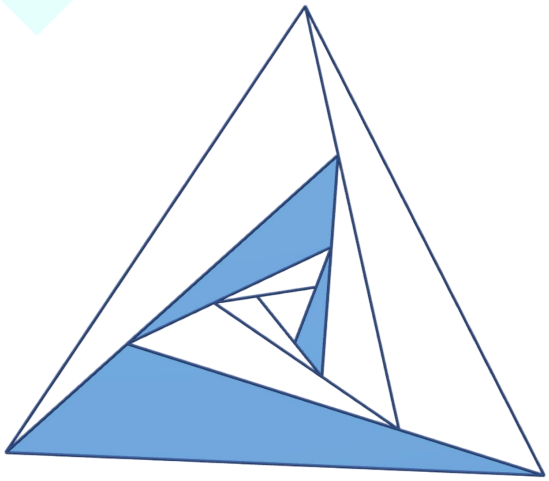

Delta3D

An app created for 3D modeling

Software Design Document

Smart Mobile Application Development (2025-2026) Group 10

Contents

1. Introduction

1.1 Purpose

This Software Design Document (SDD) provides a comprehensive guide for the development and implementation of the **Delta3D mobile application**, a 3D model sharing and community interaction platform for Android. The purpose of this document is to define the software architecture, functionalities, and components of the Delta3D app. It is intended for developers, project managers, and testers involved in the development, implementation, and maintenance of the application. This document serves as the blueprint to ensure a clear understanding of the software system's design, ensuring its efficient and effective development process.

1.2 Scope

The Delta3D mobile application is designed to provide an accessible, efficient, and creative solution for users interested in 3D modeling and content sharing. The application will offer features such as user registration, model browsing, 3D preview, content sharing, community interaction, messaging, and personal profile management. The key functionalities of the Delta3D app include:

- User registration and login
- Model browsing and search by tags
- 3D model preview with zoom, rotation, and interaction support
- Community features including posts, comments, and user interaction
- Messaging for private communication
- Model sharing and export to social media platforms
- Personal center for managing user profiles and collections

The system is designed to be scalable, supporting multiple user types ranging from casual content creators to professional users. However, the application will not include features related to advanced 3D modeling editing, which may be incorporated in future versions.

2. System Overview

2.1 System Features

The **Delta3D mobile application** is designed to offer users a comprehensive platform for creating, sharing, and interacting with 3D models in a social environment. The application provides several key functionalities that enhance the user experience:

The app supports **user registration and login**, where users can create accounts and securely authenticate through token-based systems. Once logged in, users can explore a vast library of **3D models** categorized by tags. A robust **search feature** enables quick access to models, posts, users, and tags, allowing for easy navigation and content discovery.

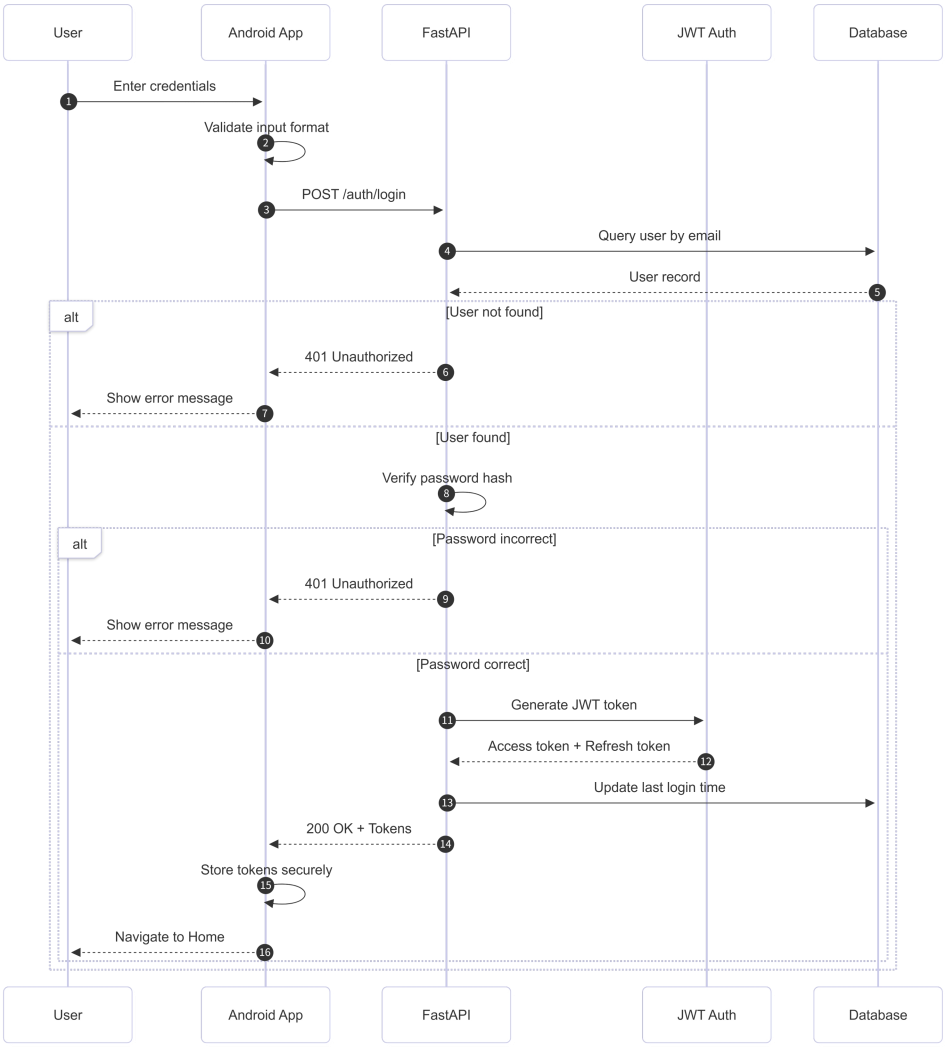

Figure2-1, Android App Login Sequence Diagram (FastAPI + JWT)

One of the standout features of **Delta3D** is the **3D model preview**. Users can interact with models in a 3D environment, manipulating them through features like zoom, rotation, and pan. This immersive interaction ensures an engaging and dynamic

experience for users.

In addition to browsing models, the app provides a **community interaction** platform where users can share their models, comment on others' work, like posts, and follow other users. **Real-time messaging** capabilities, powered by WebSocket, facilitate instant communication between users, creating a lively social space within the app.

Sharing is made simple through various **sharing options**, where users can generate links to their models or share them directly to social media platforms. The app also includes **user profile management** tools, allowing users to personalize their experience by managing their information, model collections, and social interactions.

2.2 System Architecture

The **Delta3D** system is structured around a layered architecture that divides responsibilities into clear, modular components, ensuring flexibility, scalability, and maintainability.

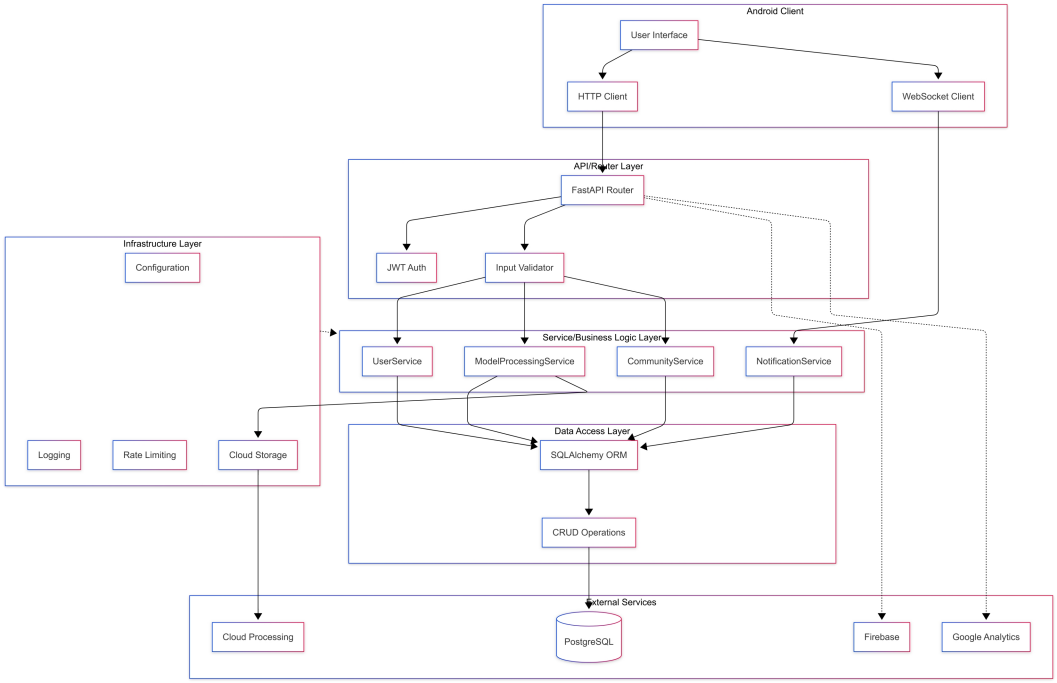

Figure2-2, System Architecture Diagram

At the top of the system is the **API/Router Layer**, which is responsible for managing incoming HTTP requests. This layer validates input, routes requests to appropriate services, and returns responses to the user. **FastAPI** is used to handle the HTTP operations, providing high performance and security. It also ensures secure communication through **JWT** authentication and authorization.

The **Service/Business Logic Layer** encapsulates core business functionalities. This layer processes requests like **model uploads, user management**, and **real-time**

interactions. Core services such as **ModelProcessingService** handle tasks like converting uploaded files into usable 3D models, while the **UserService** manages the registration, login, and profile functionalities. Additionally, **NotificationService** manages real-time user interactions, such as chat messages and activity alerts.

Data access is managed by the **Data Access Layer**, which abstracts the interaction with the database. Using **SQLAlchemy ORM**, it handles the creation, retrieval, updating, and deletion (CRUD) of data. This separation ensures that the business logic is independent of the underlying database operations, making the system easier to maintain.

The **Infrastructure Layer** forms the backbone of the system, providing necessary configurations, security, and external integrations. It is responsible for key infrastructure elements such as **configuration management, logging, API rate limiting**, and **cloud storage management**.

This architecture ensures that different components of the system remain modular, scalable, and easy to extend. Each layer interacts with the one directly below it, adhering to the principle of separation of concerns.

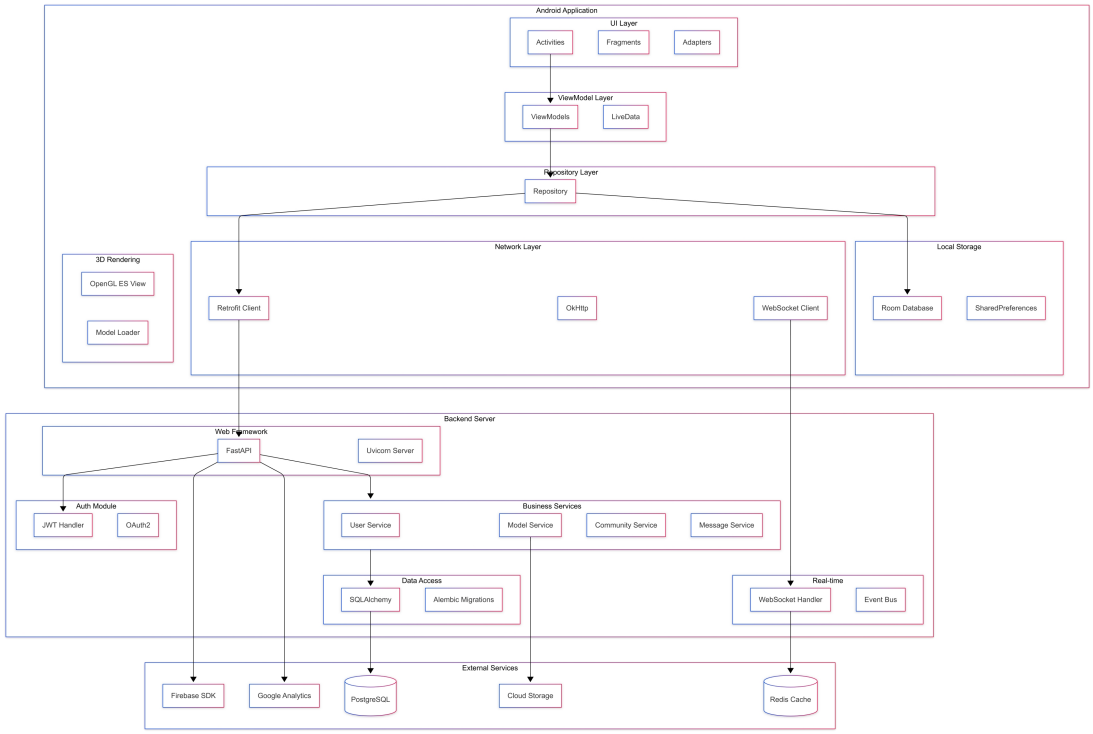

Figure2-3, End-to-End Mobile Application Design

2.3 External Interfaces

The **Delta3D** system interacts with several external components, allowing it to

provide a rich, integrated user experience. The app interfaces with **mobile devices (Android)** through RESTful APIs for user authentication, model uploads, and real-time messaging. The **WebSocket protocol** is used for real-time communication, ensuring that messages are delivered instantly between users, making the chat system highly interactive.

In addition to mobile devices, **Delta3D** also relies on **cloud services** for storing and processing 3D models. When users upload content, it is sent to the cloud, processed, and then the generated models are returned to the app for preview and sharing. This interaction with cloud services is essential for the **computational heavy lifting** required to render and handle complex 3D models efficiently.

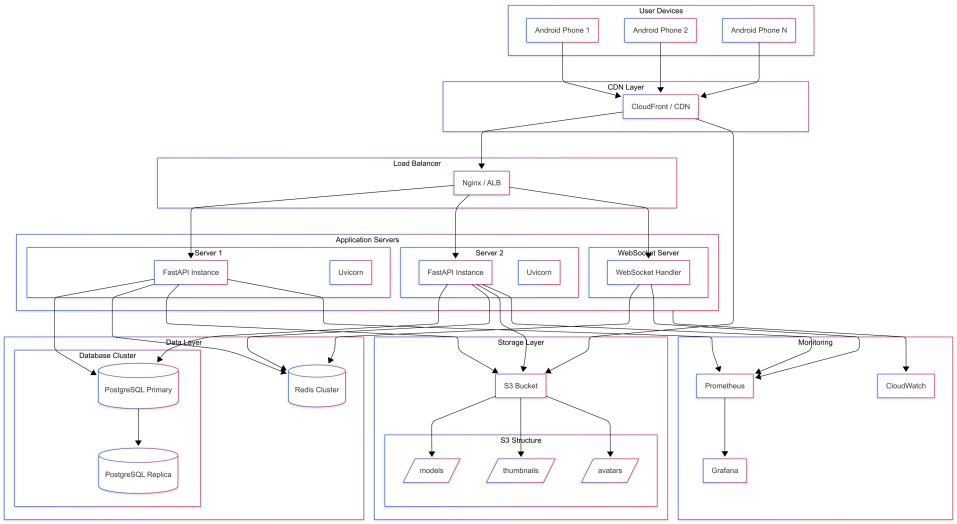

Figure2-4, Logical Layered Architecture Diagram

The system also integrates with **external APIs** such as **Firebase** for crash reporting, performance monitoring, and **Google Analytics** for tracking user interactions. These integrations help the team to monitor app performance and make data-driven decisions for improvements.

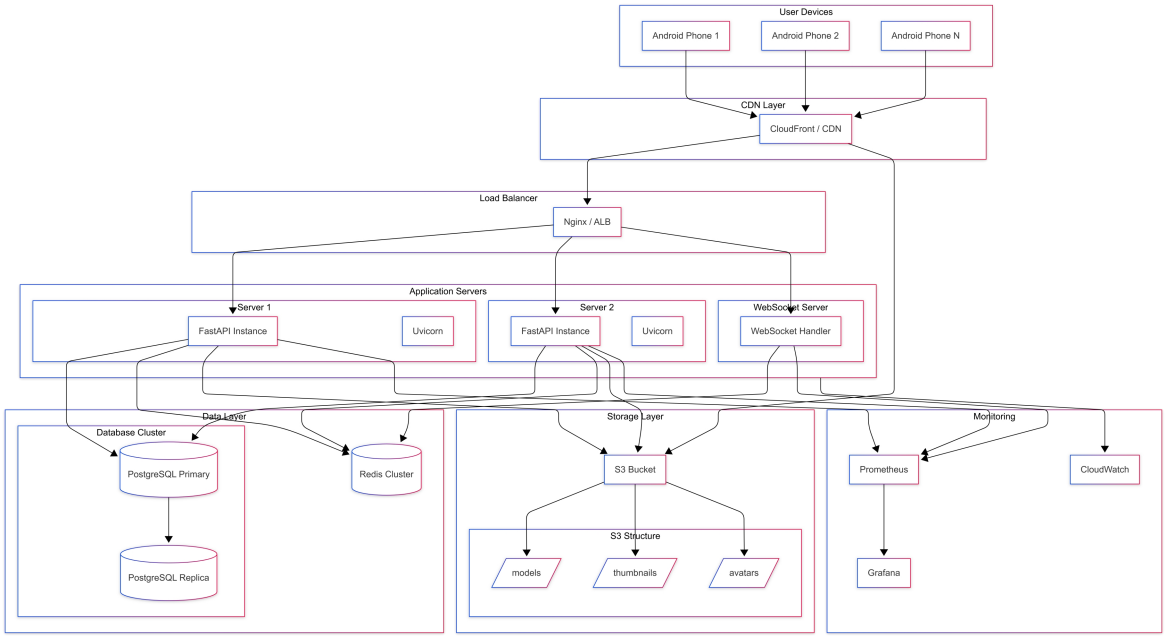

Figure2-5, Scalable Deployment Architecture

2.4 Performance Requirements

To ensure a seamless and engaging user experience, **Delta3D** must meet specific performance benchmarks. **Response times** for key user interactions, such as searching for models or loading the 3D preview, should not exceed **2 seconds**. Real-time messaging should have near-instant response times to avoid delays in communication.

The system must be able to handle **at least 100 concurrent users**, supporting real-time messaging, model browsing, and uploads without noticeable performance degradation. The backend infrastructure should be **scalable**, capable of handling increased traffic, especially during peak times, such as high-volume model uploads or community interactions.

Mobile performance is also a key consideration. The app should not exceed **300 MB** of memory usage under typical use and must be optimized for **low power consumption** to ensure extended battery life. The app should also handle **multiple 3D model uploads** and processing without significant delays or crashes.

2.5 User Feedback Requirements

User feedback is a vital component of the **Delta3D** development process, ensuring the app continuously evolves to meet user expectations. The app incorporates a **feedback collection system**, where users can submit bug reports, feature requests, or general feedback. This feedback can be gathered both through in-app prompts and customer support channels.

Once collected, the feedback is managed through a **feedback management system**, where it is categorized based on its nature (bug, feature request, etc.). The feedback will be prioritized according to its impact on user experience and system functionality. Regular review sessions will be conducted to assess the feasibility of addressing high-priority feedback.

The **feedback processing** cycle is designed to be quick and efficient. **Critical issues,** particularly those affecting app stability or core features, are addressed immediately and included in upcoming releases. Regular communication is maintained with users to inform them about the status of their feedback, ensuring they remain engaged and informed about app improvements.

3. System Design

3.1 Architectural Design

3.1.1 System Overview

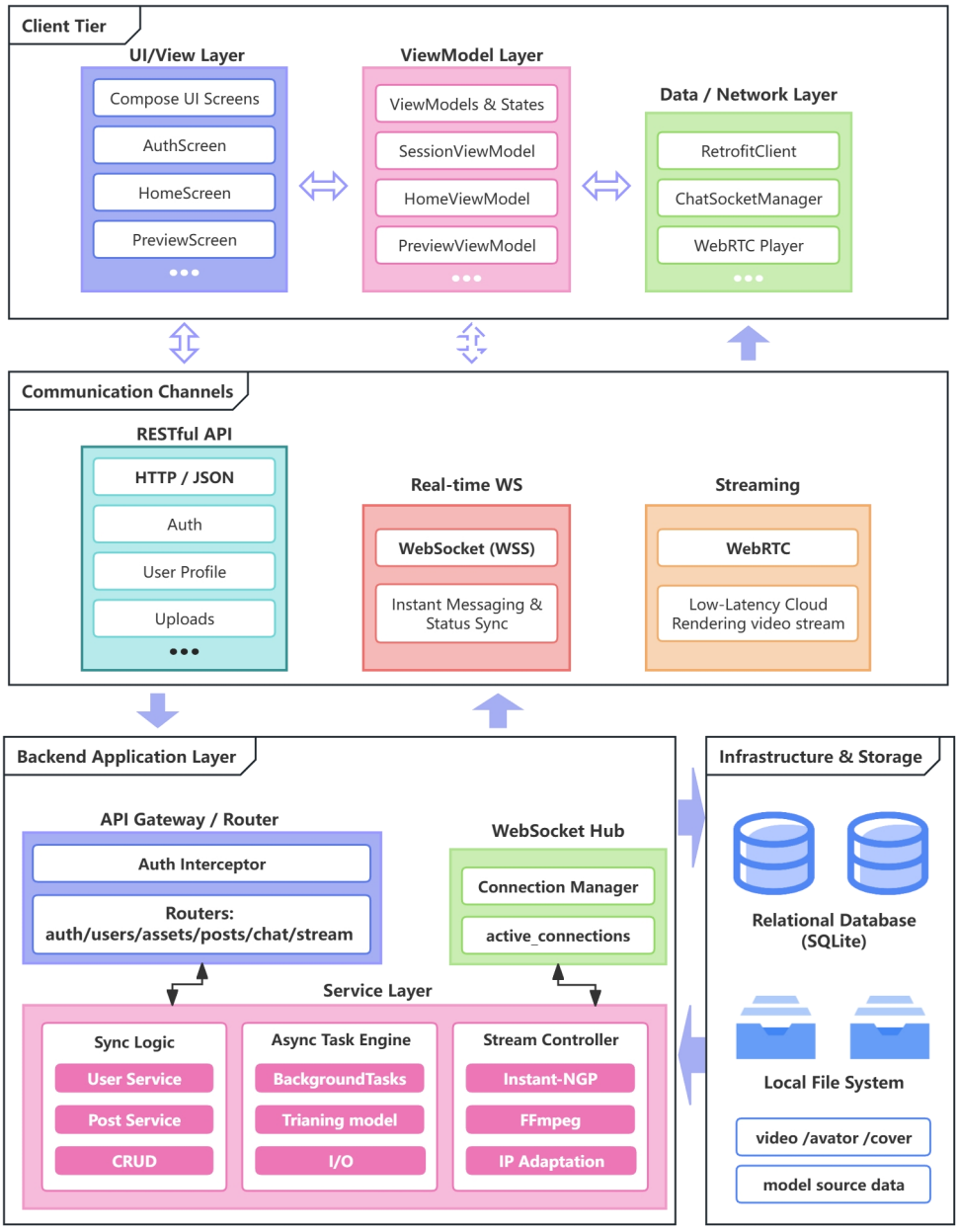

Figure 3-1, System Overall Architecture Diagram (simplified)

As illustrated in Figure 3-1, the Δ3D system is designed using a robust Client-Server (C/S) architecture, featuring a strict separation between the frontend presentation layer and the backend logic layer.

