genres table, otp_codes table, etc., but those might be beyond scope). In our case, we decided:

- **Genre**: We will have a simple approach for genres (possibly a comma-separated field in Manga or a separate Genre table and a relationship table MangaGenre). For now, we included a Genre table for normalization (with Manga-Genre join table if needed).
- **OTP codes**: That appears in the template index (likely for two-factor auth), but we decided two-factor authentication is out of scope, so we are not including OTP in our design.

The database will be implemented in PostgreSQL, and we will use an ORM or query builder for development ease. The design will include a **Database Schema document** listing all tables, columns, data types, and relations.

4. Application Design & Class Diagram

We proceeded to outline the application's internal design. This involved planning key classes and modules for both the frontend and backend, and how they relate, which we will

capture in a **Class Diagram** for the backend and possibly a component diagram or similar for the frontend.

On the **backend side (Node/Express)**:

- We identified classes (or conceptual classes) corresponding to major entities and services:
    - **User** class – representing user accounts; with methods related to authentication (or we might use built-in libraries for auth, but conceptually a User entity with methods to check password, etc.).
    - **Manga** class – representing a manga title or series. Might not have complex behavior itself, but exists as a data model.
    - **Chapter** class – representing a chapter; could include behaviors like a method to retrieve its pages or to initiate an OCR process on its pages (though OCR likely handled by a separate service class).
    - **Comment** class – data model for comments.
    - (Additionally, classes for Genre, Favorite, etc., mostly as data containers or ORM models.)
    - **TranslationService** class – to encapsulate communication with the external translation API. For example, a method translateImage(pageImage) that returns text, used by a Chapter controller when user requests translation.
    - **PaymentService** class – to handle payment processing or simulate it. E.g., a method processPayment(user, chapter) that interacts with an external API or marks a chapter as purchased.
    - **AuthService** – to handle login, logout, JWT token creation, etc..
    - **MangaService/ChapterService** – containing core logic for managing manga and chapters (e.g., saving a new chapter, retrieving approved chapters list, etc.). This might use the data layer (ORM models) to perform operations and enforce rules (like "if chapter.status = pending and user.role = Admin, then allow approval").
    - **AdminService** – possibly containing admin-specific operations (approve chapter, feature manga, manage users).

- **Controller classes** (as needed, though in Express they might just be route handler functions): e.g., AuthController, MangaController, AdminController that route HTTP requests to the service methods.

We outlined a tentative **Class Diagram** focusing on core relationships:

- The User, Manga, Chapter, Comment, etc., as entity classes (or models) with their attributes.
- Services linking to these (composition or usage relationships). For example, MangaService uses Manga and Chapter models; AuthService uses User model.
- Inheritance where applicable: We considered if classes like Reader, Uploader, Admin should inherit from User (for example, Admin could be subclass of User with extra privileges). However, since roles are mostly data-driven (permissions checks), we may not need separate subclasses in code. The design document template hinted at separate classes for Reader, Uploader, Admin, Director. This could be a design choice: making User a base class and specialized roles as subclasses. We discussed it:
  - **Option 1:** Single User class with a role field; methods check role to allow certain actions.
  - **Option 2:** Use inheritance, e.g., class Admin extends User, class Uploader extends User, each possibly with additional methods.
    We leaned towards **Option 1 (single User class)** for simplicity in implementation. The design doc's listing of classes (Reader, Uploader, Admin, Director) likely was to separate concerns for documentation, but implementing separate classes might overcomplicate the code. We will mention those classes in the design document conceptually (to discuss role-specific behaviors), but in code, they may not be separate objects. For example, an Admin can approveManga(), which could be a method available when user.role == Admin rather than an Admin class method. We took note to justify this decision in the design doc, but also to show awareness, we might still include a section describing role-specific permissions instead of literal subclasses.
- We also have to consider the **frontend components**. While class diagrams usually focus on backend, we decided to include an overview of the front-end structure: Next.js will have pages like HomePage, MangaDetailPage, ReaderPage, UploadPage, AdminDashboardPage, etc. These aren't "classes" in the classical sense but React

components. We might not detail them in a UML diagram, but we will describe them in the design document's UI/UX section.

We then assigned Nguyen Thanh Luan to draft the class diagram encompassing the above, ensuring to include relationships like:

- Composition between Manga and Chapter (a Manga has chapters),
- Association between User and Chapter (User as uploader of Chapter, User as author of Comment, etc.),
- The TranslationService and PaymentService as separate classes that might be used by Chapter or Admin services,
- Possibly depicting the pattern of controllers calling services, etc.

5. UI/UX Design Approach

The discussion moved to the front-end design and user experience. We reviewed the low-fidelity prototypes that Le Anh Duy had created during the Requirements phase. Those served as a starting point for the UI design:

- We confirmed the **overall look & feel** will follow modern web design principles: a clean layout, easy navigation menu, and responsive design for mobile compatibility. Since we use Tailwind CSS, we can rapidly style components to be responsive. We will stick to a simple, intuitive color scheme suitable for a manga site (perhaps dark text on light background for reading, with some accent colors for menus/buttons).
- **Screens to design in detail:** We enumerated which screens need complete wireframes and later hi-fidelity design:
    - **Home Page:** Lists of manga (maybe categorized by newest, most popular, featured). Search bar at top. Possibly banner for featured manga.
    - **Discover/Search Results Page:** A page to show search results or manga by category.
    - **Manga Detail Page:** Shows info about a manga title (cover image, description, list of chapters available). If a chapter is paid, show a lock icon or purchase button.
    - **Reader Page (Chapter Viewer):** Displays the manga pages. Include UI controls for translation (toggle original/translated text), navigation to next/prev page/chapter, zoom, etc. Also a comments section below the pages for that chapter.

- **Login/Register Modal or Page:** For user authentication.
- **User Profile/My List Page:** Where user can see their favorites ("My List"), reading history, maybe account settings.
- **Uploader Dashboard:** Page for uploaders to manage their uploads. Lists chapters they've uploaded and their status (approved/pending).
- **Upload Chapter Page:** Form for uploaders to add a new chapter (fields: title, choose manga or new manga, genre tags, summary, upload images).
- **Admin Dashboard:** Overview of admin functions – could have tabs for "Pending Approvals" (list of chapters awaiting review), "User Management" (list of users with ability to upgrade or lock accounts), and maybe "Reports" (site stats).
- **Chapter Approval Page (Admin):** A screen or modal where admin reviews a specific chapter (shows pages or info) and can approve or reject with a comment.
- **Statistics/Report Page:** If we implement the director requirements, an admin page showing usage statistics (we might do simple stats since our data will be limited).

Le Anh Duy presented updated wireframes for a few of these during the meeting:

- The **Reader Page wireframe** showing a manga page with an overlay "Translate" button and a language dropdown. Once translated, the text would appear either overlaid or in a sidebar. We need to decide UI-wise: likely, we'll overlay translated text bubbles or show the translated text below the image for simplicity. We will refine this in implementation; design-wise, we note the need for a toggle between original and translated view.
- The **Upload Page wireframe** with fields and an "Upload Images" section (drag-and-drop area for image files).
- The **Admin Pending Approvals** wireframe with a list of pending chapters and an action to view details.

We discussed consistency: using common navigation across pages (e.g., top nav bar with site logo, search, profile menu). Also ensuring that certain pages are role-restricted (Upload and Uploader Dashboard accessible only to Uploaders, Admin pages only to Admins). We'll handle that in frontend routing (Next.js can check user role and redirect if unauthorized).

For **UX**, we emphasized:

- The site should be intuitive for readers who may not be tech-savvy – simple buttons, clear labels (e.g., "Translate to English").
- Provide feedback on actions (e.g., "Your chapter has been submitted for approval" message for uploaders).
- Use modals or confirmation dialogs for critical actions (like admin approving/rejecting content).
- Because we have a global audience in concept (reading manga from around the world), the UI is English for our project, but we allow content translation. We noted the possibility of future i18n (interface in multiple languages) but agreed it's beyond our current scope. We will design in English for now.