Frontend Subsystem (Client Layer) The client side is a native Android application developed with Kotlin and Jetpack Compose. It adheres to the MVVM (Model-View-ViewModel) architectural pattern. The client serves as the primary interface for users to capture video data, manage 3D asset models, and interact with the community. It integrates a dedicated WebRTC Player for rendering cloud-streamed 3D content and a WebSocket Client for real-time messaging.

Backend Subsystem (Server Layer) The server side is powered by Python FastAPI, structured into a four-layer architecture (Router, Service, CRUD, Infrastructure). A key characteristic of the backend is its Asynchronous Processing Engine. By leveraging FastAPI's BackgroundTasks, resource-intensive operations, such as 3D model training via Instant-NGP, are executed asynchronously, preventing blockage bof the main thread and ensuring high system responsiveness. Additionally, a dedicated Stream Manager orchestrates external rendering processes (NGP and FFmpeg) to deliver real-time video streams.

Hybrid Communication Protocols To support diverse business requirements, the system implements a hybrid communication stack:

- **RESTful APIs (HTTP):** Handle synchronous data transactions such as user authentication and asset metadata management.
- **WebSocket (WSS)**: Provides full-duplex channels for the Instant Messaging (IM) module.
- **WebRTC**: Facilitates low-latency 3D streaming. The system includes an intelligent IP Adaptation Logic that dynamically routes streams via public or private IP addresses based on the user's network environment.

This architecture prioritizes loose coupling and non-blocking I/O, enabling the system to handle heavy computational tasks (3D rendering) while maintaining a smooth and responsive user experience for social interactions.

3.1.2 Logical Architecture

To ensure high maintainability, testability, and scalability, the system implements standard architectural patterns on both the backend and frontend.

1. Backend: Four-Layer Decoupled Architecture

As shown in Figure 3-2, The backend adopts a strict Layered Architecture to achieve Separation of Concerns (SoC). The codebase is organized into four distinct layers:

- **API / Router Layer**: Located at the top level (*endpoints/*.py*), this layer acts as the entry point. It is responsible for handling HTTP requests, validating input parameters via Pydantic schemas, and formatting JSON responses. It contains no core business logic.
- **Service Logic Layer:** This layer (e.g., *ngp/worker.py, stream_manager.py*) encapsulates the core domain logic. It manages complex workflows such as

the scheduling of **3D model training pipelines**, asynchronous task distribution, and stream process management.

- **Data Access Layer (CRUD):** All database interactions are encapsulated within the *crud/*.py* modules. This layer provides a clean abstraction for Create, Read, Update, and Delete operations, shielding the upper layers from specific ORM (SQLModel) implementation details.
- **Infrastructure Layer:** The bottom layer (e.g., *core/*.py*) provides essential cross-cutting concerns, including configuration management, **JWT-based security authentication**, database connection pooling, and WebSocket connection management.

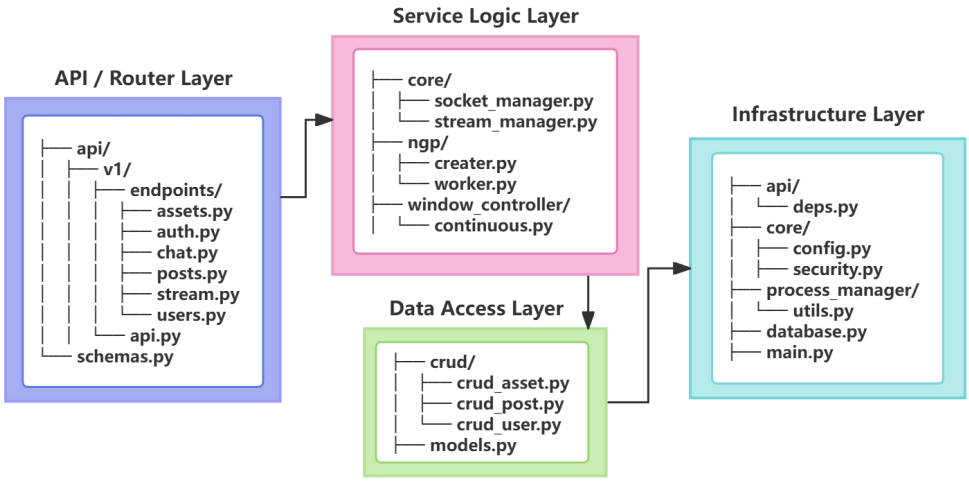

Figure 3-2, Backend and Layer Mapping

2. Frontend: MVVM Pattern with Unidirectional Data Flow

As shown in Figure 3-3, the Android client is structured using a Package-by-Feature strategy combined with the **MVVM pattern**. The architecture is divided into five logical layers to handle UI rendering, state management, and underlying infrastructure.

1) **Presentation Layer (UI & Components)**: Located in *ui/screens/** and *ui/components/**. This layer is responsible for rendering the interface and capturing user inputs.
   - **Feature Screens:** The UI is modularized by features (e.g., home, chat, preview), promoting high cohesion. Complex screens like *StreamPreviewScreen* handle custom gesture detection for 3D interaction.
   - **Reusable Components:** Shared UI elements such as *BottomNavBar* and *NetworkStatusDialog* are isolated in *ui/components* to ensure design consistency across the application.
2) **State Management Layer (ViewModel)**: Located in *ui/*/ViewModel.kt* and *ui/session/*.
   - **Global Session Scope:** The *SessionViewModel* acts as a singleton state container, managing the application's lifecycle, user authentication status, and

global unread message counters (*_totalUnreadCount*).

- **Screen Scope:** Feature-specific ViewModels (e.g., *ChatViewModel*, *StreamViewModel*) manage local UI states using *StateFlow*, ensuring a unidirectional data flow from the data layer to the UI.

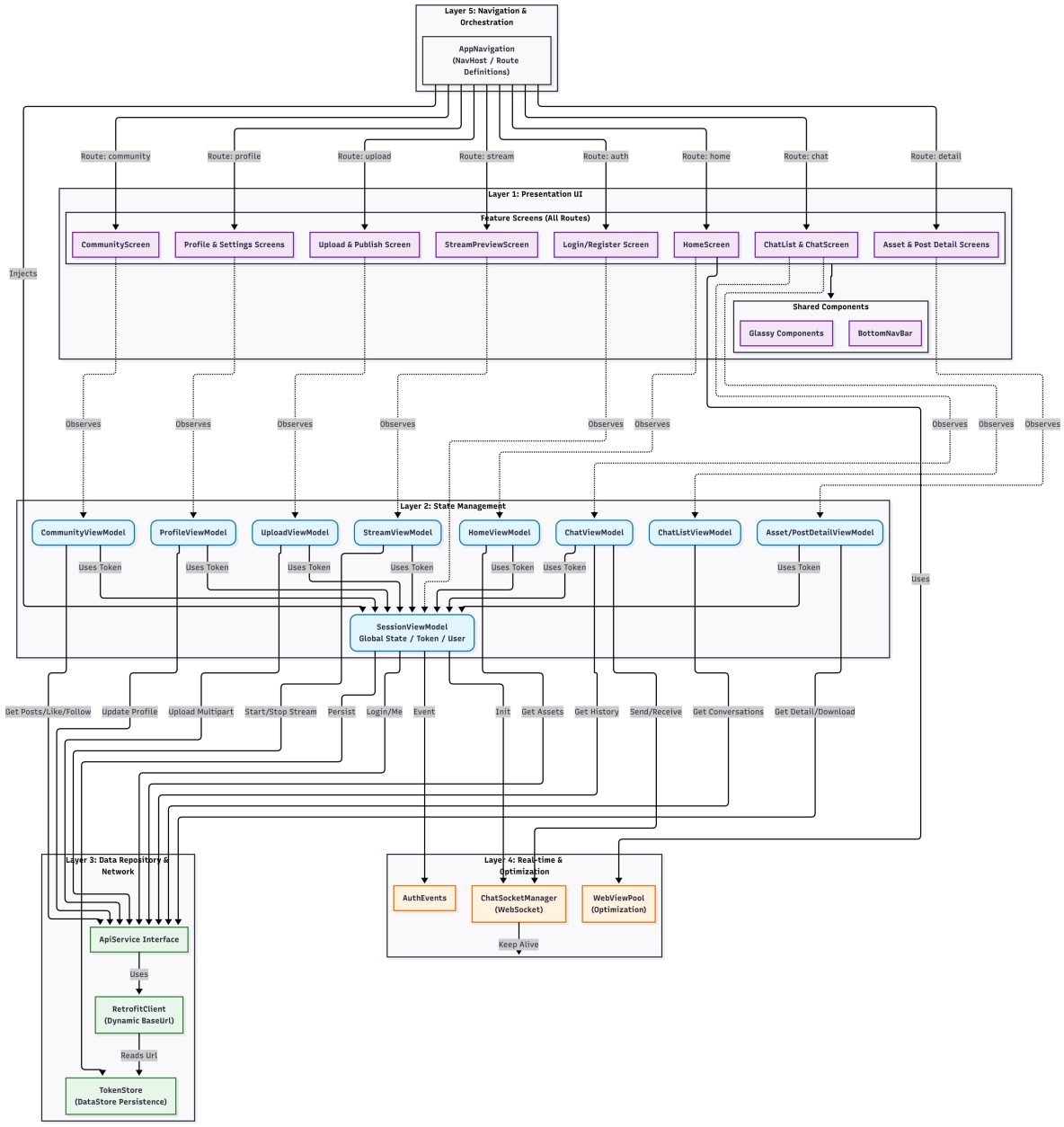

Figure 3-3, Frontend and Layer Mapping

3) **Data Repository & Network Layer**: Located in *api/* and *data/*.

- **API Contract:** *ApiService* serves as the single source of truth for all HTTP endpoints, defining strict request/response contracts using DTOs like *AssetCard* and *UserDetail*.
- **Network Interception:** A custom *AuthInterceptor* is implemented to

automatically intercept 401 Unauthorized responses and trigger global logout events, ensuring security compliance.

- **Local Persistence:** *TokenStore* encapsulates local persistence of authentication tokens using Android DataStore, abstracting the underlying storage implementation.

4) **Real-time & Infrastructure Layer**: Located in *manager/* and *utils/*. This layer handles long-running connections and performance optimizations.
   - **Socket Manager:** The *ChatSocketManager* (in *manager/*) maintains a persistent WebSocket connection. It handles heartbeat mechanisms (*pingInterval*) and message dispatching, independent of the UI lifecycle.
   - **Event Bus:** *AuthEvents* provides a decoupled communication channel for handling system-wide events (e.g., forced logout) without creating rigid dependencies between network clients and UI activities.
   - **Performance Optimization:** *WebViewPool* (in *utils/*) implements an object pooling pattern to pre-initialize and recycle WebView instances, significantly reducing page load times for web-based content.
5) **Navigation & Orchestration**
   - **Routing Hub:** *AppNavigation* defines the complete navigation graph. It acts as the dependency injection root, creating and passing the *SessionViewModel* to downstream screens.

3.2 Frontend Module Design

The Δ3D frontend primarily consists of six core modules: Infrastructure, Asset Management, Community Interaction, Real-time Communication, Streaming Control, and User Center.

3.2.1 Infrastructure & Network Module

This module serves as the underlying foundation of the system, responsible for dispatching network requests, global state management, and performance optimization, ensuring stable application operation in both Local Area Network (LAN) and Public Network environments.

1) **Dynamic Network Routing Layer**

   Core Components: *RetrofitClient, HostSelectionInterceptor, AuthInterceptor*

   Responsibility: Acts as the HTTP execution pipeline and gateway. It routes requests to the correct endpoint at runtime and centrally monitors unauthorized responses.

   Internal Logic:

   - **Dynamic host rewrite:** *HostSelectionInterceptor* intercepts requests and reads *AppConfig.currentBaseUrl*. It dynamically reconstructs the *scheme*,

host, and *port* based on the real-time network mode determined by the Session Manager, enabling hot switching without app restarts.

- **Unauthorized response monitoring**: *AuthInterceptor* inspects responses; on 401 Unauthorized, it triggers *AuthEvents* to notify the session/UI layer for a unified logout flow.

Collaboration (with Session Manager):

- *SessionViewModel* maintains the session token and provides it to API calls (passed as the Authorization header parameter in *ApiService* methods in the current implementation).

Interaction Sequence Diagram:

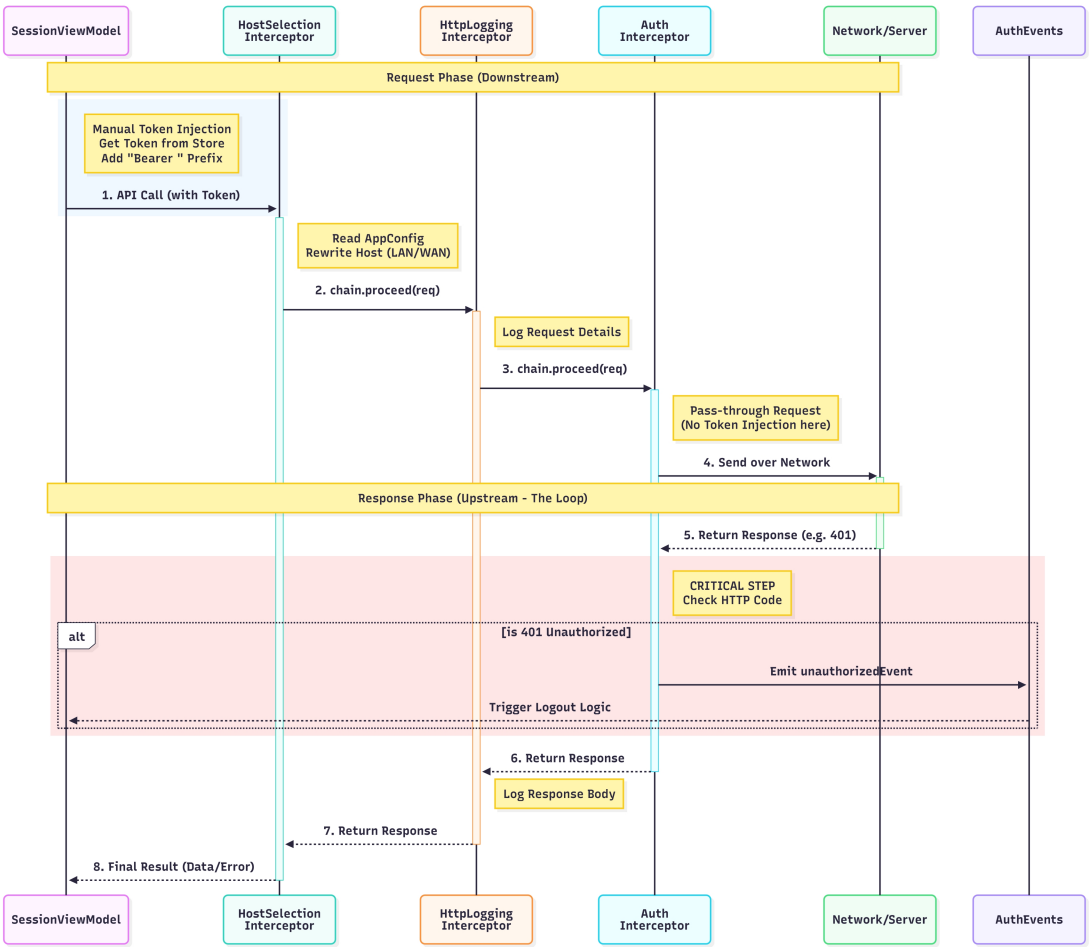

Figure 3-4, Dynamic Network Routing Sequence Diagram

2) Global Session Manager

Class Name: *SessionViewModel*

Responsibility: Serves as an app-scoped state hub. It manages user identity, persists tokens, performs network reachability probing to select the best API mode (LAN/WAN), and enforces consistent session cleanup on logout/unauthorized events.

Internal Logic:

- **Network Intelligence:** Listens for network availability via *ConnectivityManager* callbacks. Upon network availability, it debounces and performs active LAN reachability probing using raw *Socket.connect* (LAN host/port, 2s timeout). It updates *AppConfig.currentMode* only when the selected mode changes and emits a UI notification flag (*networkCheckFinished*).
- **Token Persistence & Formatting:** Restores the saved token from *TokenStore* (DataStore) at startup and keeps it in a *StateFlow*. For internal API calls (e.g., fetching profile / unread counts), it manually formats the Authorization header by adding the "Bearer " prefix when needed.
- **State Distribution**: Exposes *token, currentUser,* and *totalUnreadCount* as *StateFlow* to act as the Single Source of Truth for the UI, ensuring the App knows when data restoration is complete (*loaded*).
- **Initialization & Restoration:** Upon instantiation, it asynchronously rehydrates the session state from *TokenStore* (loading token and first-launch status) and signals the UI via the loaded flag. If a valid token is restored, it automatically chains a request to fetch the latest user profile (*fetchCurrentUser*).
- **Unread Count Aggregation:** Implements a pull-based strategy for notification badges. When refreshing user info or triggered manually, it fetches the conversation list (*getConversations*) and calculates the global unread count (*_totalUnreadCount*) by summing the unread messages of all active conversations.
- **Session Cleanup:** Subscribes to *AuthEvents.unauthorizedEvent*; when triggered or logout is called, it performs a complete cleanup: clears token/user state, resets unread counts to zero, disconnects the *ChatSocketManager*, and clears the *TokenStore* to prevent resource leaks.

3) Web Rendering Resource Pool

Class Name: *WebviewPool*

Responsibility: Addresses the "white screen" issue caused by the initialization latency of the Android native *WebView*. It manages the reuse and recycling of web containers using an Object Pool pattern.

Internal Logic:

- **Pre-warming:** During the application startup phase, the *WebViewPool* utilizes the *IdleHandler* mechanism to instantiate *WebView* objects and load basic HTML skeletons (e.g., Logo and interactive Tree) when the main thread is idle. This ensures assets are ready before the *Delta3DLogoSplash*

or *DeltaTreeScreen* requests them.

- **Recycling Strategy:** Maintains a *LinkedList<WebView>* pool. When a resource is requested, it prioritizes *obtain()* from existing instances. When a view is destroyed, it calls *recycle()*, which resets the view using *loadUrl("about:blank")* and detaches it from the parent container to ensure a clean state for the next reuse.

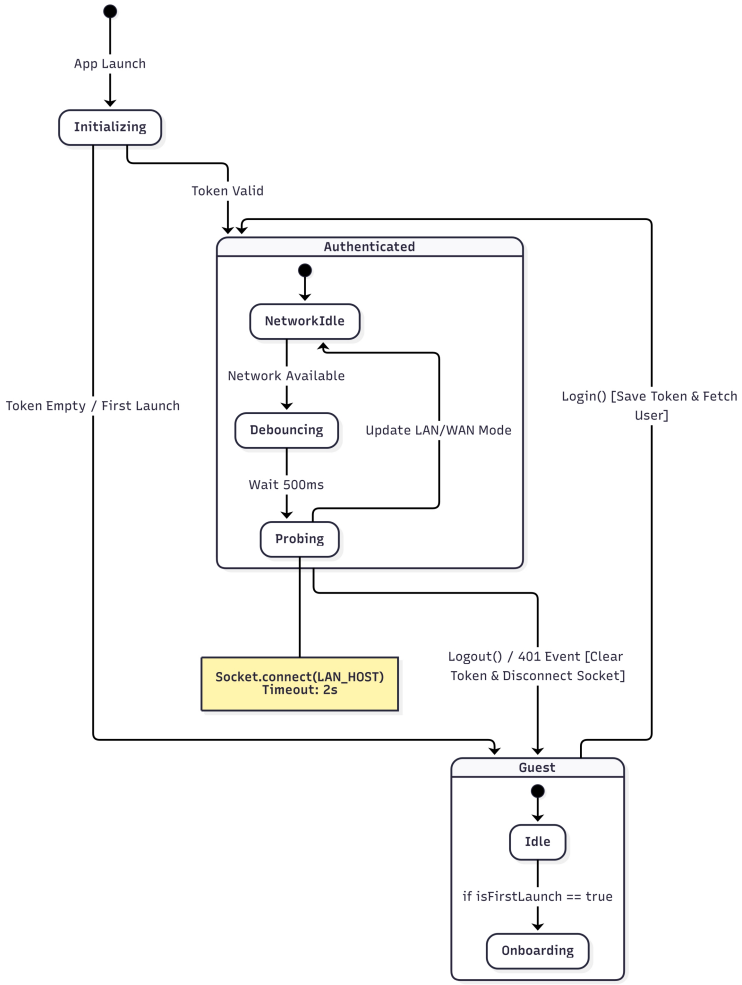

Figure 3-5, Global Session Management Flow

4) Navigation Orchestration & Routing Center

Class Name: *AppNavigation*

Responsibility: Acting as the Navigation Root, it implements the Single Activity architecture. It defines the application-wide navigation graph (*NavGraph*), manages bottom bar visibility, and serves as the Dependency Injection Root to inject the global *SessionViewModel* into various screens.

Internal Logic:

- **Route Definition:** Uses *NavHost* to declaratively define route nodes such as *auth, home, community, and detail/{id}*, handling DeepLink arguments (e.g., *assetId*).
- **Dependency Injection:** Passes the *SessionViewModel* instance down to

child screens (e.g., *HomeScreen, ProfileScreen*) during the Composable composition phase, ensuring token state consistency across the app.