Tailwind CSS will guide much of our design implementation, but at this stage we're creating static design artifacts: we will include **UI wireframe diagrams** or mockups in the design document as part of deliverables.

6. External Integration Strategy

We dedicated part of the meeting to how the design will incorporate external services (OCR/translation and payment). This was partly covered in architecture but we delved into design specifics:

- **OCR/Translation API Integration:** We will design a module (TranslationService class as mentioned) that abstracts the details. The design should specify:
    - Input/Output: it likely takes an image or image URL and returns the extracted text or translated text. We might perform OCR and translation in one API call if possible, or first OCR to get text then translate that text. We need to research the API's capabilities, but for design, we assume a single service can do both (some services might, or we combine two services).
    - Error handling: If the API call fails or times out, how our system responds (perhaps show an error message to user "Translation currently unavailable").
    - Performance: We won't cache translations in this phase design, but we note that caching could be beneficial (if multiple users request translation of the same page and the content doesn't change). As a design consideration, we could store translated text results in the database (maybe a field in Page or a separate table for TranslatedText keyed by pageID and language). This is

optional – we decided if time allows, we might implement caching of translation results to improve performance.

    - The design doc will include a sequence diagram or narrative for "Translate page" use case: illustrating the user clicking translate, the request going to backend, calling the external API, and returning the result to the frontend.

- **Payment Integration:** We need to design how the user will pay for content:
    - Possibly a simple checkout flow: user clicks "Purchase Chapter 5" → system calls PaymentService which interacts with an external gateway or simulates it, then on success marks that user now owns chapter 5.
    - We might implement a dummy credit card form or redirect to a sandbox payment page. Design-wise, we include a "Payment" module that handles this. It might either connect to a third-party payment API or simply simulate a transaction (since real payment integration can be complex). In our design, we lean towards simulating (e.g., assume success for the sake of demo), but we outline how it would work with a real API for completeness.
    - The **PurchasedChapters** table (or similar) design covers how we record access. The front-end design will show a lock icon on chapters that require purchase and an unlocked icon or label on those the user has bought. After purchase, the UI should update to allow reading the chapter.
    - We also touched on possibly a "coin" system (some team members had earlier ideas of user buying coins to spend). But given time constraints, we simplified to direct purchase per chapter. This simplification will be reflected in design (e.g., no need for a full wallet management UI; just a purchase confirmation dialog).
- **Email/Notifications:** Not strictly an external integration, but related – if time, we considered notifying users via email on certain events (e.g., upload approved). This requires an email service integration (like sending emails). We decided this is out of scope for now and will not be included in our design to avoid adding complexity.

By designing these integration points now, we clarified what components we need to include in our class diagram and architecture. For example, our PaymentService might be a stub class for now but placed in design to show where payment logic would reside.

7. Design Deliverables & Documentation

We listed all the expected deliverables for the Software Design phase to make sure our work covers everything required:

- **Architecture Diagram & Explanation:** A diagram illustrating the system's high-level architecture (as discussed, likely a block diagram of client, server, database, external services) along with text describing how data flows through the system.
- **Entity-Relationship Diagram (ERD):** Visual diagram of the database schema with tables and relationships (which Le Anh Duy is drafting). This will be accompanied by a **Data Dictionary** (table-by-table description of fields, keys, etc.) in the design document.
- **Class Diagram:** Showing the main classes, their attributes and methods, and relationships (association, inheritance if any, etc.). This will focus on the backend classes (and possibly include frontend components at a high level for completeness).
- **Sequence Diagrams (or flow diagrams) for key scenarios:** We plan to include at least one sequence diagram, for example:
    - *User reads and translates a page* – to illustrate interaction between front-end, backend, and external API.
    - *Uploader submits chapter and Admin approves* – to illustrate the workflow and how different components interact (uploader's browser → server → database, then admin's browser → server, etc.).
- **UI/UX Designs:** This includes wireframes or mockups of important screens (which Nguyen Thanh Tien will refine). We might also create a **Screen Flow diagram** showing navigation between screens (the template mentions "Screen Diagram" for user and admin subsystems). We decided to have a diagram that shows how screens are interconnected (e.g., from Home Page you can go to Manga Detail, then to Reader, etc., and how admin screens are separate).
- **Design Decisions & Rationale:** A section in the document will detail decisions we made (e.g., why we chose 3-tier architecture, why using inheritance or not for user roles, why choosing certain data models). Many of these we noted during this meeting to be written down.
- **Updated Schedule & Plan for Implementation:** While more of project management, we will include how the design phase transitions to implementation, confirming timelines. (Our Gantt chart indicates the design phase should conclude

by mid-December so that implementation can start and be in full swing by late December.)

We noted that the **Software Design Document** is due by mid-December (tentatively December 15, 2025). We are aiming to finish these deliverables a few days in advance for review.

8. Task Assignments

After fleshing out what needs to be done, we assigned tasks to team members for the design phase deliverables:

- **Le Anh Duy:** Responsible for the **Database Design**. Duy will finalize the ERD diagram and write the accompanying description of each entity (including any assumptions about data types or constraints). Also, Duy will contribute to the architecture diagram focusing on the data layer integration. *Target Completion:* Draft ERD by Dec 5 for team review.
- **Tran Gia Huy:** Responsible for the **Application Architecture and Class Diagram**. Huy will create the high-level architecture diagram and lead drafting the class diagram, depicting the system's modules, classes, and their interactions. Huy will also write about the chosen architecture and major patterns in the design document. *Target:* Architecture & class diagrams draft by Dec 7.
- **Nguyen Thanh Luan:** Will work on **Detailed Component Design and Sequence Diagrams**. Luan will take charge of writing the sections of the document that describe each major component's design (e.g., how the translation feature is designed, how the payment process is designed) and draw sequence diagrams for the key use case flows. Also, Luan will document any design patterns utilized in those flows. *Target:* Sequence diagrams and write-ups by Dec 10.
- **Nguyen Thanh Tien:** Responsible for the **UI/UX Design artifacts**. Tien will refine the wireframes for all important screens, ensuring consistency in style and adherence to requirements. Tien will compile these into the design document (possibly as figures) and write a brief UI/UX section explaining the design decisions (layout, navigation, responsiveness). *Target:* Completed set of wireframes by Dec 8.
- **All Members:** Once individual parts are drafted, the team will convene to integrate them into the final Software Design Document. We set an internal deadline of **December 12, 2025** to have a full draft ready for proofreading, leaving a couple of days to refine it before submission on Dec 15. Everyone will review each other's

sections to ensure consistency (e.g., class names in the class diagram match those referenced in the sequence diagrams and ERD, etc.).

Additionally, since the implementation phase overlaps slightly, we agreed that while doing design, we can begin some preparatory work (like setting up the Git repository, installing necessary frameworks) in parallel, as long as it doesn't distract from completing the design deliverables.

9. Preparation for Implementation

Before closing, we touched on preparation steps for the upcoming Implementation phase, given it's around the corner:

- **Development Environment Setup:** We checked that all members have the required tools ready: Node.js 18+, a PostgreSQL environment (we might use a cloud database like Supabase or a local instance for development). Since we plan to deploy the database on a cloud service (per our plan), we will design with connection strings that can be easily switched from development (local or cloud dev DB) to production (if any).
- We will use **GitHub** for version control (the repository was created after the last phase). We discussed adopting a branch strategy (each feature or module on its own branch, with pull requests for merging to main) and enforcing code reviews by at least one other member. This wasn't a formal part of design, but an important team agreement.
- **Frameworks/Libraries confirmation:** We confirmed we will use an ORM (most likely Prisma or Sequelize for Node) to implement the database layer as it aligns with our design and will speed up development. Also, we mentioned using a testing library (like Jest) for unit tests later – this influences how we structure code but is more relevant in implementation planning.
- **Coding Standards:** The team decided to follow common styling guidelines (Prettier for code formatting, ESLint for linting in the project) to ensure code consistency. This is a minor note but will be helpful once multiple people start coding simultaneously.