- **Global Overlay Management:** Manages global dialogs, such as the *NetworkStatusDialog*, at the top level of the navigation graph, ensuring network status prompts overlay any business page.

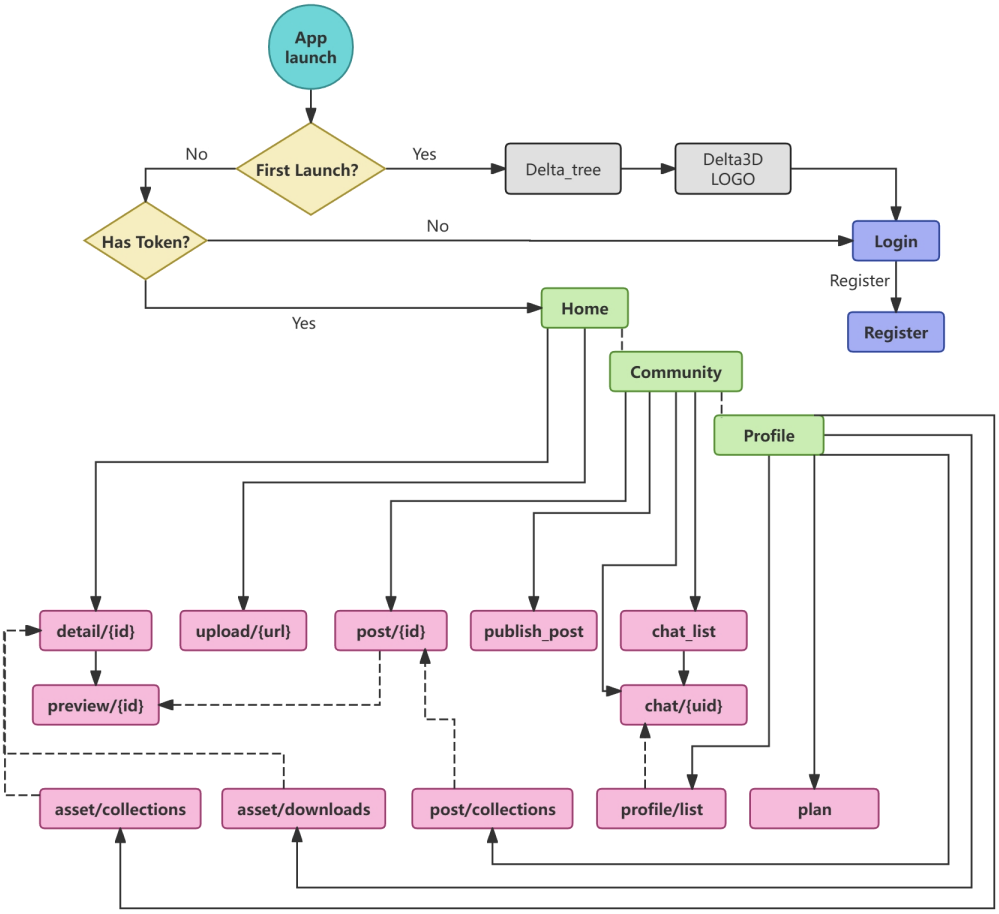

Figure 3-6, Global Navigation Graph and Startup Logic

3.2.2 3D Model Asset Management Module

This module is responsible for the display, retrieval, detail viewing, and file downloading of 3D models, serving as the core entry point for content consumption.

1) **Model Asset List and Display**

View Layer: *HomeScreen* implements a Responsive Layout Strategy using *LazyVerticalStaggeredGrid* to create a waterfall effect that adapts to asset cover images. It features a Parallax Scroll System via a custom *NestedScrollConnection*, which intercepts scroll events (*onPreScroll/onPostScroll*) to dynamically calculate header height and opacity, creating a seamless collapsing effect for the

3D header. Additionally, it supports an animated transition between grid view and list view (additionally includes tags) modes using *AnimatedContent*.

ViewModel Layer: *HomeViewModel* serves as the state holder, managing the *_displayAssets* flow for UI rendering and *_processingCount* to track active server-side tasks.

Search Algorithm: A Debounced Filtering Mechanism is implemented using Kotlin Coroutines. Input text triggers a *delay(600)* suspension to prevent excessive computation during rapid typing. The filter logic performs local matching against both asset titles and tags to update the display list without repeated network requests.

Optimistic UI Pattern: The "Collect" toggle action immediately updates the local UI state (flipping the *isCollected* boolean) before the network request completes. A backup of the original state (*backupAssets*) is maintained to allow for an automatic rollback if the API call fails, ensuring a responsive user experience.

Hybrid Rendering: The *HomeTreeCard* component integrates web technologies via *AndroidView* to render a Three.js-based 3D scene (*delta_tree.html*). It utilizes a *WebViewClient* to inject JavaScript for state synchronization, such as resetting camera positions, and manages touch conflicts between the web canvas and native scrolling.

2) **Model Asset Creation and Uploading**

Client-Side Validation: *UploadViewModel* implements a Pre-flight Check using *ContentResolver* and *MediaMetadataRetriever*. It enforces strict constraints locally (blocking files larger than 50MB or longer than 60 seconds) to reduce unnecessary server load and bandwidth usage.

Algorithmic Time Estimation: Instead of relying solely on backend feedback, the client calculates an Estimated Processing Time using a linear regression model: *BaseTime (30s) + (SizeMB / 4) * 15s*. This provides immediate feedback to the user regarding task duration.

Tag Recommendation Engine: The system aggregates distinct tags from the user's existing asset library via *fetchUserTags*, combining them with a default set (e.g., "Human", "Sci-Fi") to provide context-aware autocomplete suggestions during the upload process.

Stream Management: The module handles file I/O by converting the selected content *Uri* into a temporary *File* in the application cache using *FileOutputStream*. This ensures a stable file stream is available for the Retrofit *MultipartBody* request and handles cleanup automatically after upload.

3) **Model Asset Details and Status Visualization**

Status Timeline System: The *ProcessingTimeline* component visualizes the asset lifecycle ("Queued" -> "Processing" -> "Ready") with estimated time. It uses a *Canvas* to draw connected nodes and dynamic progress lines that change color based on the current status (e.g., Red for failure, Green/Cyan for success).
Non-Linear Progress Simulation: For model in the "Processing" state, the UI implements an Asymptotic Progress Logic. If the processing time exceeds the initial estimate, the progress bar enters an "overtime" curve *(overtime / (overtime + 3.0f))*, slowly approaching 99% but never reaching completion until the server confirms the status, preventing misleading "100%" indicators.
Lifecycle-Aware Synchronization: The module integrates a *LifecycleEventObserver* within *HomeScreen* to monitor *ON_RESUME* events. This triggers an automatic data refresh when returning from the detail view or background, ensuring status updates (like a completed processing task) are reflected immediately.
Multimedia Preview: The *ImageCarouselHeader* employs a fallback strategy for generating previews. While the 3D model is processing, it generates a sequence of image URLs (e.g., /images/0001.jpg) derived from the video frames with file path of model to display a rotating slideshow above.

4) **Model Asset Interaction and Feedback**

Unified Feedback System: A *GlassyFeedbackPopup* component is implemented to standardize system notifications. It decouples UI feedback from business logic by observing a reactive state, supporting different feedback types (Success, Error, Info) with custom animations.
Download Management: *AssetDetailViewModel* handles Format Mapping, converting labels (e.g., "Source Data") into backend-compatible file types (e.g., "msgpack"). The actual download is offloaded to the Android *DownloadManager*, ensuring background persistence and system-level notifications even if the app is closed.
Community Publish: At the same time, clicking "Share" can also directly pre-select the model and then enter the *PublishPostScreen* by calling *onNavigateToPublish(assetId)*.
Technical Specifications Display: The *TechSpecsCard* dynamically renders technical metadata, such as Triangle count, Vertex count, and Material sets, providing deeper insights for developer-centric users.

3.2.3 Community Interaction Module

1) **Feed & Discovery System**

Dual-Stream Architecture:

- The *CommunityScreen* implements a toggleable view strategy (*Explore vs. Following*). State management involves a client-side filter (*_onlyShowFollowing*) applied to the master post list, allowing instant switching without redundant network requests.
- Visual transitions between tabs use *animateFloatAsState* for scaling and opacity changes, creating a fluid header interaction.

Optimistic UI Pattern (Social Actions):

- Social interactions (Like, Collect, Follow) employ an Optimistic Update Strategy. The *CommunityViewModel* immediately updates the local state (e.g., flipping *isLiked* and incrementing counters) before the API call completes.
- **Rollback Mechanism**: If the backend request fails (captured via *try-catch*), the state is reverted to its original value, and a UI event is emitted to trigger a *GlassToast* error notification.

Hybrid Search Filtering:

- The search function combines network data with local filtering. It checks multiple fields simultaneously: post titles, descriptions, owner names, and tags.
- To optimize performance, a coroutine-based Debounce (*delay(800)*) is applied to the search input flow, preventing calculation during rapid typing.

2) **Post Details & Engagement**

Immersive Detail View:

- *PostDetailScreen* features a *LazyColumn* layout that integrates a multimedia header (*ImageCarouselHeader*), rich text content, and an interactive comment section.
- **Micro-Interactions**: The Like and Collect buttons (*AnimatedInteractionButton*) utilize *spring* physics-based animations (damping ratio 0.4, stiffness 400) to provide tactile feedback upon activation.

Context-Aware Sharing:

- The module includes a *ShareActionDialog* and a *ModalBottomSheet* for sharing. It fetches the user's recent chat conversations via *RetrofitClient* to populate a "Quick Share" list.
- When sharing to a chat, it generates a specialized rich-media link (parsed by *ShareLinkUtils*), enabling the recipient to see a preview card instead of a raw URL.

Technical Metadata Visualization:

- The screen reuses the *TechSpecsCard* component to display 3D-specific data (triangle count, texture resolution) derived from the associated asset, providing technical context to the social post.

3) Real-Time Communication (Chat)

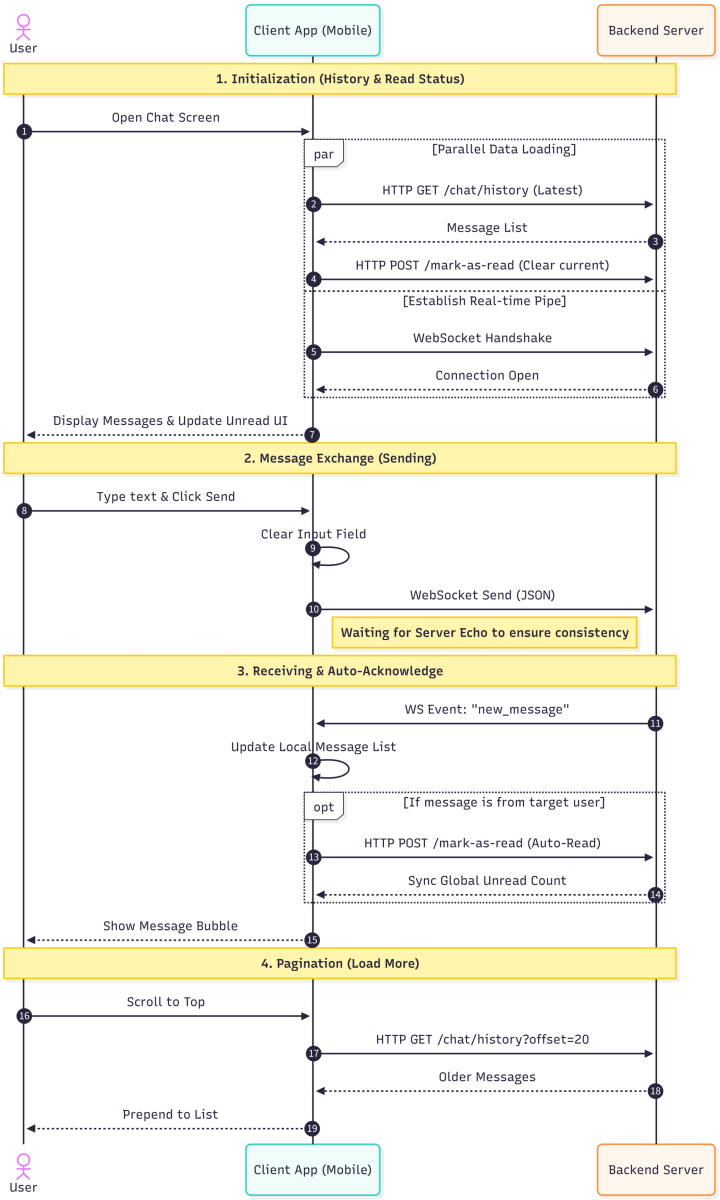

Figure 3-7, Chat Module Core Interaction Sequence Diagram (simplified)

Hybrid Messaging Architecture:

- The system combines REST API for history pagination with WebSocket for real-time delivery.
- *ChatViewModel* initializes by loading historical messages (reversed for bottom-up rendering) and concurrently establishes a socket connection via *ChatSocketManager*.

Smart Pagination & Scroll Management:

- The chat UI uses a *LazyColumn* with *reverseLayout = true*. This ensures that new messages naturally appear at the bottom without complex scroll calculations.
- Infinite scrolling is implemented via *snapshotFlow*. It monitors the *firstVisibleItemIndex* and triggers *loadMoreHistory* when the user scrolls near the "top" (visually the bottom of the data list).

Rich Media Rendering:

- The *MessageBubble* component dynamically discriminates between plain text and rich content. It detects special share links using *ShareLinkUtils* and renders a *PostShareCard* (displaying cover image, title, and type) instead of text, allowing direct navigation to the shared model.

Read Status Synchronization:

- The module implements automatic read receipts. When entering a chat or receiving a message while the screen is active, *markMessagesAsRead* is invoked to sync the status with the backend.

4) Content Publishing Flow

Asset-First Publishing:

- *PublishPostScreen* enforces an asset-centric workflow. Users must select a "Completed" model from their library using a custom *ModelSelectorDropdown* before writing content.
- The dropdown features an expandable UI with animation (*animateContentSize*), rendering asset thumbnails and status indicators directly in the selection list.

Privacy & Permissions Control:

- Users can configure granular permissions at the time of publishing, including Visibility scopes (*Public/Followers*) and a toggle for *Allow Download* (controlling source file access for others).

3.2.4 Streaming & Control Module

1) Real-time Stream Rendering

Hybrid WebRTC Architecture:

- **Strategy:** To bypass the complexity and compatibility issues of native Android WebRTC implementations, the module adopts a WebView Bridge Strategy. *StreamPreviewScreen* embeds a highly customized *WebView* via *AndroidView* to load the local asset *webrtc_player.html*.
- **WHEP Protocol Integration**: The frontend HTML5 player implements the WHEP (WebRTC-HTTP Egestion Protocol). It uses the *fetch* API to send an

SDP Offer to the backend stream URL (automatically appending the */whep* suffix), completing ICE negotiation and establishing a P2P connection for millisecond-level low-latency video transmission.

- **WebView Pooling**: To mitigate the first-frame latency caused by WebView initialization, the *WebViewPool* is introduced. WebView instances are retrieved directly from the pool upon entering the page and recycled upon exit, significantly improving page load speeds.

2) Remote Control System

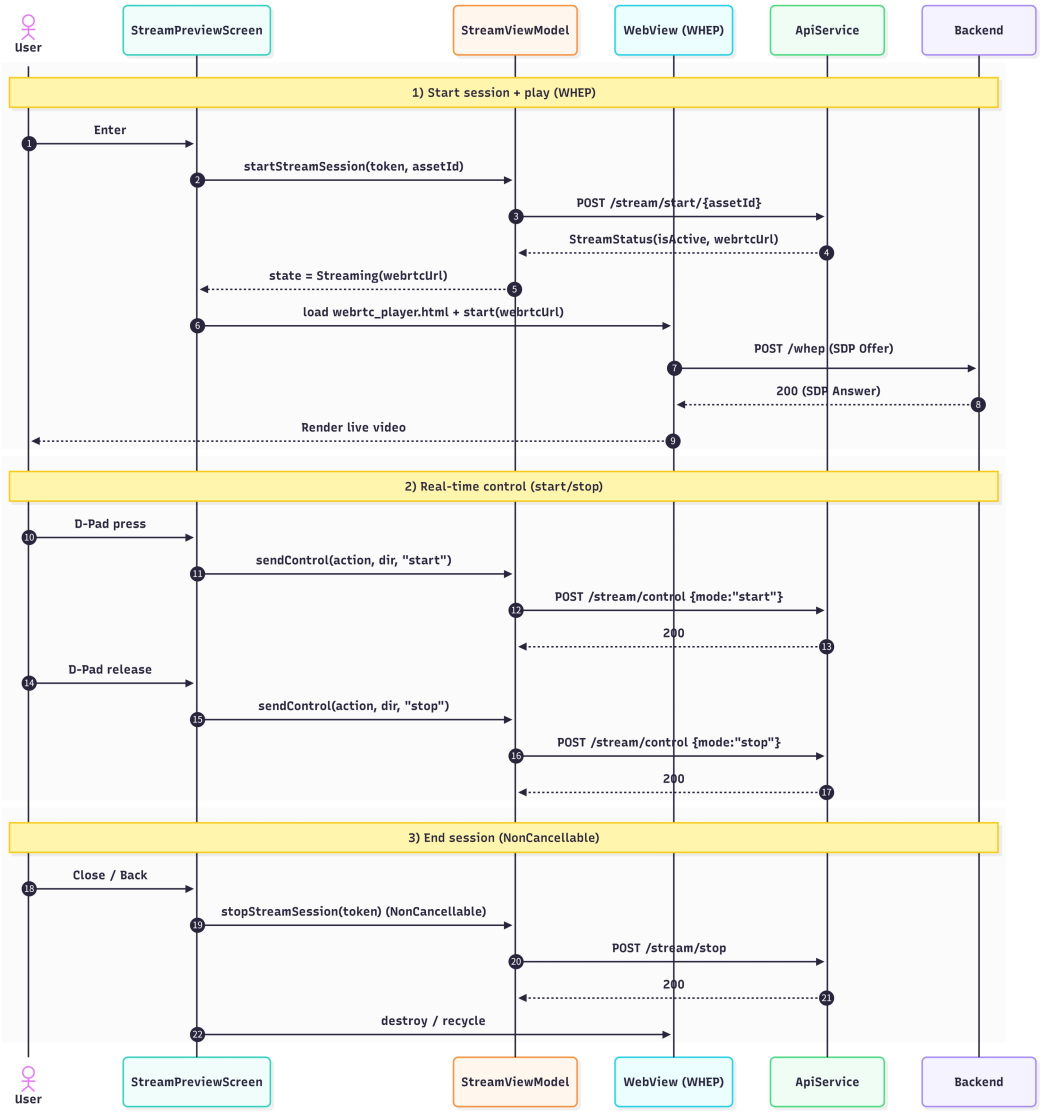

Figure 3-8, Real-time Streaming & Remote Control Interaction Sequence Diagram

Command-Response Model:

- Control instructions are structured as a *ControlCommand* data class, comprising three dimensions:
  - **Action**: The type of movement (Rotate / Pan / Zoom).

- **Direction**: The direction vector (Up / Down / Left / Right / In / Out).
- **Mode**: The trigger state (Start / Stop).

- This design allows the backend to precisely handle "press-to-move" and "release-to-stop" operational logic.

Continuous Touch Interaction:

- **Custom Long-Press Component**: A *RepeatButton* component is implemented using *pointerInput* and *detectTapGestures* to listen for low-level *onPress* and *tryAwaitRelease* events.
- **Logic**: When the user holds down a directional key, a *mode="start"* command is sent. Upon release, a *mode="stop"* command is immediately dispatched. This simulates the tactile experience of a physical D-Pad, enabling smooth, continuous manipulation of the cloud-based 3D model.

Mode Switching Logic:

- The control panel (*StreamControlOverlay*) maintains a local state *isPanMode*. Users can seamlessly switch between "Rotate" and "Pan" modes, with D-Pad directional signals automatically mapping to the context of the current action.

3) Session & Lifecycle Management

Non-Blocking Resource Cleanup:

- The *stopStreamSession* method in *StreamViewModel* utilizes the *withContext(NonCancellable)* context. This ensures that the network request to stop the stream (*stopStream*) is successfully dispatched even if the ViewModel's main coroutine scope is cancelled due to the user killing the app or navigating away quickly, preventing server-side resource leaks.

Automatic Signaling Injection:

- *StreamPreviewScreen* monitors *uiState*. Once the state transitions to Streaming, it automatically injects the backend-generated stream URL into the WebView's *start()* function via *evaluateJavascript*, triggering the WebRTC connection process without manual user intervention.

Adaptive Reconnection:

- The HTML5 player implements an internal reconnection logic. When a connection fails or drops, the retryCount increments, and the UI status text updates (e.g., "ATTEMPTING (1/20)...") until a connection is re-established or the maximum retry limit is reached, providing a robust experience under weak network conditions.

3.2.5 User Center & Personalization Module

1) **Identity & Profile Management**

Atomic Profile Updates:

- The *ProfileViewModel* implements a transactional update flow for user profiles. It handles multipart form data to upload avatars (converting *Uri* to *File* via *ContentResolver*) while simultaneously updating text fields (Nickname, Bio, Gender).
- **Session Synchronization**: Upon a successful profile update, it triggers *sessionVm.refreshUser()* to strictly ensure that the global user session (cached in *SessionViewModel*) remains consistent with the backend state, preventing stale data across the app.

Relationship Management:

- The *UserListScreen* serves as a unified interface for both "Followers" and "Following" lists. It utilizes *ProfileViewModel* to fetch paginated relationship data.
- **In-List Interaction**: Users can toggle follow states directly within these lists. The system employs **optimistic updates** to instantly reflect changes (e.g., changing "Follow" to "Following") while the network request processes in the background.

2) **Unified Collection System**

Dual-Type Aggregation:

- The module provides specialized views for different content types: *SavedAssetsScreen* for 3D models and *SavedPostsScreen* for community posts.
- **Component Reuse Strategy**: Instead of duplicating UI logic, it reuses the core rendering components *ProductListView* (from Model Asset Module) and *CommunityFeedItem* (from Community Module), ensuring visual consistency and reducing code maintenance.

Reactive State Management:

- In *SavedPostsViewModel*, social actions (Like, Collect, Follow) are handled with an **Optimistic UI pattern**. For example, when un-collecting an item, it is immediately removed from the local list to maintain UI responsiveness.
- **Error Handling**: If the API call fails (e.g., network error), the state rollback mechanism restores the item to the list and notifies the user via a global UI event.