This part of the discussion was not a formal design topic, but we included it in the minutes as it was discussed and helps ensure a smooth transition to implementation once design is done.

Decisions Made

- **Three-Tier Architecture Adopted:** We formally decided on a three-layer architecture (Presentation, Application, Data) for the system, as it fits our needs for separation of concerns and scalability. This architecture will be depicted in the design document and followed during implementation.
- **Use of Established Frameworks/Patterns:** The design will leverage known frameworks (Next.js, Express) and we will implement design patterns (like MVC, Adapter for external services) to organize the codebase. We decided not to deviate into any experimental or overly complex architecture given our timeframe – sticking to proven approaches is preferred.
- **Consolidation of User Roles in Design:** Rather than creating separate subclasses for Reader, Uploader, Admin in our code, we decided to use a single User model with role attributes. This decision keeps the design simpler and aligns with how user roles are often handled (via permissions). In the documentation, we will still acknowledge role-specific behavior but clarify it will be implemented via conditional logic or role checks, not separate class hierarchies.
- **All core database entities finalized:** The meeting resulted in agreeing on the set of database tables (as listed in the ERD discussion). Notably, we included tables for user actions (favorites, comments, purchases) to fully support requirements. The decision to include a separate Genre table (for extensibility) was made, and to drop possible tables we won't use (like OTP codes).
- **External Services Integration Approach:** Decided that for translation, we will use a cloud API and not store translations persistently (unless we add caching as an enhancement). For payments, we will simulate transactions. These choices will be documented as design decisions and mean less overhead in implementation (but still fulfilling the requirement of demonstrating the feature).
- **UI Design Consistency:** Chose to maintain a consistent UI framework for all subsystems: meaning the admin interface will be web-based and integrated into the same frontend app (just hidden behind admin login). We will not create a separate admin application. Admin pages will reuse components wherever possible (e.g., listing chapters might use similar components for user's view and admin's view but with extra controls). This decision simplifies design and development (one unified frontend).
- **Timeline Commitment:** The team reaffirmed the deadline for the design phase deliverables (Dec 15, 2025) and committed to the task assignments listed. We also

decided on a mid-phase check-in (Dec 7, 2025) to verify progress on diagrams and sections, which is effectively another short meeting or async review.

Action Items Summary

- **Nguyen Thanh Tien:** Complete the ERD and database schema description. Ensure all required fields and relationships to support the requirements (e.g., favorites, purchases) are included. *Due:* 01/12/2025.
- **Tran Gia Huy:** Produce the system architecture diagram and class diagram. Document the rationale behind the chosen architecture and any patterns. *Due:* 01/12/2025.
- **Nguyen Thanh Luan:** Create sequence diagrams for at least two key scenarios (User reads & translates manga, Uploader & Admin content approval flow). Write descriptions for the translation and payment integration design. *Due:* 1/12/2025.
- **Le Anh Duy:** Develop high-fidelity wireframes for the main UI screens and compile a UI design section (with screen flow diagram). *Due:* 01/12/2025.
- **All Members:** Integrate all design artifacts into the Software Design Document. Meet on 27/11/2025 for a thorough review session. Incorporate any feedback from that review and finalize the document for submission by 01/12/2025.
- **Preparation for Implementation:** Before the next phase meeting, each member will ensure their development environment is set up (Node, Next.js, database connection). Also, we will collaboratively set up the GitHub repository with proper access for all and create an initial project structure.
- **Next Meeting:** The **Implementation Phase Planning Meeting** is tentatively scheduled for **December 4, 2025** (immediately after design phase completion) to plan the coding sprint in detail. This will be confirmed as we approach that date.