3) **Activity Tracking & History**

Download Audit Trail:

- *DownloadHistoryScreen* provides a persistent log of user asset consumption.

It fetches history via *DownloadHistoryViewModel*.

- **Timezone Adaptation**: A custom *TimeUtils* logic is applied to timestamp strings, converting server UTC times to the user's local context for better readability.

Visual Status Indicators:

- The history list utilizes distinct icons (e.g., *Lock* vs *DownloadDone*) to visually differentiate between accessible assets and those that might have expired or changed permissions.

4) **Subscription & Plan View**

Static Configuration Rendering:

- *PlanSettingsScreen* renders the user's current subscription tier ("Free Plan"). It uses a *Canvas* with *Brush.radialGradient* to create a premium visual effect for the plan card, distinguishing it from standard UI elements.

3.3 Backend Module Design

The backend system is built on **Python FastAPI**, leveraging its asynchronous capabilities to handle high-concurrency network I/O while orchestrating heavy computational tasks (3D training and rendering). The architecture follows strict separation of concerns, divided into four core modules: API Gateway, Core Service Logic, Real-time Communication, and Infrastructure & Data Access.

3.3.1 API Gateway & Router Module

This module acts as the unified entry point for all client requests, responsible for protocol parsing, parameter validation, and security interception.

1) **Router Dispatcher**

Component: *api_router* (Instance in *api.py*)

Responsibility: Aggregates sub-routers (*auth, posts, assets, stream, chat*) into a single routing table, applying prefix grouping and tag management for automatic API documentation generation.

Internal Logic:

- **Dependency Injection:** The system heavily utilizes FastAPI's *Depends* system. For instance, the Core Function *get_current_user* is injected into protected endpoints. It automatically parses the JWT *Authorization* header and retrieves the *User* object before the business logic executes.
- **Schema Validation:** All request bodies are strictly validated against Pydantic models (e.g., *PostCreate, MessageCreate*). This ensures that invalid data types are rejected with *422 Unprocessable Entity* before reaching the service layer.

2) **Security & Authentication**

Core Functions: *create_access_token* & *get_current_user* (in *deps.py*, *security.py*)

Responsibility: Implements the OAuth2 Password Bearer flow and stateless JWT verification.

Internal Logic:

- **Stateless Verification:** *create_access_token* generates a token using the *HS256* algorithm and a server-side *SECRET_KEY*.
- **Context Injection:** The *get_current_user* function acts as a gatekeeper. It decodes the token payload to extract the *sub* (User ID), verifies identity without maintaining a session table, and injects the *current_user* instance into the request context.

3.3.2 Core Service Logic Module

This module is the "brain" of the backend, encapsulating complex business logic, background task scheduling, and external process management.

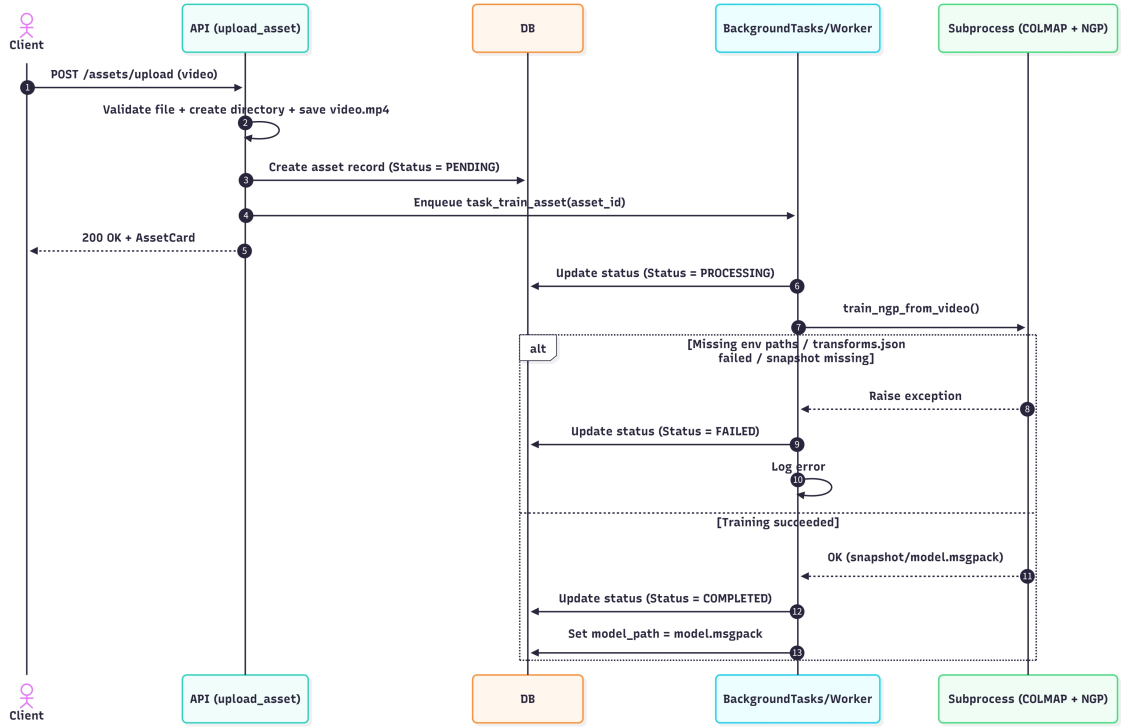

Figure 3-9, Model Upload & Training Workflow Sequence Diagram

1) **Asynchronous Training Engine**

Core Function: *task_train_asset* (in *worker.py*)

Responsibility: Manages the lifecycle of 3D model generation tasks (Video -> NeRF Model) without blocking the main HTTP thread.

Internal Logic:

- **Fire-and-Forget Scheduling:** The */assets/upload* endpoint utilizes FastAPI's *BackgroundTasks* to enqueue the *task_train_asset* function immediately after returning the *200 OK* response.
- **Pipeline Orchestration:** This function sequentially executes the photogrammetry pipeline:
  - **Preprocessing:** Calls *colmap2nerf.py* to extract frames and calculate camera poses.
  - **Training:** Invokes the Instant-NGP training script via the *NonBlockingCommandRunner* utility class.
  - **State Transition:** Updates the database state atomically from *PENDING -> PROCESSING -> COMPLETED*.

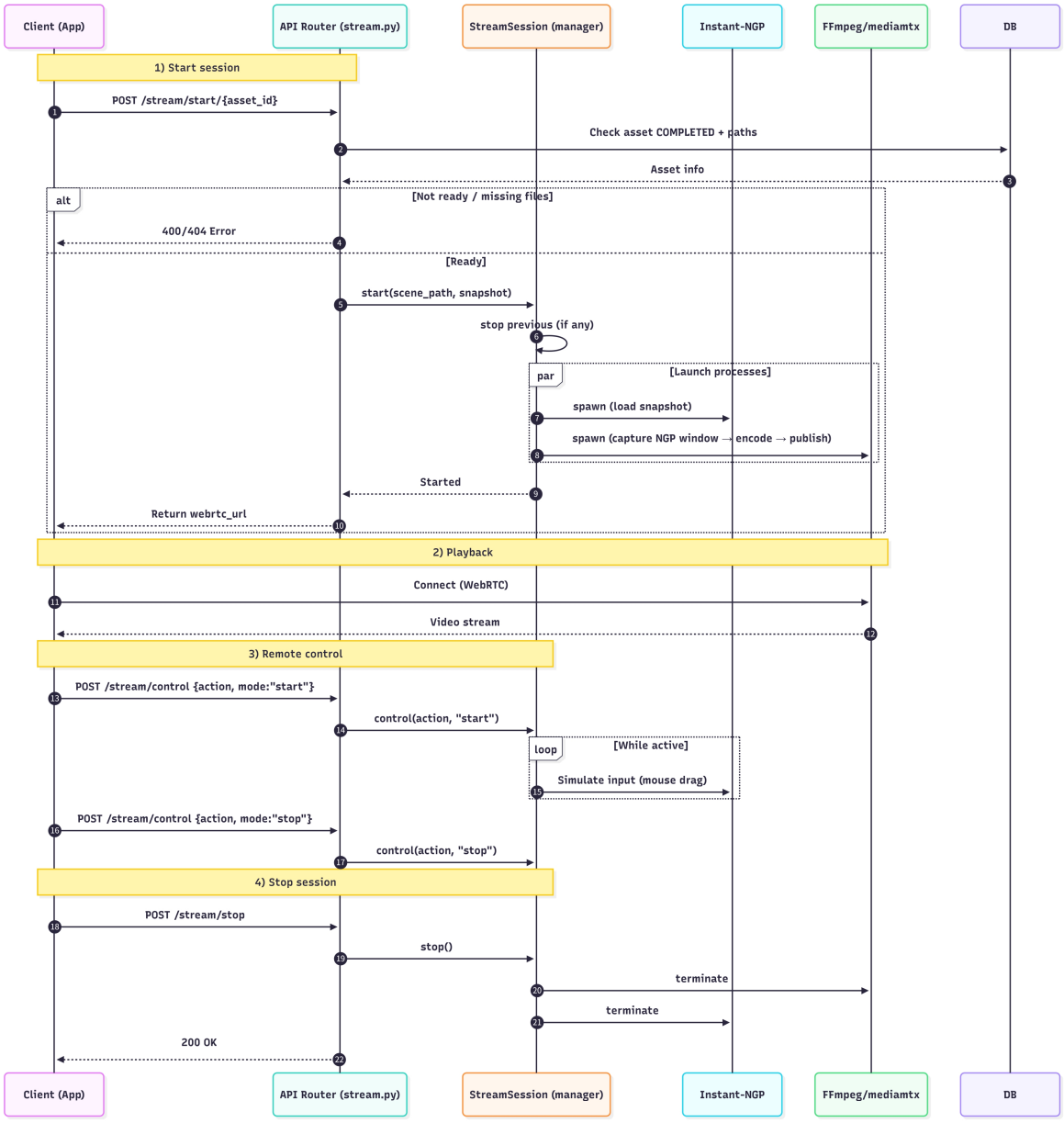

Figure 3-10, Model Preview & Control Sequence Diagram

2) **Interactive Stream Manager**

Class Name: *InteractiveStreamSession* (in *stream_manager.py*)

Responsibility: A global singleton object that manages the lifecycle of the Cloud Rendering Session, orchestrating the Instant-NGP GUI and FFMPEG streaming processes.

Internal Logic:

- **Dual-Process Orchestration:** It spawns two daemon threads: one for *instant-ngp.exe* (rendering the 3D scene) and one for *ffmpeg* (capturing the window).
- **Low-Latency Encoding:** The FFMPEG command is configured with specific flags for performance:
  - **Hardware Acceleration:** Uses *h264_nvenc* to offload encoding to the NVIDIA GPU.
  - **Zero Latency Tuning: C**onfigured with *-tune ll* (Low Latency), *-preset llhq*, and *-zerolatency* to minimize encoding buffer delay.
  - **Capture Strategy:** Uses *gdigrab* to capture the specific window titled "Instant Neural Graphics Primitives" at 30 FPS.
- **Network Awareness:** In *stream.py*, the system detects the request origin. It returns the public IP address if the request comes from the internet (*47.107.xxx.xxx*) or the local IP (*http://{host}:8889/live*) for LAN access, ensuring correct WebRTC signaling paths.

3) **Remote Control System**

Class Name: *ContinuousController* (in *continuous.py*)

Responsibility: Translates RESTful API commands into continuous mouse events on the server, simulating physical user interaction.

Internal Logic:

- **Kinetic Simulation Algorithm:** To prevent jerky camera movements, it implements a *pyautogui.easeInOutQuad* smoothing function. This creates a natural acceleration and deceleration curve for mouse movements (Pan/Rotate).
- **Threaded Action Loop:** When a *mode="start"* command is received, it spawns a dedicated thread that continuously injects mouse deltas (e.g., *pyautogui.moveRel*) until a *mode="stop"* signal triggers the *threading.Event*.
- **Coordinate Clamping:** It calculates the target window's center and bounds (*win32gui.GetWindowRect*), ensuring the mouse cursor is strictly clamped within the NGP window area to prevent accidental clicks outside the renderer.

3.3.3 Real-time Communication Module

This module handles bi-directional communication for the instant messaging (IM) feature.

1) **WebSocket Hub**

   Class Name: *ConnectionManager* (in *socket_manager.py*)

   Responsibility: Manages active WebSocket connections and message broadcasting.

   Internal Logic:

   - **Connection Map:** Maintains an in-memory dictionary *active_connections: Dict[int, WebSocket]* mapping User IDs to socket instances.
   - **Targeted Push:** The *send_personal_message* method looks up the recipient's socket. If online, it pushes the JSON payload immediately; if offline, the message is stored in the database for later retrieval via the HTTP history API.

2) **Hybrid Messaging Strategy**

   Module: *chat.py* (Endpoints)

   Responsibility: Combines HTTP persistence with WebSocket delivery.

   - **Internal Logic:**
   - **Sync-to-Async Bridge:** Since database operations (*save_message_sync*) are synchronous/blocking, the WebSocket loop utilizes *asyncio.get_running_loop().run_in_executor* to execute DB writes in a thread pool, preventing the async event loop from freezing during high-concurrency writes.
   - **Dual-Channel Delivery:** The */chat/send* (HTTP) endpoint not only writes to the DB but also calls *manager.send_personal_message*. This allows non-chat interfaces (like the "Share" button in the Post) to trigger real-time notifications to the recipient.

3.3.4 Infrastructure & Data Access Module

This layer abstracts the underlying database interactions, ensuring data integrity and query efficiency.

1) **Data Access Layer (CRUD)**

   Module: *crud_*.py* (e.g., *crud_post.py, crud_asset.py*)

   Responsibility: Encapsulates complex SQL queries and transaction management using **SQLModel**.

   Internal Logic:

   - **Complex Aggregation:** The *get_community_posts* function performs multi-table logic in a single pass. It fetches posts and simultaneously queries related tables (*InteractionLike, PostCollection, UserFollow*) to determine the

is_liked, is_collected, and *is_following* states for the current user.

- **Atomic Transactions:** Functions like *toggle_like* manage the entire lifecycle within a transaction: checking existence, updating the record, adjusting the *like_count* counter, and committing the change atomically to ensure data consistency.

2) **Database Session Management**

Core Function: *get_session* (in *database.py*)

Responsibility: Provides database sessions to the API layer via a generator pattern.

Internal Logic:

- It utilizes a *yield*-based context manager. A new *Session(engine)* is created for each request and automatically closed after the request finishes, ensuring proper resource cleanup.

3.4 Data Design

3.4.1 Client-Side Data Architecture

We defined and implemented **27 data entity models**.

1) **Core Model Asset Data Models**

Model Assets are the core objects of the system. To optimize network bandwidth, the system employs a "Read-Write Separation" and "Tiered Loading" strategy, defining the following data classes:

Class Name	Description & Usage	Key Fields
AssetCard	**Lightweight model for list views.** Used for waterfall flow displays, it contains only covers and basic metadata to reduce traffic consumption during list loading. It includes a UI property height for generating staggered layout heights.	*id, title, coverUrl, description, ownerId, height, tags, isCollected*
AssetDetail	**Full-detail model for detail model screen**. Contains complete 3D asset data, including source video URLs and generated model file paths for the viewer and player.	*id, title, videoUrl, modelUrl, description, remark, tags, status, createdAt,*

	It also includes the estimated time for timeline calculation.	*estimatedGenSeconds*
AssetUpdateRequest	**Asset update request body**. A Data Transfer Object (DTO) used for users to edit asset information, supporting partial updates.	*title, description, remark, tags*
DownloadResponse	**Download response model**. Encapsulates the temporary download link and filename generated by the backend.	*url, filename*
ReportRequest	**Issue report request body**. Used to submit feedback regarding content violations or system errors.	*category, content*

2) Social & Community Interaction Models

The community module involves complex multi-level data structures. The system uses a **Nested Models** design to aggregate posts, associated assets, user info, and comment lists in a single API request.

Class Name	Description & Usage	Key Fields
PostCard	**Post card model**. Used for community main screen displays, aggregating basic post info with interaction statistics.	*postId, assetId, coverUrl, viewCount, likeCount, ownerName, ownerAvatar, isLiked, isCollected, isFollowing, hasCommented*
AssetDetail	**Post detail aggregation model**. A core composite object nesting *PostAssetInfo* and *CommentOut* lists. It includes the current user's interaction status, such as whether they have liked or collected the post.	*postId, assetId, publishedAt, allowDownload, view/collect/commentCount, ownerInfo, content, visibility, isLiked, isFollowing, asset (Nested), comments (List)*
PostAssetInfo	**Embedded model asset info.** Specifically used for asset display within the post detail page, stripped of non-public management fields.	*id, title, videoUrl, modelUrl, description, remark, tags, status*

PostCreateRequest	**Post creation request body.** Defines the input specifications when a user creates new content.	*assetId, content, visibility, allowDownload*
CommentOut	**Comment output model**. Used to display individual comment content and the publisher's information.	*id, username, avatarUrl, content, createdAt*

3) User Identity & Authentication Models

To protect user privacy, the system strictly distinguishes between "Private Profiles" and "Public Profiles."

Class Name	Description & Usage	Key Fields
UserDetail	**Private user profile.** Used exclusively for the "Profile" page, containing complete personal details and private statistics.	*id, username, gender, avatarUrl, coverUrl, bio followerCount, followingCount, likedTotalCount*
UserOut	**Public user profile**. Used for follower/following lists and comment sections, removing sensitive statistics while retaining public-facing info.	*id, username, gender, avatarUrl, coverUrl, bio*
RegisterResponse	**Registration Output.** Returned by the server upon successful account creation, confirming the new user's basic identity.	*id, username, avatarUrl*
LoginResponse	**Authentication response.** Contains the JWT token used for identity verification in subsequent API requests.	*accessToken, tokenType*
UserUpdate	**Profile update request.** All fields are nullable to support incremental updates.	*username?, bio?, gender?*

4) Real-Time Streaming & Control Protocols

For cloud rendering interactions, the system utilizes a RESTful HTTP protocol to manage the session and send control signals. This design decouples the control layer from the video transmission layer (WebRTC).

Class Name	Description & Usage	Key Fields
ControlCommand	**Control payload object.** Encapsulates the user's interaction intent as a JSON body sent via HTTP POST. It supports a "press-and-hold" interaction model by distinguishing between "start" and "stop" modes.	*action (Enum), direction (Enum), mode ("start"/"stop")*
StreamStatus	**Session status response.** Returned by the backend when initializing a stream, containing the WHEP URL required by the player.	*isActive, webrtcUrl, currentAssetId*
StreamActionType	**Action Enumeration.** Strictly defines valid operations to prevent invalid API calls. Maps directly to server-side logic.	*ROTATE, PAN, ZOOM*
StreamDirection	**Direction Enumeration.** Defines the coordinate vectors for camera movement in the 3D space.	*UP, DOWN, LEFT, RIGHT, IN, OUT*

5) Chat & Messaging Models

The chat module adopts a bi-directional data flow design utilizing **WebSockets** for real-time latency.

Class Name	Description & Usage	Key Fields
ChatMessage	**Message Entity.** Represents a single chat record stored in the database and rendered in the UI, including a read status flag.	*senderId, receiverId, content, isRead, createdAt*
ChatConversation	**Conversation List Item.** A summary model for the chat list, displaying the contact person along with the latest message snippet and unread count.	*username, lastMessage, unreadCount, avatarUrl*
WSMessageSend	**WebSocket Outgoing Frame.** The payload structure used to	*receiverId, content*

	send real-time text messages from client to server via the WebSocket channel.	
WSEvent	**WebSocket Incoming Frame.** A generic envelope for handling server-push events, allowing the client to distinguish between different event types (e.g., "new_message").	*type, data (Nested ChatMessage)*

6) UI State Management (Sealed Classes)

The front-end architecture utilizes **Sealed Classes** to implement a Finite State Machine (FSM), ensuring the UI remains in a single, deterministic state at any given time.

General Loading States (Patterns used in *DetailUiState, PostUiState*):

- *Loading*: Triggers a loading animation/skeleton screen.
- *Success(data)*: Carries the strongly-typed data payload, such as *AssetDetail* or *PostDetail*, ensuring type safety during rendering.
- *Error(msg)*: Carries exception information to display user-friendly error messages or retry buttons.

Complex Business States:

- ***UploadState*:** Beyond standard states, it includes an *Idle* state and works alongside *_resourceError* in *UploadViewModel* to handle local pre-validation logic for file size (>50MB) and duration (>60s) before any network request is made.
- ***StreamUiState*:** Includes a specialized *Streaming(url)* state. This ensures the WebRTC player component is only initialized and mounted after the backend has successfully provisioned the WHEP url.
- ***DownloadEvent*:** Defined as a sealed class with *Success* and *Error* subtypes. Unlike UI states, these are emitted via a *SharedFlow* to handle "one-shot" effects like Toasts, preventing duplicate notifications upon screen rotation.

Component Configuration Data:

- ***BottomNavItem***: A sealed class that defines the static routing structure for the bottom navigation bar. It binds route strings ("*home*", "*community*") to vector icon resources, allowing the *BottomNavBar* component to render tabs dynamically based on this data model.

3.4.2 Database Schema (Server-Side Design)

The server-side data persistence layer is constructed using **SQLModel** (a combination of SQLAlchemy and Pydantic), mapping Python object-oriented models directly to a relational database (currently configured for SQLite, compatible with PostgreSQL). The database design adheres to the Third Normal Form (3NF) to minimize data redundancy while strategically employing denormalization for performance critical read operations. **All 10 tables' definitions are written in *models.py***.

1) User Identity & Social Graph

The *User* table serves as the central entity of the system, managing authentication credentials and personal profiles. The social graph is implemented via self-referential many-to-many relationships.

- **User Entity (*User* Table)**:
  - **Identity:** Uses *username* (Unique Index) and *password_hash* for secure authentication.
  - **Profile Data:** Stores demographic data using strict Enum constraints (*Gender*: Male, Female, Other, Secret) to ensure data consistency.
  - **Performance Optimization (Denormalization):** To optimize high-frequency read operations on the "Profile" page, statistical counters (*follower_count, following_count, liked_total_count*) are cached directly in the *User* table. This avoids expensive *COUNT(*)* aggregation queries on the association tables during standard page loads.
- **Social Connections (*UserFollow* Table)**:
  - A pure association table implementing the Many-to-Many relationship between users.
  - **Composite Primary Key:** defined as (*follower_id, followed_id*), which enforces a unique constraint at the database level, preventing a user from following the same target twice.

2) Model Asset Management & Lifecycle

The ***ModelAsset*** table represents the core 3D content. The design strictly separates the "Input" (Source Video) from the "Output" (Generated Model).

Column / Feature	Description & Design Intent
Asset State Machine	The *status* column uses the *AssetStatus* Enum (*pending, processing, completed, failed*) to track the asynchronous training lifecycle of the NeRF model.

Path Separation	Distinct columns for *video_path* (upload source) and *model_path* (training result) allow the system to maintain a link to the original footage even after the 3D model is generated.
Metadata Storage	The *tags* field utilizes a *JSON* column type. This allows for schema-less flexibility, enabling users to add an arbitrary number of descriptive tags without altering the table structure.
UI Hints	The *height* field is stored permanently to support the "Staggered Grid Layout" (Waterfall flow) on the frontend, preventing layout shifts when loading images. It also includes the *estimated_gen_seconds* for the frontend timeline calculation

3) Community Content & Distribution

The system employs a "Wrapper Pattern" to transform a private *ModelAsset* into a public *CommunityPost*.

- ***CommunityPost* Table:**
  - Acts as the social container for an asset. It references *asset_id* (Foreign Key) but maintains its own *content* (text) and *visibility* settings (*PUBLIC, FOLLOWERS, PRIVATE*).
  - **Decoupling:** This design allows a user to delete a post without deleting the underlying 3D asset, or to re-post the same asset with different context.
  - **Interaction Stats:** Similar to the User table, *CommunityPost* maintains cached counters for *view_count, like_count, collect_count,* and *comment_count* to support rapid feed rendering.

- **Interaction Tables (*InteractionLike, PostCollection*):**
  - These join tables manage user interactions. *InteractionLike* records a timestamp *created_at* to potentially support "Trending" algorithms based on recent activity.

4) Communication & Auditing

- **Direct Messaging (*Message* Table):**
  - Stores chat history with a simple dual-index design on *sender_id* and *receiver_id*.
  - Includes an *is_read boolean* flag to support the "Unread Message Badge"

logic in the UI.

- Audit Trail (*DownloadRecord* Table):
  - Tracks which user downloaded which asset via *(user_id, asset_id)*. This table is essential for analytics (calculating hot resources) and security auditing.

5) Data Integrity & Constraints

The schema enforces integrity through several mechanisms:

- **Foreign Keys:** All relationships (e.g., *owner_id* in *ModelAsset, user_id* in *Comment* table) utilize strict Foreign Key constraints to prevent orphan records.
- **Enum Constraints:** Fields like *Visibility, AssetStatus,* and *Gender* are restricted to specific Enum values, preventing invalid state injection from the API layer.
- **Cascade Logic:** The SQLModel relationships (e.g., *back_populates*) are configured to ensure that object-relational mapping handles data loading consistencies automatically.

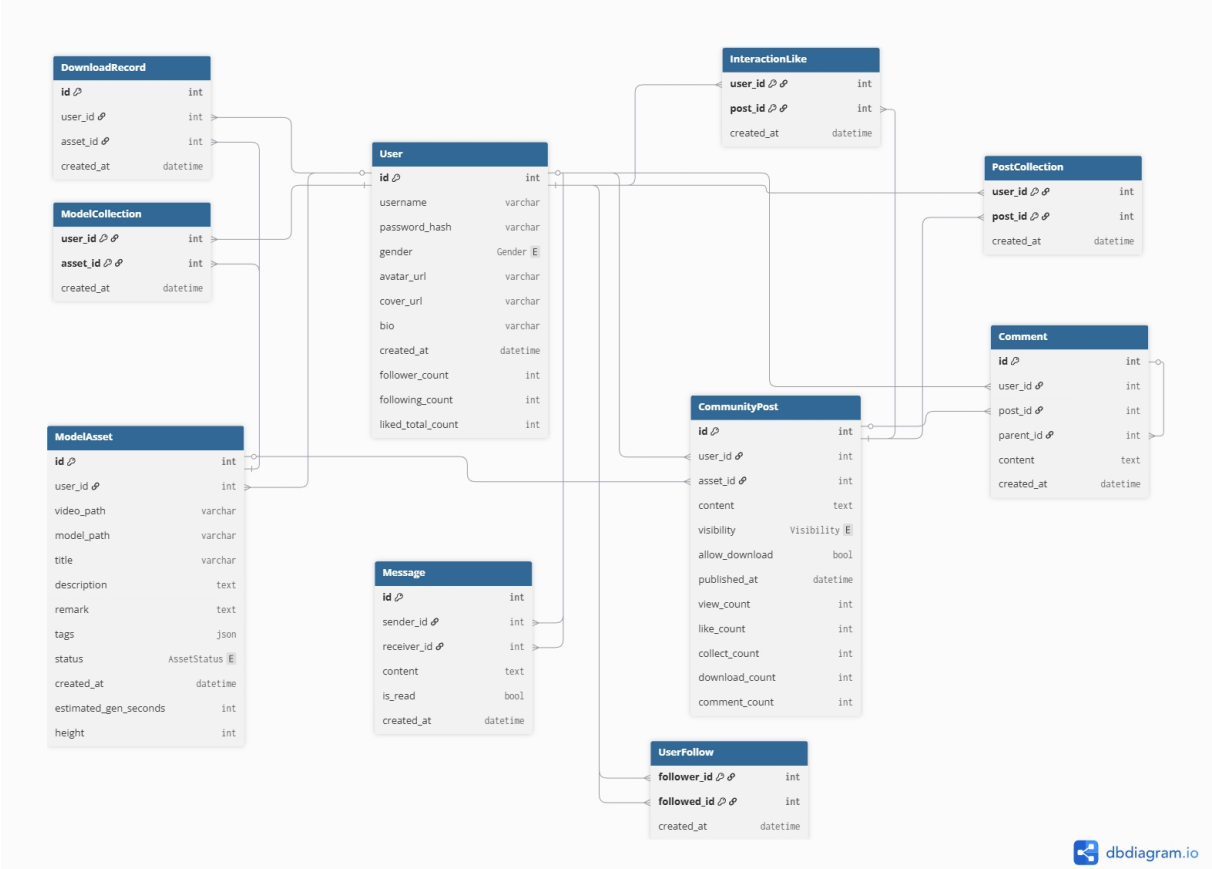

Figure 3-11, the Entity–Relationship Diagram of the server-side database schema

3.5 System Interface Design

The system adopts a decoupled frontend and backend architecture. Communication is primarily based on **HTTP/1.1 (RESTful API)**, with **WebSockets** utilized for specific real-time services. The backend is developed using the **FastAPI framework**, while

the frontend encapsulates network requests via **Retrofit**. The interface design follows the principles of high cohesion and low coupling, ensuring data transmission security and consistency. The following includes all **35 implemented APIs.**

3.5.1 Design Principles and Specifications

1) **RESTful Architectural Style**: All resources (Users, Assets, Posts) are manipulated through standard HTTP methods (GET, POST, PATCH, DELETE).
2) **Unified Authentication Mechanism**: **JWT (JSON Web Token)** is used for stateless identity verification. Except for registration and login interfaces, all requests must carry the ***Authorization: Bearer <token> header***.
3) **Data Interaction Format**: The default format for request and response bodies is ***application/json***. File upload interfaces utilize the ***multipart/form-data*** format.
4) **Standardized Responses**: Interfaces follow unified HTTP status code specifications:
   - **200 OK**: Request successful.
   - **400 Bad Request**: Parameter validation failed (e.g., username already exists).
   - **401 Unauthorized**: Token expired or invalid.
   - **403 Forbidden**: Insufficient permissions (e.g., attempting to delete an asset owned by another user).
   - **404 Not Found**: Resource does not exist.

3.5.2 Authentication and User Management Interfaces

This module manages the user lifecycle, including registration, login, and profile maintenance.

Interface Name	HTTP Method	Endpoint	Description
User Registration	**POST**	/api/v1/auth/register	User Registration
User Login	**POST**	/api/v1/auth/login	JWT Login (OAuth2 Form)
Get My Profile	**GET**	/api/v1/users/me	Fetch Current User Details

Update Profile	**PATCH**	/api/v1/users/me	***UserUpdate*, update Profile Metadata**
Update Avator	**POST**	/api/v1/users/me/avatar	Upload/Update Profile Avatar
Update Cover	**POST**	/api/v1/users/me/cover	Upload/Update Homepage Cover
Follow/Unfollow	**POST**	/api/v1/users/{id}/follow	Path: target_user_id
User Follower	**GET**	/api/v1/users/{id}/followers	List User's Followers
User Following	**GET**	/api/v1/users/{id}/following	List User's Following

Detailed Specification (Key User Interfaces)

- **Social Toggle (*follow_user*)**:
  - **Logic**: An atomic operation that adds a follow record or deletes it if it already exists.
  - **Response**: *ToggleResponse* {is_active, new_count} (if the user is now followed and returns latest follower count).
- **Media Update (*update_my_avatar*)**:
  - **Request**: *multipart/form-data* (file).
  - **Process**: Generates a unique filename using UUID and user ID, saves to *static/uploads/avatars/*, and updates the database record.

3.5.3 Core Model Asset Management Interfaces

This module handles the upload of private 3D assets, scheduling of training tasks, and file downloads.

Interface Name	HTTP Method	Endpoint	Description
Model Asset Upload and Creation	**POST**	/api/v1/assets/upload	Returns an ***AssetCard***, allowing the client to use the "Processing" state in the list.
Get My Assets	**GET**	/api/v1/assets/me	Returns ***List[AssetCard]*** sorted by creation time in descending order.

Asset Detail	**GET**	/api/v1/assets/{id}	Returns ***AssetDetail***. Includes the private remark field; only accessible by the owner.
Asset Collection	**POST**	/api/v1/assets/{id}/collect	Toggle Model Asset Collection
Collected Assets	**GET**	/api/v1/assets/me/collected	List Collected Assets
Download Asset	**POST**	/api/v1/assets/{id}/download	Request Model Download URL
Download History	**GET**	/api/v1/assets/me/downloads	Fetch Download History
Issue Report	**POST**	/api/v1/assets/{id}/report	Submit Asset Issue Report

Detailed Specification:

- **Download Management:**
  - **Logic:** Validates model completion before logging a ***DownloadRecord*** and returning a specific file path based on *file_type* (obj, glb, ply, msgpack).
- **Reporting (*report_issue*)**:
  - **Process**: Logs reports to *asset_reports.log* with timestamp, user ID, and category for administrative auditing.

3.5.4 Community and Social Interaction Interfaces

This module enables the transition from "Private Assets" to "Public Posts" and facilitates user interaction.

1) Post Publishing

- **Endpoint:** POST /api/v1/posts/publish
- **Validation Logic:**
  a) Verifies the asset belongs to the current user.
  b) Ensures the asset status is *COMPLETED* (unready models cannot be published).
  c) Prevents duplicate publishing (via *get_post_by_asset_id*).
- **Data:** Accepts ***PostCreate*** (*asset_id, content, visibility, allowDownload*).

2) Feed and Interactions

Interface Name	HTTP Method	Endpoint	Description
Community Feed	**GET**	/api/v1/posts/community	Aggregated query. Returns all public posts mixed with the user's interaction states (*is_liked, is_collected*) to reduce extra requests.
Like Post	**POST**	/api/v1/posts/{id}/like	Toggles state. If liked, it removes the like; otherwise, it adds it. Synchronously updates *User.liked_total_count*.
Post Comment	**POST**	/api/v1/posts/{id}/comments	Writes to the comment table and increments *post.comment_count*.
Post Detail	**GET**	/api/v1/posts/{id}	Automatically increments *view_count*. Returns an aggregated object with nested comments and asset info.
My Post	**GET**	/api/v1/posts/users/me/posts	List all my public posts
User Post	**GET**	/api/v1/posts/users/{id}/posts	List a specific user's posts
Collect Post	**POST**	/api/v1/posts/{id}/collect	Toggle post collection of specific post
Post Collections	**GET**	/api/v1/posts/me/collected	List my collected posts

3.5.5 Real-Time Services (Streaming & Messaging)

1) Cloud Rendering Control

Uses HTTP to manage the state of the rendering session; the video stream is handled

via WebRTC.

- **Start Session**: POST /api/v1/stream/start/{asset_id} - Validates model path, identifies network (Public vs. Local IP), and initiates the renderer.
- **Stop Session**: POST /api/v1/stream/stop - Terminates the active rendering instance.
- **Camera Control**: POST /api/v1/stream/control - Uses *ControlCommand* with *mode="start/stop"* to simulate continuous rotation, panning, or zooming.

2) Instant Messaging (Chat System)

A hybrid model using WebSockets for real-time delivery and HTTP for persistence.

- **WebSocket Channel**: WS /api/v1/chat/ws/{user_id} - Pushes real-time *new_message* events to the recipient.
- **Chat History**: GET /api/v1/chat/history/{other_id} - Paged retrieval of past messages between two users.
- **Conversation Summary**: GET /api/v1/chat/conversations - Aggregated list of active chats with unread message counts.
- **Read Management**: POST /api/v1/chat/conversations/{id}/read - Marks all messages from a specific sender as "read".
- **HTTP Send**: POST /api/v1/chat/send - Standard HTTP fallback for sharing post cards or offline messaging.

4. User Interface Design

4.1 UI Overview

The design of our application's user interface focuses on simplicity and ease of use, ensuring that users can quickly get started and enjoy a smooth experience. The UI design philosophy revolves around **user experience (UX)** and **functional usability (UI)**. Through simple layouts and clear functional module separations, users can efficiently complete tasks without confusion. In terms of color selection, we use low-saturation backgrounds to create a comfortable and clear visual effect. This color scheme effectively reduces visual fatigue and enhances the app's professional and technological feel.

In **Stage Two of the design**, we paid particular attention to the clear layout of each functional page, ensuring that functional areas are reasonably separated, allowing users to easily identify and click the required functions. The homepage uses a **card-based waterfall layout**, making it easy to display and filter different types of 3D models, while supporting interactive actions to show more details. Each card is designed with **minimalistic text,** ensuring that the information is clear and concise. A unified **bottom navigation** and **humanized icon design** enhance user convenience, providing a **sense of space and immersion** across the interface. In addition, the icon design and text layout have high contrast, improving the intuitive nature of user operations.

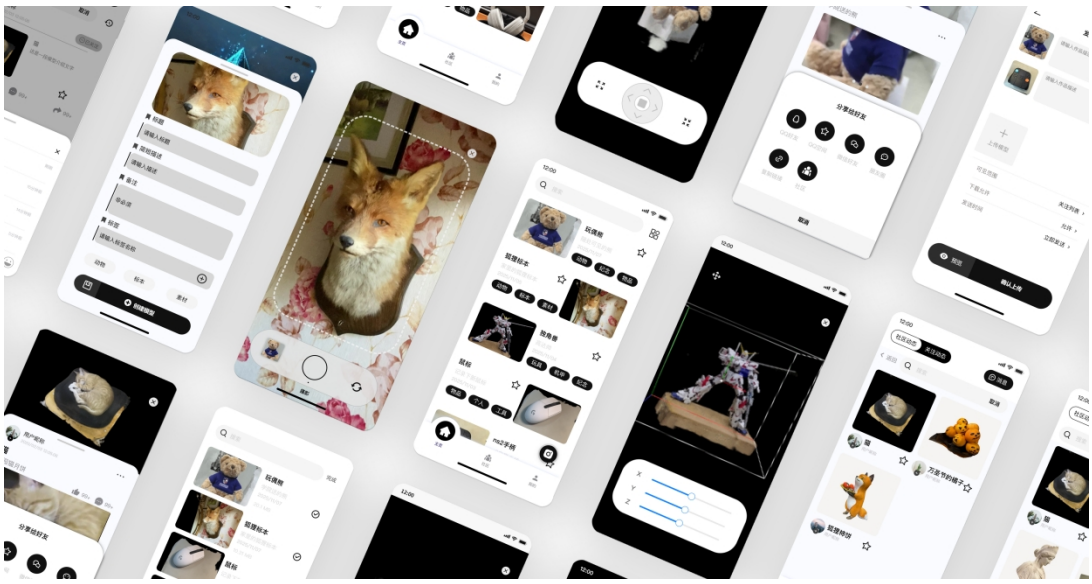

Figure 4-1, UI prototype overview

We incorporated **rounded corners** for the cards, adding a soft and modern aesthetic to the design. This feature not only enhances the visual appeal but also aligns with the overall minimalistic and friendly user interface design principles. Additionally, a smooth **responsive layout** was implemented to ensure the interface adjusts seamlessly

to various screen sizes, optimizing the user experience across different devices.

Following the **minimalistic style** design principle, the main color is **low-saturation dark color (#0C1427)** paired with **large areas of neutral tones (#89898E)**, ensuring the interface is both simple and rich in depth. The background color uses **#F5F5F5,** maintaining a clean visual feel. The layout of the interface is clear, with attention focused on the 3D models shot by the user, and the clean neutral background provides a simplified layout, making every detail stand out.

The **Modao prototype design platform** was used to design and display the structure and interaction effects of each interface, ensuring that each design is precisely implemented. During the UI design process, we also created detailed **user flow diagrams (Userflow)** to showcase the steps users take within the app. Additionally, **UI isometric diagrams** helped further clarify the hierarchy of interface elements and structures, providing a more intuitive view.

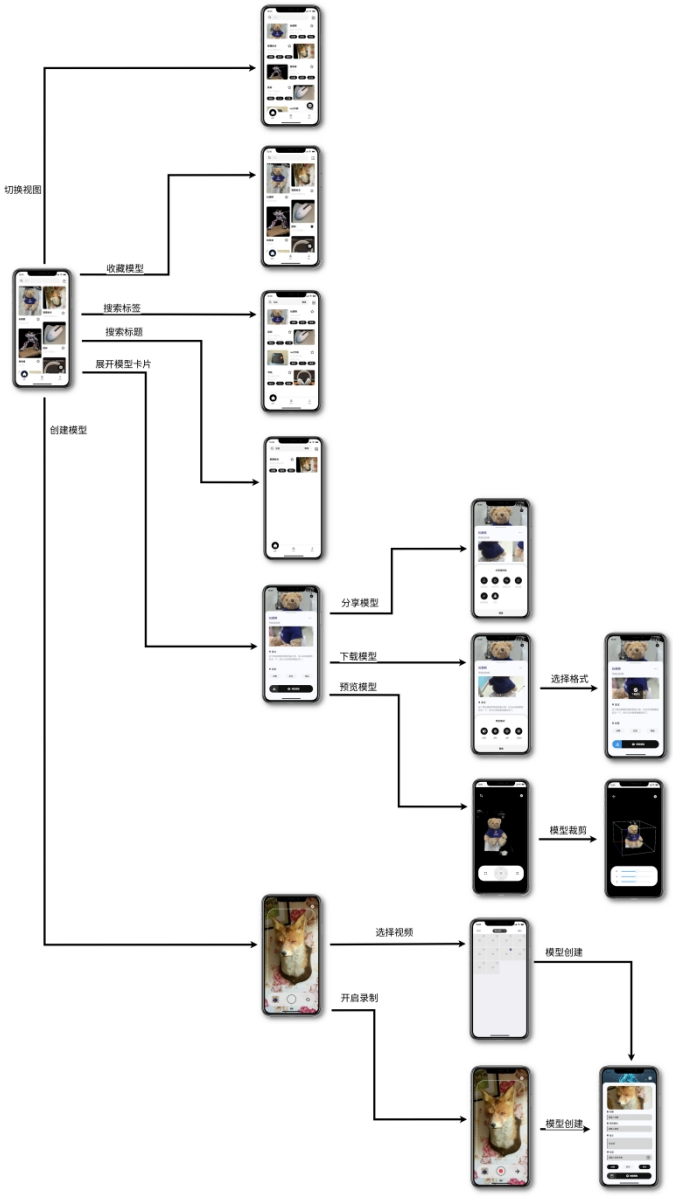

Figure 4-2, user-flow chart

Figure 4-2 clarifies what users can accomplish starting from the home screen: switching feeds and favorites, searching by tags/titles, expanding model cards, and then moving into detail actions such as previewing, downloading, sharing, and selecting export formats. It also captures the modeling pipeline from video selection/recording to model creation. In software design, this figure defines screen entry/exit points, highlights core conversion paths (e.g., Home → Detail/Preview → Export/Share), and acts as a blueprint for test-case design—each transition can be mapped to functional and scenario-based tests.

In the **third stage of optimization,** UI design was further enhanced in terms of color, layout, and interaction logic. We officially established the following color scheme for the app: **Primary color (60%): Dark Green (#1A323A); Secondary color (30%): White (#FFFFFF); Accent color (10%): Cyan (#62FFD8).**

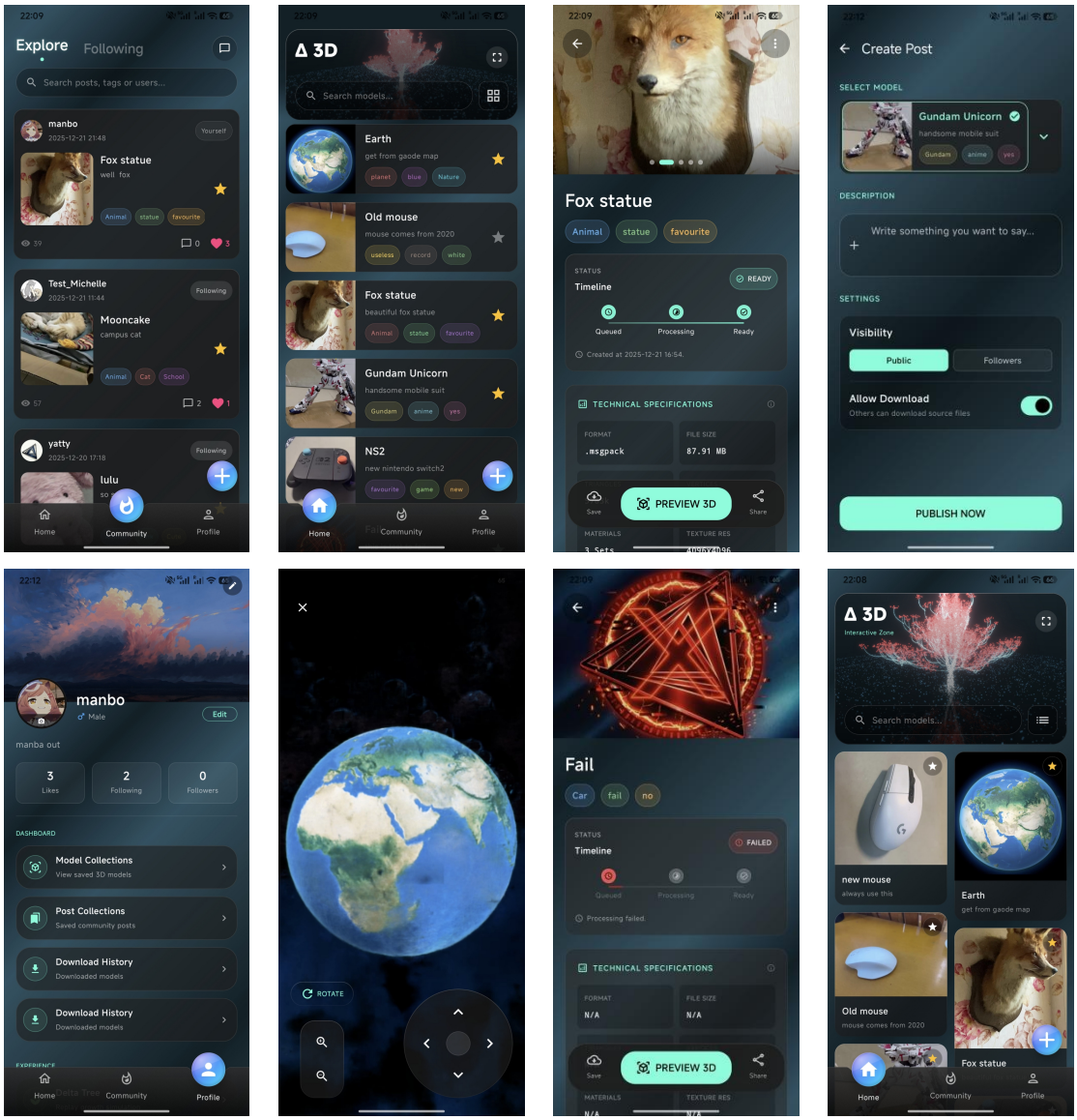

Figure 4-3, UI design snapshots

This combination of dark green for primary elements, white for background and secondary components, and cyan for emphasis, creates a balanced and visually appealing interface. The transparent background colors, combined with this palette, provide a visual hierarchy while maintaining a minimalist style. The visual and technological feel of the interface was further emphasized. We optimized the icons and buttons to ensure smooth user operations and responsive interactions. We also focused on adaptability across different devices, ensuring that the interface displays consistently regardless of the device used. To improve the visual experience, we **store card coordinates in the database** to ensure that the card waterfall layout on the homepage and community page maintains the same height with each refresh. To enhance visual comfort and brand recognition, dark green is systematically applied to key components, effectively reducing visual fatigue.

4.2 UI Prototypes

In our UI prototypes, we focused on demonstrating the interaction and layout of the key functional pages. The **login page** is simple, with input fields and buttons that are clear and easy to use, ensuring that users can quickly enter their account information and log in. The button color contrasts sharply with the background, making it easy for users to locate the action points. The **homepage** uses large images and a card-based layout to highlight the app's core functions, allowing users to easily browse different categories of 3D models. Through simple swipe and click actions, users can interact with each model and view more detailed information.

For the **personal settings page,** clear divisions between functional areas help users operate efficiently. Each setting label is visible, allowing users to quickly find the functions they need, reducing operational complexity. For the **3D model preview page,** we specifically optimized interactions for rotation, zooming, and panning, allowing users to intuitively manipulate the model and observe real-time changes. At the same time, we added more interactive feedback to the **back and history buttons** to help users easily trace their previous steps.

During the **optimization,** we achieved significant improvements in system-level interactions and visual experiences:

- **App-wide entrance animation:** A unique opening experience was designed for the entire app. After the first installation and launch, users will see an entrance animation composed of a **Three.js** simulated point cloud particle tree and the delta3D logo, creating a strong first impression. Afterward, the animation automatically hides when users return to the app, with the option to trigger it again through the search bar or personal page.

- **Performance and navigation optimization:** By introducing the **Webview**

thread pool and unified routing management mechanism, we significantly reduced loading time during page transitions, making navigation as smooth as native apps. At the same time, we designed a **search bar that follows on scroll,** which expands to show the dynamic particle tree when users scroll up and makes it easy to search when they scroll down, balancing both functionality and interactive enjoyment.

These optimizations made the user's operation more intuitive, with each interaction responding instantly, significantly increasing overall user satisfaction.

4.3 User Experience Feedback

During the UI design process, we gathered user feedback to further refine the interface and interaction design. Below are two examples of feedback points regarding UI/UX design and the corresponding improvements:

- **Unclear progress bar display:** Users reported that the progress bar lacked clear indicators during uploads and processing, making the waiting time uncertain. In response, we added “estimated upload time” and percentage display, along with timeout warnings and notifications to reduce uncertainty during long waits.

- **Low label color contrast:** In the list view, some label colors were too similar, making them hard to distinguish. We improved the label color palette by increasing the contrast, making it easier for users to differentiate between labels.

These improvements have significantly enhanced the app's usability and user satisfaction, ensuring that the interface design better meets user needs and continues to improve based on ongoing feedback. Through agile responses to user feedback, we ensure that the UI design remains efficient and user-friendly.

5. Key Technologies

5.1 Authentication and Authorization

The Δ3D system implements a stateless authentication architecture based on **JSON Web Tokens (JWT)** and the **OAuth2** password flow standard. This approach ensures scalability and security by eliminating the need for server-side session storage while maintaining secure access to protected resources. Meanwhile, the frontend utilizes mechanisms to minimize the number of user logins as much as possible to optimize the user experience.

5.1.1 Backend Security Architecture

The server-side authentication is built using the FastAPI security module. The system utilizes *OAuth2PasswordBearer* to handle token retrieval from the request headers.

- **Password Hashing:** To ensure data integrity and security, user passwords are never stored in plaintext. The system employs **Bcrypt** hashing via the *passlib* library. When a user attempts to register or login, the backend verifies the provided credentials against the stored hash using *verify_password* and generates a secure hash using *get_password_hash*.
- **Token Generation:** Upon successful credential verification, the server generates a JWT signed with the ***HS256*** algorithm and a secret key defined in the application configuration. The payload includes the subject (user ID) and an expiration timestamp, currently set to 30 days to enhance user experience by reducing login frequency.
- **Dependency Injection:** Route protection is implemented using FastAPI's dependency injection system. The *get_current_user* dependency automatically extracts the token, decodes it, and validates the user existence before granting access to protected endpoints like /api/v1/assets/upload.

5.1.2 Client-Side Implementation

The Android client manages authentication states using a reactive architecture powered by **Jetpack DataStore** and **Kotlin Coroutines**.

- **Secure Persistence:** The ***TokenStore*** persists the access token using Android's *Preferences DataStore* via coroutine-based APIs (Flow and suspend functions), replacing legacy *SharedPreferences* with a more consistent and safer key-typed storage approach.
- **Reactive State Management & Auto-Restoration:** The ***SessionViewModel*** acts

as the single source of truth for authentication. Upon application initialization, the ***SessionViewModel*** executes an automatic restoration logic within its *init* block. It asynchronously retrieves the persisted token from ***TokenStore***. If a valid token is detected, the system immediately restores the in-memory session state and triggers a background *fetchCurrentUser* request to synchronize the latest user profile. This ensures that returning users are seamlessly authenticated and their information is up-to-date without manual intervention.

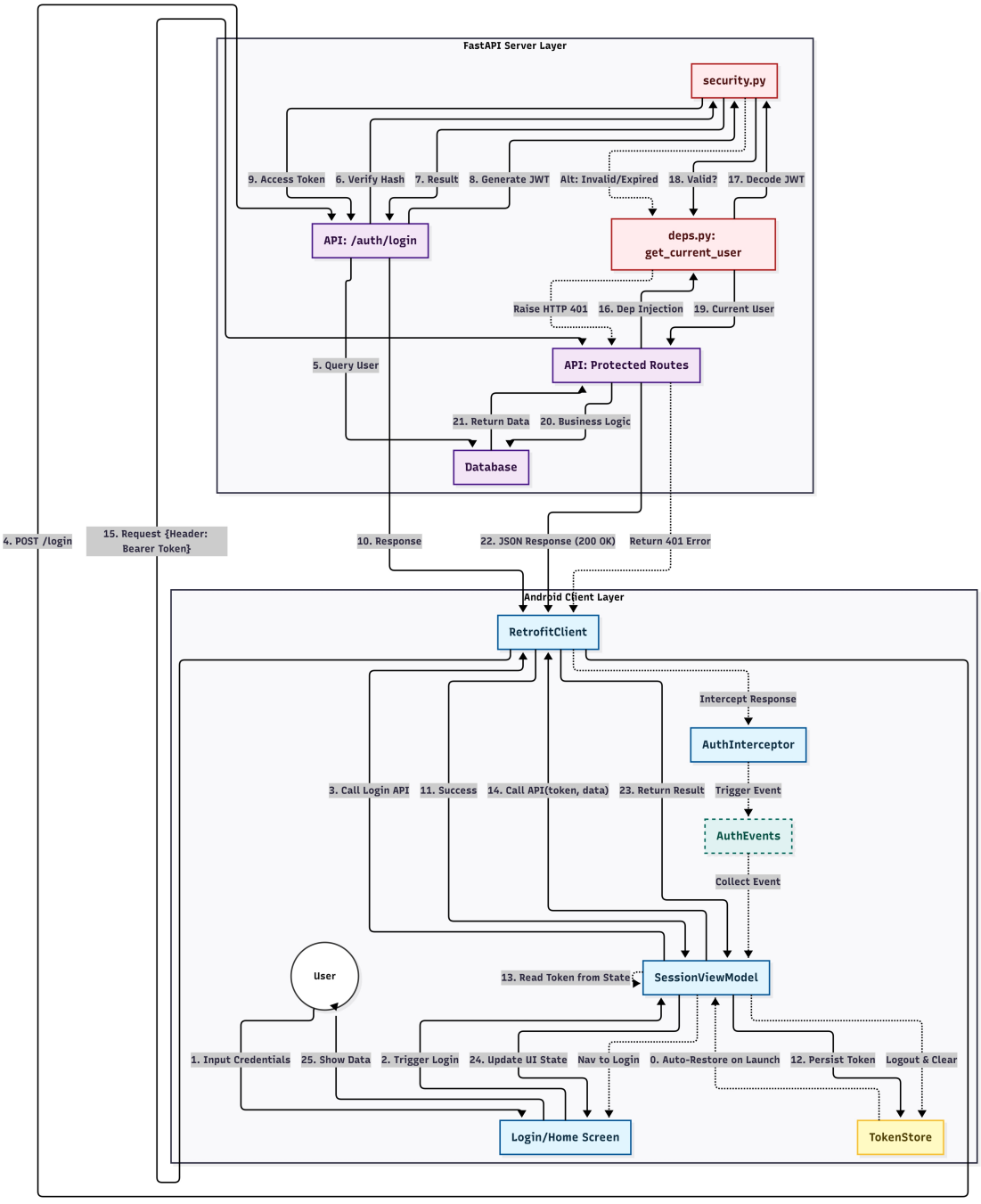

Figure 5-1, Authentication & Authorization Flowchart

- **Explicit Header Propagation:** Unlike implicit interception strategies, the system employs an explicit header passing mechanism. The ***ApiService*** interface defines the *@Header("Authorization")* parameter for each protected endpoint. Business logic components subscribe to the session state, format the token with the standard "Bearer " prefix, and explicitly pass it to the network layer. This maximizes code transparency and ensures network requests are strictly coupled with a valid authentication state.
- **Reactive Error Handling:** The system implements a Global Unauthorized Handler using a shared event bus (***AuthEvents***). If the backend returns a **401 Unauthorized error**, the event propagates to the ***SessionViewModel***, which automatically clears the local session and redirects the user to the login screen, completing the security lifecycle.

5.2 Key UI and Performance Optimization

To deliver a responsive and visually stable user experience, our Δ3D application implements specific optimization strategies for data rendering, search interaction, and visual consistency.

5.2.1 Local Search & Debouncing

For *HomeScreen* and *CommunityScreen*, since the application pre-loads model assets and posts data for display, the search function is implemented as a purely client-side filtering mechanism, eliminating the latency associated with network-based search.

- **Local Filtering:** Instead of querying the server for every search term, the app filters the locally cached *_allAssets* and *_allPosts* list. This provides instantaneous feedback to the user and allows for complex filtering logic (e.g., matching against titles and tags simultaneously) without server overhead.
- **Coroutines Debouncing:** Despite being local, filtering a large dataset can still impact UI frame rates if triggered on every keystroke. A **600ms** debounce delay (implemented via Kotlin Coroutines delay() and Job cancellation) ensures that the filtering logic only executes when the user pauses typing. This prevents unnecessary recomposition cycles and ensures the UI thread remains smooth.

5.2.2 Deterministic Layout Rendering

To ensure fluid scrolling and minimize memory footprint, the application employs a highly optimized rendering strategy:

- **On-Demand Lazy Loading:** Utilizing the ***Coil*** (Coroutine Image Loading) library, image resources are requested asynchronously only when they enter the viewport. Integrated within *LazyVerticalStaggeredGrid* and *LazyColumn*, this

ensures that off-screen images do not consume bandwidth or memory, significantly improving the startup speed and responsiveness of long lists.

- **Visual Continuity:** The *crossfade(true)* parameter is enabled for image requests, providing a smooth transition as images materialize, eliminating jarring "pop-in" effects.
- **Layout Stability:** As mentioned previously, by reserving layout space using pre-computed ***height*** metadata from the database, the lazy-loading process occurs within fixed bounds, effectively preventing Cumulative Layout Shift (CLS) even as images load asynchronously.

5.2.3 Intelligent Visual Consistency (Tag Color System)

To maintain visual coherence in the tagging system without hardcoding infinite color variations, a custom **Tag Color Binder** algorithm was developed.

- **Stateful Palette Mapping:** The system utilizes a stateful map (*mutableStateMapOf*) to bind specific tags to colors from a curated *TagPalette* during the session, which ensures that the same tag can have the same color.
- **Collision Avoidance:** If the number of tags exceeds the palette size, the system dynamically generates new colors using HSL (Hue, Saturation, Lightness) calculations. By shifting the hue by approximately 137.5 degrees (Golden Angle) for each new tag, the system maximizes visual distinction between adjacent tags and prevents color collisions, ensuring text remains legible and aesthetically pleasing.

5.2.4 Optimistic UI Updates

To bridge the gap between user action and server response, **Optimistic UI** patterns are applied to interaction-heavy features like "Collect," "Like," and "Follow."

- **Immediate Feedback:** When a user interacts with an item (e.g., clicking the Star icon), the local model state is updated instantaneously (*_allAssets.map { ... }* and *_allPosts.map{...}*), providing immediate visual confirmation.
- **Background Synchronization:** The network request is sent in the background. If the API call fails, a rollback mechanism reverts the local state to its previous value. This decoupling makes the app feel significantly faster than the actual network latency would allow.

5.2.5 Complex Scroll Interactions

The Home Screen features a collapsible interactive header (*HomeTreeCard*) that reacts to the user's scroll gestures, implemented via a custom *NestedScrollConnection*.

- **Physics-Based Interaction:** The connection intercepts scroll events (*onPreScroll* and *onPostScroll*) to calculate a *heightOffsetPx*. This dynamically adjusts the header's height and the alpha transparency of internal elements (like the 3D *webview* overlay) based on the scroll direction.
- **Smooth Transition:** The implementation ensures that the refresh gesture (Pull-to-Refresh) does not conflict with the header collapse animation, providing a seamless native-feeling scroll experience.

5.3 3D Model Generation and Training Pipeline

The core functionality of the Δ 3D application relies on a robust pipeline that transforms user-uploaded 2D videos into navigable 3D scenes. This section details the technical implementation of the upload validation, asynchronous task scheduling, and the neural rendering workflow using Instant-NGP and COLMAP.

5.3.1 Client-Side Video Pre-processing and Validation

To ensure high success rates in training and optimize server resource usage, the Android client implements strict pre-validation logic before a file is transmitted.

- **Metadata Extraction**: Utilizing the Android *MediaMetadataRetriever* and *ContentResolver,* the application extracts critical video properties locally. Specifically, the system validates the file size and duration.
- **Resource Constraints:** To adhere to server computing limits, the client enforces a maximum file size of **50MB** and a maximum duration of **60 seconds**. If the video exceeds these thresholds ("Limit: 50MB" or "Limit: 60s"), the upload action is blocked at the UI level to prevent unnecessary network traffic.
- **Time Estimation Algorithm:** The client predicts the server-side processing time to calculate the timeline in *AssetDetailScreen* and improve user experience. The estimation algorithm is defined as $T_{total} = T_{base} + \left(S_{file}/4.0\right) \times 15.0$, where T_{base} is a fixed initialization overhead (30 seconds) and S_{file} is the file size in MB. This calculated value is sent to the backend alongside the file.

5.3.2 Asynchronous Task Scheduling and State Management

Given that 3D model training is a compute-intensive process that cannot be handled within a synchronous HTTP request cycle, the backend employs an asynchronous architecture using FastAPI's *BackgroundTasks*.

- **Task Delegation:** Upon receiving a POST /api/v1/assets/upload request, the server immediately saves the raw video file into a dedicated directory named with a unique UUID. It creates a database record with an initial status of *PENDING*

and returns a response to the client immediately, preventing timeout errors.

- **State Machine:** A background worker (*task_train_asset*) takes over the execution. The asset status transitions through a strictly defined lifecycle:
  a) **PENDING:** File uploaded, waiting for the worker.
  b) **PROCESSING:** Training script is currently running.
  c) **COMPLETED:** Training finished, and the .msgpack model file is generated.
  d) **FAILED:** An exception or error occurred during the COLMAP or NeRF stage. This state machine allows the frontend to poll status updates and display real-time feedback detail to the user.

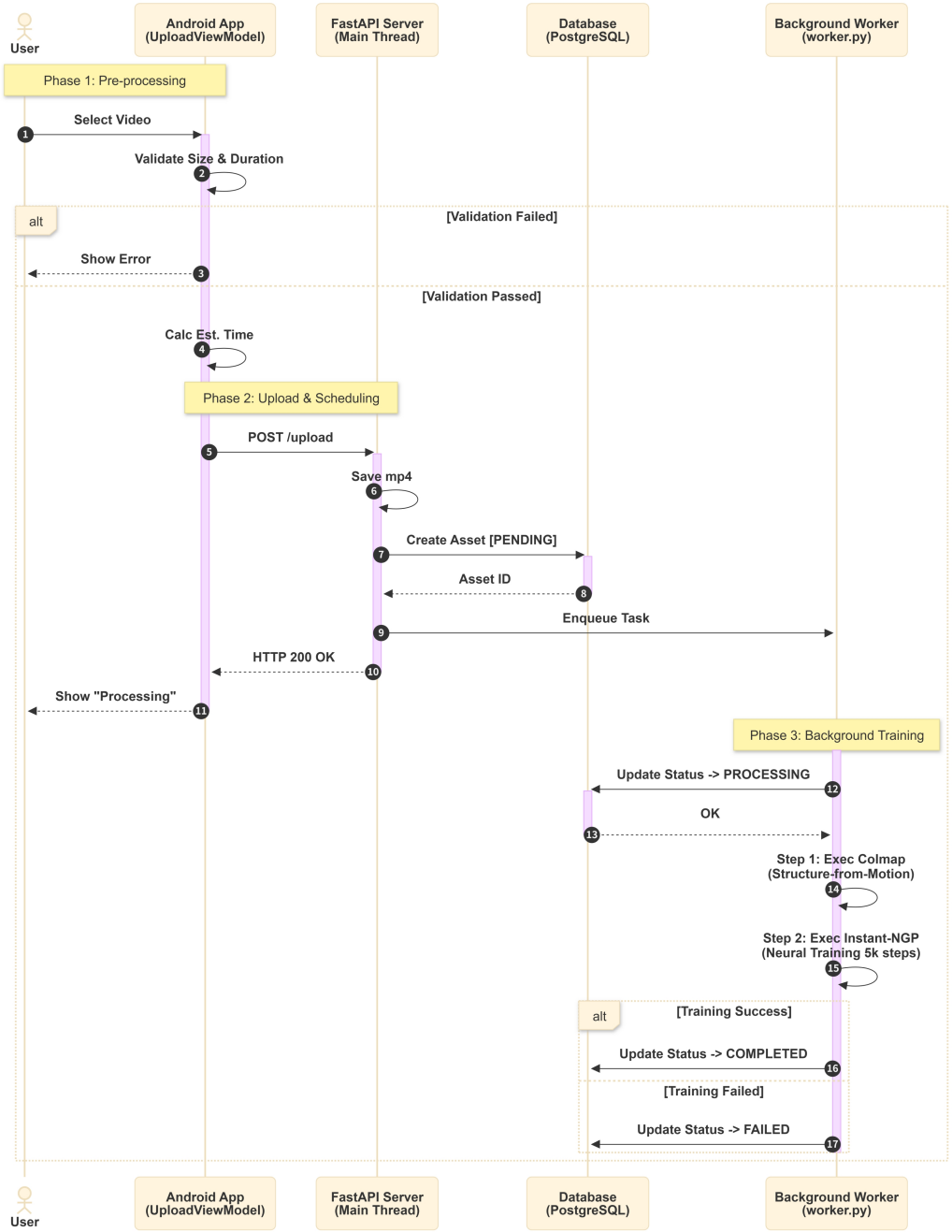

Figure 5-2, Model Generation and Training Pipeline Sequence Diagram (simplified)

5.3.3 Core Training Engine (COLMAP & Instant-NGP)

The generation of the 3D asset involves a two-stage pipeline executed via Python subprocesses, orchestrated by a custom *NonBlockingCommandRunner*.

a) **Structure-from-Motion (SfM):** The system first invokes the *colmap2nerf.py* script to cut frames and analyze the video frames. This process performs feature extraction and matching (configured with *--colmap_matcher exhaustive* for maximum accuracy) to calculate the camera intrinsic and extrinsic parameters. The output is a *transforms.json* file, which maps 2D frames to 3D spatial coordinates.

b) **Neural Radiance Fields (NeRF) Training:** The system utilizes **NVIDIA Instant-NGP** for real-time neural graphics primitive training. The pipeline executes the training script (*run.py*) using the data from the previous step.

- **Parameters:** The training is configured to run for **5000 steps** (*--n_steps 5000*), which balances generation speed with model visual fidelity.
- **Output:** The final result is a model snapshot saved in ***.msgpack*** format. This binary file contains the trained neural network weights and hash encoding data, which can be used for subsequential cloud rendering.

5.4 Real-time Communication and Chat System Implementation

The instant messaging module in Δ3D is built upon a hybrid architecture that combines **WebSocket** for low-latency, real-time bidirectional communication and **RESTful HTTP** APIs for data persistence, history synchronization, and inbox management.

5.4.1 Architecture and Protocol Design

The communication layer follows a client-server model designed to ensure message delivery and state synchronization.

- **WebSocket Protocol:**
  - **Connection:** The client establishes a persistent connection via *ChatSocketManager* using the *wss://* (or *ws://*) protocol. The handshake URL includes the user ID (api/v1/chat/ws/{user_id}) to map the active socket to a specific user session.
  - **Data Structure:** Messages are serialized using JSON. The system uses a strict data contract defined in *WSEvent* (Kotlin) and corresponding dictionaries in Python. A typical payload includes *type* (e.g., "new_message") and *data* (containing *sender_id, receiver_id, content,* and timestamps).
  - **Message Routing:** On the backend (*socket_manager.py*), a

ConnectionManager class maintains an in-memory dictionary *active_connections: Dict[int, WebSocket]*. When a message is received, the server asynchronously routes the payload to the specific *receiver_id*'s WebSocket instance while simultaneously echoing it back to the *sender_id* to ensure UI consistency across multiple devices.

- **Database Synchronization (Async/Sync Bridge):**
  - Since the backend uses *SQLModel* (synchronous) within an asynchronous FastAPI WebSocket loop, a specific technical challenge was non-blocking database writes.
  - **Solution:** The system utilizes *asyncio.get_running_loop().run_in_executor* to offload the blocking *save_message_sync* database operation to a separate thread pool. This ensures that the high-concurrency WebSocket event loop remains unblocked while persisting chat logs to the relational database.

5.4.2 Client-Side Connection Management

To handle the instability of mobile networks, the Android client implements a robust *ChatSocketManager* singleton responsible for the lifecycle of the connection.

- **Heartbeat Mechanism:** To prevent "ghost" connections where the link is dead but the socket remains open, the client utilizes OkHttp's *pingInterval*. The *ChatSocketManager* configures the client to send a ping frame every **15 seconds**. If the server fails to respond, the connection is deemed broken.
- **Automatic Reconnection:**
  - The system implements a self-healing mechanism using Kotlin Coroutines (*SupervisorJob*).
  - Upon receiving an *onFailure* or *onClosing* event, the manager triggers *scheduleReconnect()*. This function employs a delay strategy (currently set to 3 seconds) before attempting to re-establish the connection using cached authentication tokens (*cachedToken*).
  - **State Management:** Flags such as *isConnecting* and active job cancellation prevent race conditions where multiple reconnection attempts might stack up.

5.4.3 Data Loading and Pagination Strategy

To optimize memory usage and rendering performance, the chat UI does not load the entire conversation history at once.

- **Reverse Pagination:**
  - The message list uses a **LazyColumn** with *reverseLayout = true*.

- **Initial Load:** The *ChatViewModel* fetches the most recent 20 messages via an HTTP GET request (/api/v1/chat/history) sorted by time descending.
- **Infinite Scrolling:** As the user scrolls upward, a *snapshotFlow* monitors the visible item index. When the user approaches the end of the current list, the system triggers *loadMoreHistory()* using an *offset* and limit parameter.

- **List Merging:**
  - Historical messages (fetched via HTTP) and real-time messages (pushed via WebSocket) are merged into a single *MutableStateFlow*.
  - The *distinctBy { it.id }* operator is applied to prevent duplicate entries that might occur during network race conditions (e.g., a message arriving via WebSocket simultaneous to a history refresh).

5.4.4 Rich Media Sharing via Custom URI Schemes

A key feature of Δ3D is the ability to share 3D assets directly within the chat. This is implemented not by sending complex objects, but through a custom lightweight parsing protocol defined in *ShareLinkUtils*.

- **Encoding:** When a user shares a post from the *PostDetailScreen*, the app generates a specialized URI string:
  delta3d://share/post?id={id}&title={encoded_title}&cover={encoded_url}
- **Transmission:** This URI is sent as a standard text message. This ensures backward compatibility and simplifies the WebSocket payload.
- **Rendering:**
  - The *MessageBubble* component detects if a message content starts with the specific scheme (*delta3d://share/post*).
  - If detected, it intercepts the rendering pipeline. Instead of a text bubble, it parses the URI parameters using *Uri.parse* and renders a ***PostShareCard***.
  - This card displays the cached 3D model cover image and title, and provides a clickable surface that navigates the user to the *PostDetailScreen*.

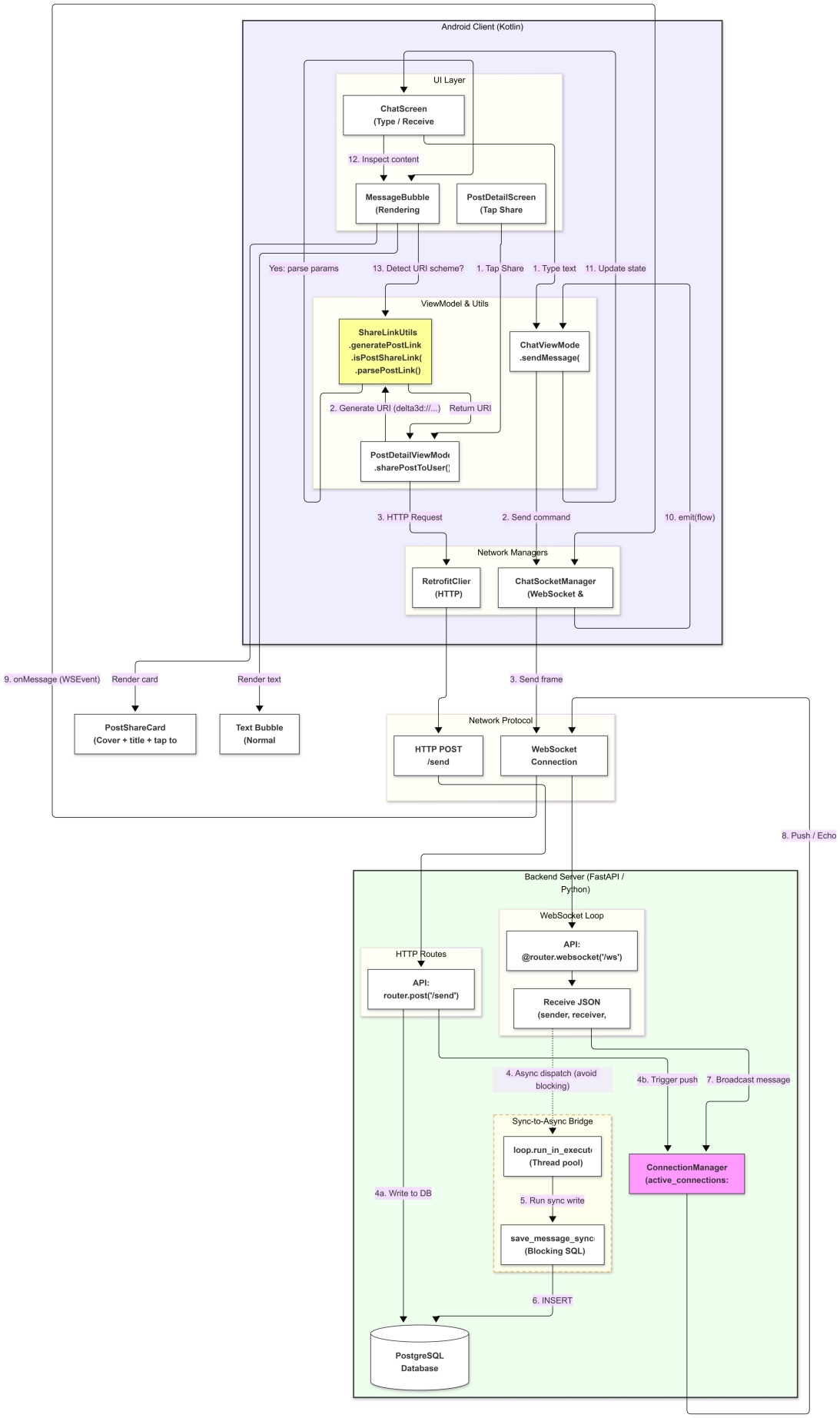
Android Client (Kotlin)
UI Layer
ChatScreen
(Type / Receive
12. Inspect content
MessageBubble
(Rendering
PostDetailScreen
(Tap Share
Yes: parse params
13. Detect URI scheme?
1. Tap Share
1. Type text
11. Update state
ViewModel & Utils
ShareLinkUtils
.generatePostLink
.isPostShareLink(
.parsePostLink()
ChatViewMode
.sendMessage(
2. Generate URI (delta3d://...)
Return URI
PostDetailViewMod
.sharePostToUser()
3. HTTP Request
2. Send command
10. emit(flow)
Network Managers
RetrofitClier
(HTTP)
ChatSocketManager
(WebSocket &
9. onMessage (WSEvent)
Render card
Render text
3. Send frame
PostShareCard
(Cover + title + tap to
Text Bubble
(Normal
Network Protocol
HTTP POST
/send
WebSocket
Connection
8. Push / Echo
Backend Server (FastAPI /
Python)
WebSocket Loop
API:
@router.websocket('/ws')
HTTP Routes
API:
router.post('/send')
Receive JSON
(sender, receiver,
4. Async dispatch (avoid
blocking)
4b. Trigger push
7. Broadcast message
Sync-to-Async Bridge
loop.run_in_execut
(Thread pool)
ConnectionManager
(active_connections:
4a. Write to DB
5. Run sync write
save_message_sync
(Blocking SQL)
6. INSERT
PostgreSQL
Database

Figure 5-3, Hybrid Message Orchestration Flow Diagram

5.4.5 Message Delivery Mechanisms

The system supports two distinct methods for sending messages, adapted to the user's context:

a) **In-Chat Sending (WebSocket):** When inside *ChatScreen*, messages are sent via *socketManager.sendMessage.* This is purely essentially fire-and-forget; the UI updates only when the server echoes the message back via the *messageFlow*, confirming receipt.

b) **External Sharing (HTTP Fallback):** When sharing a model from the "Post Detail" page (*PostDetailScreen*), a WebSocket connection might not be active or necessary for a background action. In this case, the *PostDetailViewModel* uses the HTTP POST api/v1/chat/send endpoint. The backend handles the database insertion and manually triggers the WebSocket manager to push the message to the recipient if they are online.

5.4.6 Notification and Read Status Synchronization

To enhance user engagement and ensure timely communication, the system implements a multi-level notification mechanism that synchronizes the "unread" status across the application globally.

- **Global Unread State Management:**
  - **Centralized Source of Truth:** The application maintains a global unread counter within the *SessionViewModel*. This ViewModel exposes a *StateFlow<Int>* (e.g., *unreadCount*) that serves as the single source of truth for the entire app's notification UI.
  - **Synchronization Triggers:** The global count is updated in real-time through three pathways:
    a) **Initial Load:** Upon app launch, the *SessionViewModel* queries the backend API to fetch the aggregate count of unread messages.
    b) **Real-time Updates:** When a new message arrives via WebSocket (as described in 5.4.1), the client triggers a refresh of the unread count.
    c) **User Action:** When a user enters a specific chat, the *markAsRead* API is invoked, prompting a re-calculation of the remaining unread messages.
- **UI Representation:**
  - **Community Dashboard (Red Dot Indicator):** In the *CommunityScreen*, the top navigation bar observes the global *unreadCount* from *SessionViewModel*. If the count is greater than zero, a visual indicator (Red Badge) is rendered

over the chat icon. This provides a non-intrusive alert to the user while they are browsing the community feed.

- **Conversation List (Granular Counts):** The *ChatListScreen* displays a detailed breakdown. The *ChatListViewModel* fetches a list of *ChatConversation* objects, where each object contains a specific *unread_count* field calculated by the backend. The UI renders a numeric badge (e.g., "3") next to the corresponding contact's avatar for any conversation with unread messages.

- **Backend Aggregation Logic:**
  - The backend (*chat.py*) facilitates this by providing a specialized endpoint (e.g., api/v1/chat/conversations) that performs a grouped aggregation query on the database. It counts messages where *is_read = False* and *receiver_id = current_user_id,* grouped by the *sender_id.* This ensures that the client receives the latest unread counts for each conversation in a single HTTP request, minimizing network overhead.

5.5 Real-time 3D Model Preview and Interactive Streaming

To enable high-fidelity visualization of Neural Radiance Fields (NeRF) on mobile devices without consuming local computing resources, the system adopts a **Server-Side Rendering (SSR)** architecture. The core implementation relies on a low-latency video streaming pipeline combined with a remote control feedback loop.

5.5.1 Architecture Overview

The model preview module functions on a **Pixel Streaming** paradigm. The heavy lifting of rendering the 3D scene is performed by **Instant-NGP** on the high-performance backend server. The visual output is captured, encoded, and streamed to the Android client via WebRTC. Concurrently, user touch gestures on the Android device are converted into control commands and sent back to the server to manipulate the renderer's viewport.

The data flow is defined as follows:

1. **Rendering:** *Instant-NGP* renders the scene on the server GPU.
2. **Capture & Encoding:** *FFmpeg* captures the window and encodes it using NVENC.
3. **Transmission:** The stream is pushed via RTSP to a **MediaMTX** server.
4. **Playback:** The Android client consumes the stream via WebRTC (WHEP) inside a *WebView*.
5. **Control:** User inputs are processed by *pyautogui* to simulate mouse events on the server.

5.5.2 Visual Rendering and Screen Capture

The system manages the lifecycle of the rendering process via the *InteractiveStreamSession* manager.

- **Process Management:** The backend leverages *ExternalCommandRunner* to launch the *Instant Neural Graphics Primitives* (Instant-NGP) GUI in a headless-capable environment.
- **Window Capture:** Instead of processing raw frame buffers, the system utilizes the **FFmpeg gdigrab** device to capture the specific window content. The command targets the window title *title=Instant Neural Graphics Primitives* to ensure only the relevant viewport is streamed, excluding unrelated desktop artifacts.

5.5.3 Low-Latency Encoding and Transmission Pipeline

Achieving real-time interaction requires minimizing the "Motion-to-Photon" latency. The system implements a highly optimized FFmpeg pipeline defined in *stream_manager.py*:

a) **Hardware Acceleration:** The system utilizes the **NVIDIA NVENC** (*-vcodec h264_nvenc*) encoder to offload CPU usage and accelerate compression.

b) **Low-Latency Tuning:**
   - **Preset:** *-preset llhq* (Low Latency High Quality) and *-tune ll* (Low Latency) are strictly applied.
   - **GOP Structure:** The Group of Pictures (GOP) is minimized (*-g 15*) with no B-frames (*-bf 0*) to prevent frame reordering delays.
   - **Buffer Control:** Input buffers are disabled (*-fflags +genpts+flush_packets+nobuffer*) and the analyze duration is set to zero to ensure immediate stream transmission.
   - **Bitrate Control:** Constant Quantization Parameter (*-rc constqp, -qp 19*) is used to maintain image quality over variable bitrate, crucial for preserving 3D details.

c) **Stream Relay (MediaMTX):** FFmpeg pushes the feed via **RTSP (TCP)** to the local MediaMTX instance (rtsp://localhost:8555/live). TCP transport (*-rtsp_flags prefer_tcp*) is enforced to prevent packet loss artifacts, relying on the high bandwidth of the local loopback.

5.5.4 WebRTC Playback and WHEP Integration

The Android client does not use a standard video player (like ExoPlayer) due to high latency in HLS/DASH. Instead, it implements a **WebRTC** solution via a bridging HTML5 interface.

- **Dynamic Stream URL Provisioning**: Upon a successful api/v1/start/{asset_id} request, the backend API calculates the appropriate access endpoint based on the request's origin. If the backend detects a public IP access, it returns a WebRTC URL configured with the public gateway; otherwise, it provides a local intranet address. This URL points to the MediaMTX WebRTC server (typically on port 8889 or 29655).
- **WHEP Protocol:** The frontend uses the **WebRTC HTTP Egress Protocol (WHEP)** for signaling. The webrtc_player.html script dynamically constructs the negotiation endpoint by appending /whep to the stream URL. It then initiates the session by sending the local SDP offer via a **POST** request to this target URL to exchange capabilities with the MediaMTX server.
- **WebView Optimization:** To bridge the native UI and the WebRTC logic, the app uses a custom *WebViewPool*. This utility pre-warms *WebView* instances during the application's *IdleHandler* phase (in *Application.onCreate*). This eliminates the significant overhead of initializing the Chromium engine when the user enters the preview screen.
- **Network Intelligence:** The *SessionViewModel* actively monitors network conditions to ensure the client is on a compatible network before initiating the stream. It differentiates between LAN and WAN usage, ensuring that the backend's resolved stream URL matches the client's current connectivity status.

5.5.5 Remote Interactive Control Logic

The system implements a robust "Remote Desktop" style control scheme, allowing users to manipulate the 3D model (Rotate, Pan, Zoom) via the mobile interface.

a) **Command Transmission:** The Android UI (*StreamPreviewScreen.kt*) captures gestures via a D-Pad and mode switches (Rotate vs. Pan). These are serialized into *ControlCommand* objects (Action, Direction, Start/Stop) and sent to the FastAPI backend.

b) **Automated Input Simulation:** The backend utilizes a *ContinuousController* (in *continuous.py*) to map API requests to OS-level mouse events using *pyautogui* and *win32gui*.
   - **Window Targeting:** The controller dynamically locates the NGP window handle and calculates its center and boundary coordinates (*RECT*).
   - **Smoothing Algorithm:** To counteract network jitter and provide a fluid experience, mouse movements use an **Ease-In-Out Quad** smoothing function (*pyautogui.easeInOutQuad*).
   - **Momentum & Physics:** The controller implements a thread-based

continuous movement loop. It calculates acceleration and deceleration phases, preventing abrupt cursor stops and simulating the inertia felt in native touch interactions.

- **Boundary Clamping**: The logic includes safety mechanisms to clamp cursor coordinates within the specific window *rect*, ensuring mouse events (drags/scrolls) never drift outside the rendering viewport.

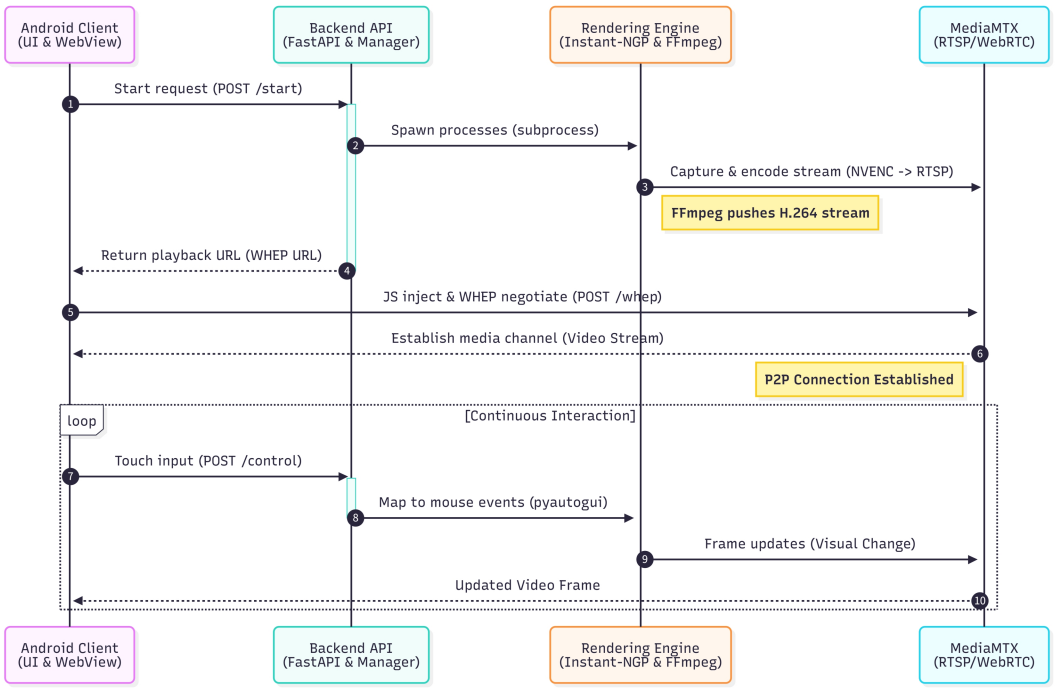

Figure 5-4, Sequence Diagram for Real-time 3D Model Preview and Interactive Streaming

5.6 Hybrid Network Architecture & NAT Traversal

To ensure the Δ 3D backend and streaming services, which hosted on a local Windows workstation behind a NAT, are accessible to mobile users on both public networks (WAN) and the local network (LAN), the system employs a hybrid connectivity strategy. This involves **FRP (Fast Reverse Proxy)** for NAT traversal and an **application-layer dynamic switching mechanism**.

5.6.1 FRP-Based NAT Traversal

Since the backend server resides on a private subnet (127.0.0.1 locally), direct access from the public internet is impossible. We utilize **FRP** to tunnel traffic from a public relay server (Aliyun, IP: 47.107.130.88) to the local machine.

The configuration (*frpc.ini*) maps three distinct communication channels required for

the application's operation:

a) **REST API Tunnel (TCP):**
   - **Local:** 127.0.0.1:8000 (FastAPI backend)
   - **Remote:** 47.107.130.88:29654
   - **Purpose:** Handles all standard HTTP requests (Authentication, Model Asset management, Chat, Post).

b) **WebRTC Signaling Tunnel (TCP):**
   - **Local:** 127.0.0.1:8889 (MediaMTX WebRTC interface)
   - **Remote:** 47.107.130.88:29655
   - **Purpose:** Transmits SDP (Session Description Protocol) offers/answers for video streaming.

c) **WebRTC ICE/Media Tunnel (UDP):**
   - **Local:** 127.0.0.1:8189
   - **Remote:** 47.107.130.88:8189
   - **Purpose:** Handles the actual UDP transmission of video/audio packets. This requires a direct UDP port mapping to ensure low-latency streaming through the proxy.

5.6.2 Client-Side Intelligent Network Detection

The Android client implements a **self-adaptive network detection mechanism** in *SessionViewModel.kt* to prioritize low-latency LAN connections over the WAN tunnel.

- **Socket Probing:** Upon app launch or network status change, the *checkAndSelectBestNetwork()* coroutine attempts to establish a raw Socket connection to the configured LAN Host (10.252.130.135) on Port 8000 with a **2-second timeout**.
- **State Switching:**
  - **Success:** The app switches *AppConfig.currentMode* to *NetworkMode.LAN*.
  - **Failure/Timeout:** The app defaults to *NetworkMode.WAN*.
  - **User Notification:** A *NetworkStatusDialog* informs the user of the detected environment (Local vs. Remote) to manage expectations regarding streaming latency and upload limits.

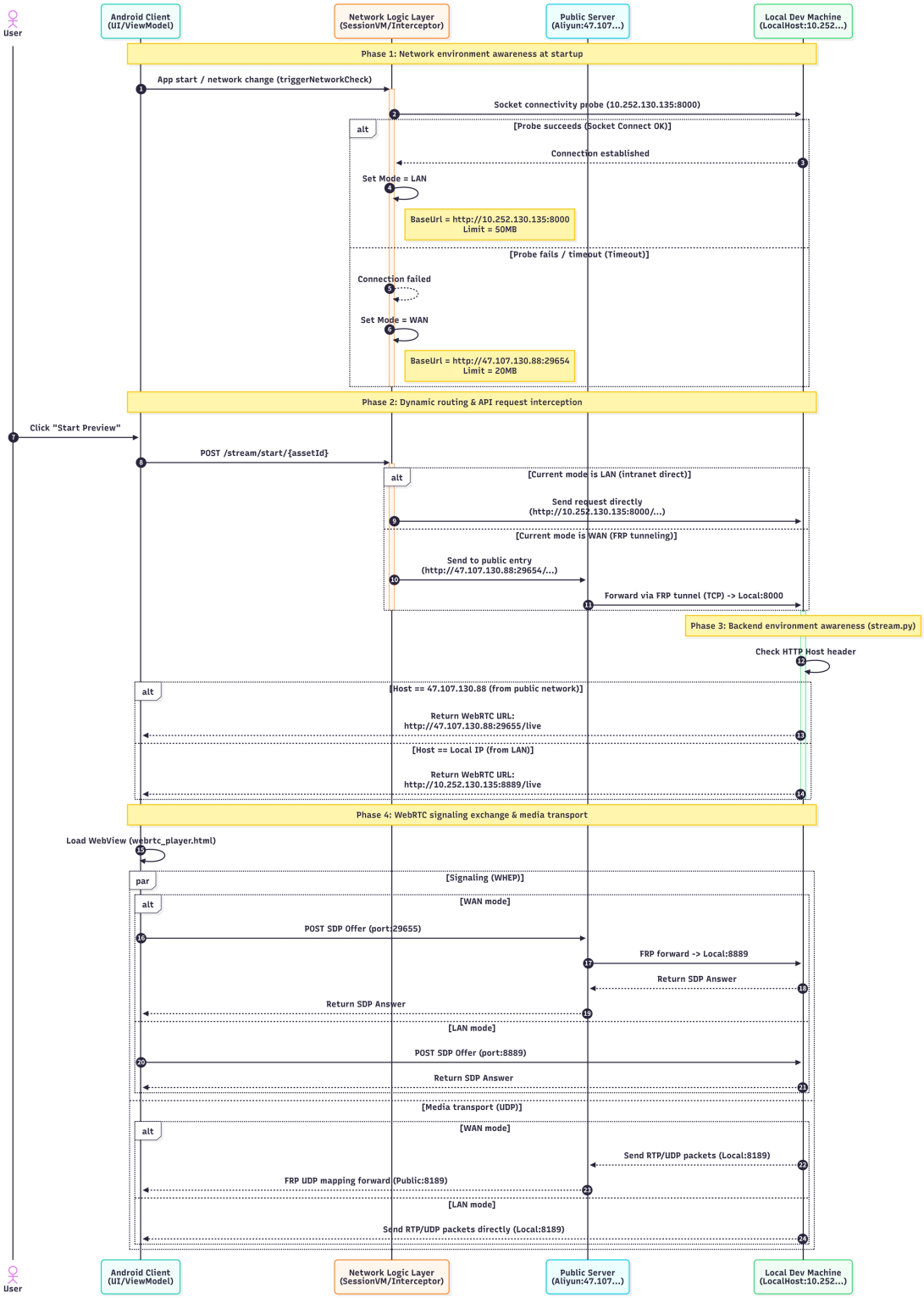

Figure 5-5, LAN/WAN Adaptive Routing + FRP Tunneling WebRTC Streaming Sequence Diagram

5.6.3 Dynamic API Routing (Interceptor Pattern)

To support seamless switching without restarting the app, the networking layer utilizes an OkHttp *Interceptor*.

- **HostSelectionInterceptor:** Defined in *RetrofitClient.kt,* this interceptor

intercepts every outgoing HTTP request. It checks *AppConfig.currentBaseUrl* (which is updated by the detection logic above) and dynamically rewrites the request's host and port at runtime.

- **LAN Mode:** Requests are rewritten to http://10.252.130.135:8000.
- **WAN Mode:** Requests are rewritten to http://47.107.130.88:29654.

5.6.4 Context-Aware Streaming Strategy (Backend)

While the frontend handles API routing, the **WebRTC video stream** requires specific handling on the backend because the streaming protocol (WHEP) relies on returning specific resource URLs.

The backend endpoint api/v1/stream/start/{asset_id} (in *stream.py*) implements logic to distinguish the access source:

a) **Request Inspection:** The server inspects the *Host* header of the incoming HTTP request.

b) **Conditional URL Generation:**
   - If Host == Public IP (47.107.130.88): The server recognizes the request is coming via the FRP tunnel. It returns the WebRTC URL using the mapped external port (29655).
   - If Host == Local IP: The server assumes a direct LAN connection and returns the WebRTC URL using the local port (8889).

This special handling ensures that when the frontend initiates a stream, the video player (WebView) receives a valid signaling URL that matches the current network context, preventing connection failures caused by port mismatches during NAT traversal.

5.7 Unstructured Data Storage and Access

The Δ3D application manages significant volumes of unstructured data, including user-uploaded source videos, processed 3D model files, profile avatars, and cover images. The system employs a hybrid storage strategy where metadata is persisted in the relational database, while binary large objects (BLOBs) are managed via a structured file system with RESTful access patterns.

5.7.1 Storage Architecture and Configuration

The system adopts a **Separation of Concerns** strategy defined by the backend configuration:

- **Structured Metadata:** The database stores references (relative paths like */static/uploads/...*) rather than binary data.
- **Physical Storage Root:** All unstructured data is anchored to a configurable root directory (UPLOAD_DIR, defaulting to *./static/uploads*). This allows the storage

location to be modified via environment variables without code changes, facilitating deployment across different environments.

5.7.2 Server-Side File Management

The backend utilizes Python's *pathlib* for OS-agnostic path manipulation and *shutil* for high-performance I/O streaming.

1) **Directory Hierarchy:**
   - **User Assets:** Stored in *static/uploads/avatars* and *static/uploads/covers*.
   - **3D Model Assets:** Each asset is isolated in a unique directory: *static/uploads/{asset_uuid}/*. This directory contains the source *video.mp4*, the generated *model.msgpack*, and a subdirectory */images/* for preview frames.
2) **Storage Lifecycle linked to Pipeline:**
   - **Ingestion:** When the upload described in Section 5.3.1 completes, the server uses *shutil.copyfileobj* to stream the binary data directly to disk as *video.mp4*.
   - **Asynchronous Artifact Generation:** As the training pipeline (Section 5.3.3) executes, new files (*transforms.json, model.msgpack*) are incrementally written to the asset's directory. This decoupling ensures that the file system structure evolves from a single video file to a complete 3D asset package without blocking the main thread.
3) **Naming Conventions:**
   - **Collision Avoidance:** Model Asset directories use *uuid.uuid4().hex* to guarantee uniqueness. User profile images use the pattern *{user_id}_{uuid_fragment}.{ext}*.
   - **Sanitization:** Filenames are sanitized server-side during download generation to ensure compatibility across different operating system file systems.

5.7.3 Client-Side Access and URL Resolution

The client accesses unstructured data through standard HTTP GET requests, utilizing dynamic URL construction and caching strategies.

1) **Dynamic URL Resolution:**
   - The frontend (*AppConfig.currentBaseUrl*) dynamically prefixes the relative paths stored in the database, like *${AppConfig.currentBaseUrl.removeSuffix("/")}${avatorUrl}*.
2) **Convention-Based Preview Retrieval:**
   - To reduce database overhead, the system uses a Convention over

Configuration approach for image sequences.

- In *AssetDetailScreen.kt* and *PostDetailScreen.kt*, the *generateRandomImageUrls* function does not query the API for individual image paths. Instead, it programmatically constructs URLs by appending sequential counters (e.g., *.../images/0004.jpg*) to the asset's base path. This relies on the server-side processing pipeline always outputting frames to the standardized */images/* subdirectory.

3) Optimized Rendering:

- **Coil Integration:** The *AsyncImage* component is used for rendering. It handles memory caching, disk caching, and cross-fading (*crossfade(300)*).
- **Fallback Mechanisms:** The UI actively monitors asset status. If a model asset is marked "failed", the client intercepts the image request and renders a static resource (*/static/states/error.png*) instead of attempting a network call.

5.7.4 Data Transmission and Download Management

The system implements robust mechanisms for transferring large unstructured files between client and server.

1) Multipart Upload Protocol:

- The *ApiService.kt* interface defines *uploadAsset* using *@Multipart*. Binary data is streamed via *MultipartBody.Part* alongside the metadata *RequestBody*. This allows the server to receive the binary stream and the structural data in a single HTTP transaction, ensuring atomicity.

2) Native Android Download Management:

- For retrieving 3D model files, the Android client delegates the task to the Android *DownloadManager* system service rather than using a synchronous *Retrofit* call.
- **Configuration:** The request is configured to save files to the public *DIRECTORY_DOWNLOADS*, set notification visibility (*VISIBILITY_VISIBLE_NOTIFY_COMPLETED*), and allow downloads over roaming/metered networks.
- **Format Selection:** The client requests specific file formats (OBJ, GLB, PLY) via query parameters, which the backend resolves to the appropriate file path before streaming the response.

6. Testing and User Experience Analysis

6.1 User Testing Methodology

To comprehensively evaluate the application's stability, usability, security, and compatibility, we employed a multi-dimensional and systematic user testing approach. Testing was conducted across several professional platforms, including **Firebase Test Lab, Kobiton, Testin, WeTest, and UXcam**, covering more than 30 Android device models (both emulators and real devices). This broad device matrix enabled us to test the application across various fragmentation scenarios and ensure compatibility and stability across different devices and environments.

The testing involved not only automated test scripts but also expert testers and real users. Expert testing focused on deep functional and experiential evaluations, while real-user behavior data collected via platforms like UXcam provided insights based on actual user interactions.

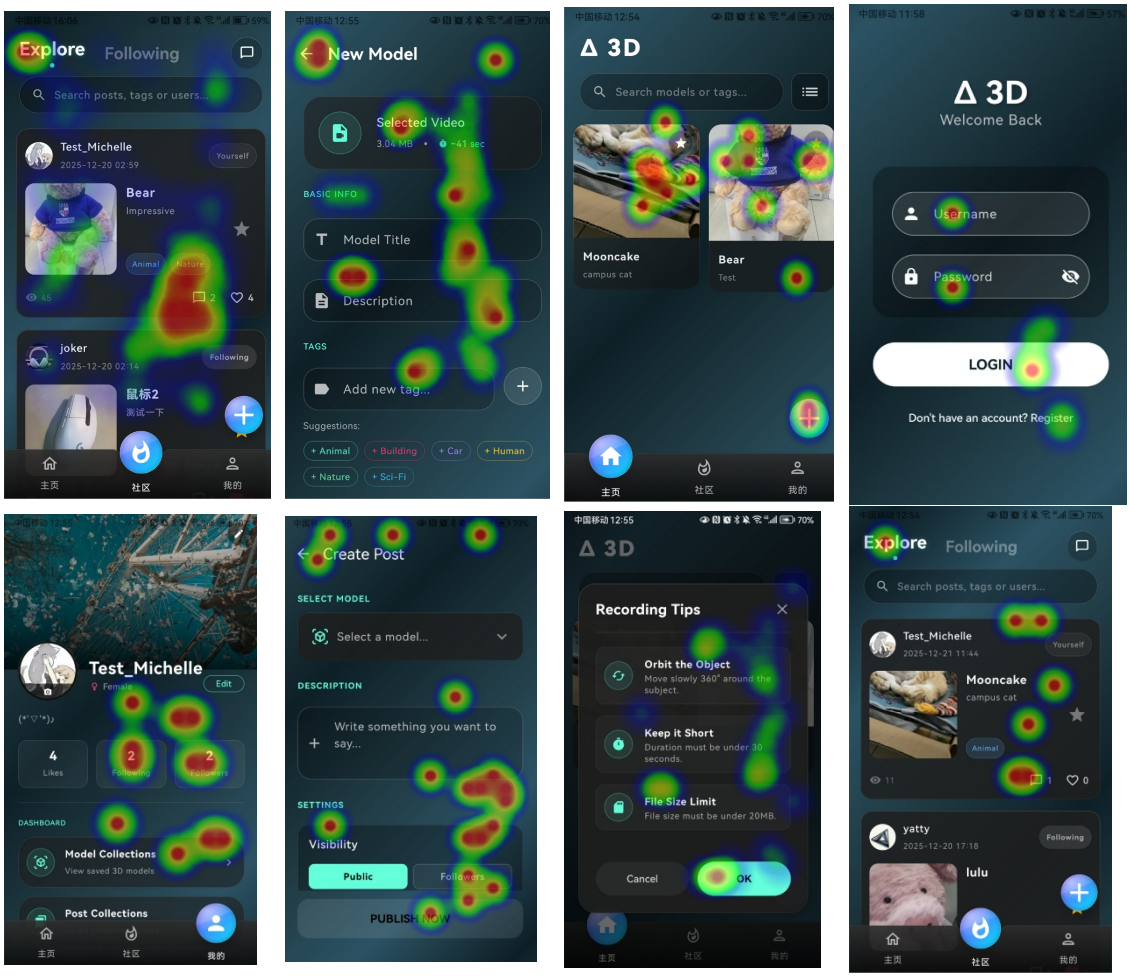

Figure 6-1, Interaction heatmaps across key screens

As shown in Figure 6-1, these heatmaps reveal where actual users actually tap and focus across major screens. Dense hotspots around cards, search bars, floating actions,

login CTAs, or dialog confirmations indicate high-frequency interaction zones. Scattered hotspots or repeated tapping patterns often suggest insufficient feedback, slow loading, small touch targets, or mis-taps.

For the testing methods and evaluation criteria, we combined a variety of testing approaches. **Monkey random operation testing** and **Firebase Robo automated exploration tests** were conducted to evaluate the application's stability and crash rates under abnormal operations. **Compatibility testing** was performed across multiple real devices and emulators to ensure the consistency of core functionality and the UI. Additionally, **black - box testing,** which utilized equivalence partitioning and boundary value analysis, was employed to verify functional logic and exception handling.

We also closely monitored key performance indicators (KPIs) during **performance testing**, including **installation/launch time, CPU usage, memory consumption, network traffic, battery temperature, and UI smoothness (FPS)**. This helped ensure that the app runs smoothly while consuming reasonable system resources. Below are some of the results:

Connection	Average (ms)	Minimum (ms)	Maximum (ms)	Median (ms)	
Browser to Device	467	310	2,670	400	Details
Browser to Server	449	289	2,651	384	Details
Server to deviceShare	18	15	35	16	Details
deviceShare to Device	N/A	N/A	N/A	N/A	Details
Device to Browser	461	86	6,859	428	Details
TOTAL	928	396	9,529	828	Details

Figure 6-2, End-to-end latency breakdown table

This table decomposes an interaction into multiple segments and reports average, median, and maximum latencies. The most actionable insight is typically the tail latency (max): if the server-internal segment is small, the bottleneck is not on the backend; spikes on Device→Browser or TOTAL indicate occasional stutters caused by network variability, device rendering, WebView loading, or scheduling. This evidence motivates engineering optimizations such as unified routing, WebView pooling, and systematic weak-network scenario testing to reduce tail latency and improve consistency.

Figure 6-3, RAM usage over time

In Figure 5, The pattern “rapid ramp-up → plateau → mild fluctuations” typically indicates resource allocation during startup/first-screen loading, followed by stable operation. A sharp drop mid-run often corresponds to page transitions, process recreation, or garbage collection. The key diagnostic is whether the curve shows sustained linear growth; if not, the risk of obvious memory leaks is low.

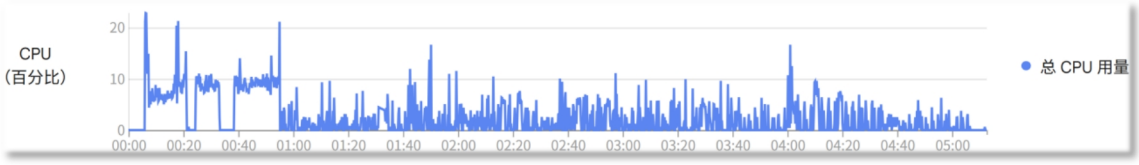

Figure 6-4, CPU usage over time

The CPU stays low most of the time with occasional spikes. When spikes are brief and scattered, they usually reflect bursty tasks (rendering, decoding, callbacks, list recomposition) rather than sustained heavy load. From a UX perspective, what matters is whether spikes align with dropped frames or unresponsive taps.

Figure 6-5, Cross-device performance dashboard

This dashboard consolidates key metrics from multi-device testing and highlights best/worst devices. Its value lies in exposing variability beyond averages: even if overall startup is fast, a few devices may experience slow installs or cold starts, creating a tail of poor UX.

For **security testing**, we performed mobile security scans to identify security risks and potential vulnerabilities within the APK. Additionally, **accessibility testing** was conducted in accordance with **WCAG standards**, automatically detecting and manually verifying accessibility features like color contrast, touch target size, and content labeling. Finally, we simulated weak network conditions and disconnections to test the application's robustness and fault tolerance.

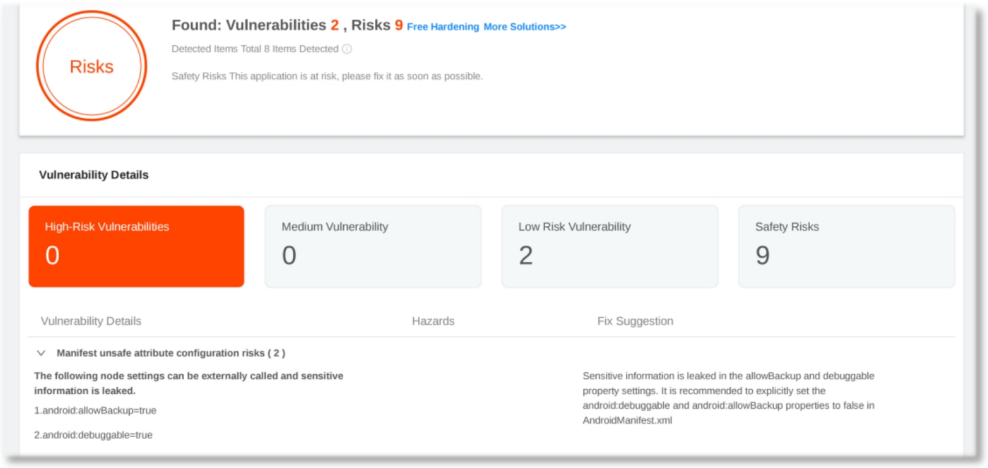

Figure 6-6, Mobile security scan summary

Building on existing tests, we are currently supplementing with **automated regression testing for core user journeys** and a **weak-network/abnormal scenario matrix** to further enhance test coverage and efficiency.

6.2 Identified Usability Issues - Initial Version

Through comprehensive testing of the initial version, several key usability, performance, and security issues were identified:

The **accessibility issues** were among the most prominent. Some clickable elements had a **touch target size** smaller than the recommended **48dp**, which impaired user accuracy during interactions. Furthermore, we found that the **color contrast** between text and background on several elements did not meet the **WCAG AA/AAA standards.** For instance, the contrast of the "good" text was only **4.08:1**, well below the **AAA requirement of 7:1**, leading to poor readability. Kobiton testing flagged 66 issues, all related to color contrast. Additionally, some icons or controls lacked the required contentDescription or semantic labels, making them unrecognizable by screen readers, which can prevent users with visual impairments from interacting with the app effectively.

In terms of **performance and compatibility**, while the app performed well on most devices, there were **significant performance variances** across different models. The installation time ranged from **2.58s to 24.97s**, and cold launch time varied from **0.14s to 5.85s**, indicating that some devices had a poor experience. There were also **occasional stuttering** issues, with **long-tail delays** in end-to-end data transmission. For instance, **Device→Browser** had a **maximum jitter of 6859ms,** and the total delay reached **9529ms**, indicating sporadic severe lag.

In terms of **functional and interaction design flaws,** the **core user journey** faced significant bottlenecks. Analysis of user behavior revealed that many users exited

quickly from the **MainActivity** page, and repeated "tapping" behavior was observed on the **Stream_Preview** page, suggesting **interface lag, unresponsive clicks, or process blockages.** Also, during key processing stages such as uploading and modeling, the **progress bar** lacked incremental feedback, leaving users unaware of the current progress and leading to waiting anxiety. **State synchronization issues** were also identified, particularly when switching accounts, where the chat sender's identity was not updated in time, creating synchronization risks.

Regarding **security risks,** several vulnerabilities were identified, including **insecure Manifest configurations** (e.g., debuggable, allowBackup), unnecessary **exposed components** (e.g., exported), **plain HTTP transmission,** and outdated **third-party SDKs** (e.g., Okio, Retrofit), all of which need to be addressed promptly to ensure the app's security.

6.3 Iterative Improvements Based on User Feedback

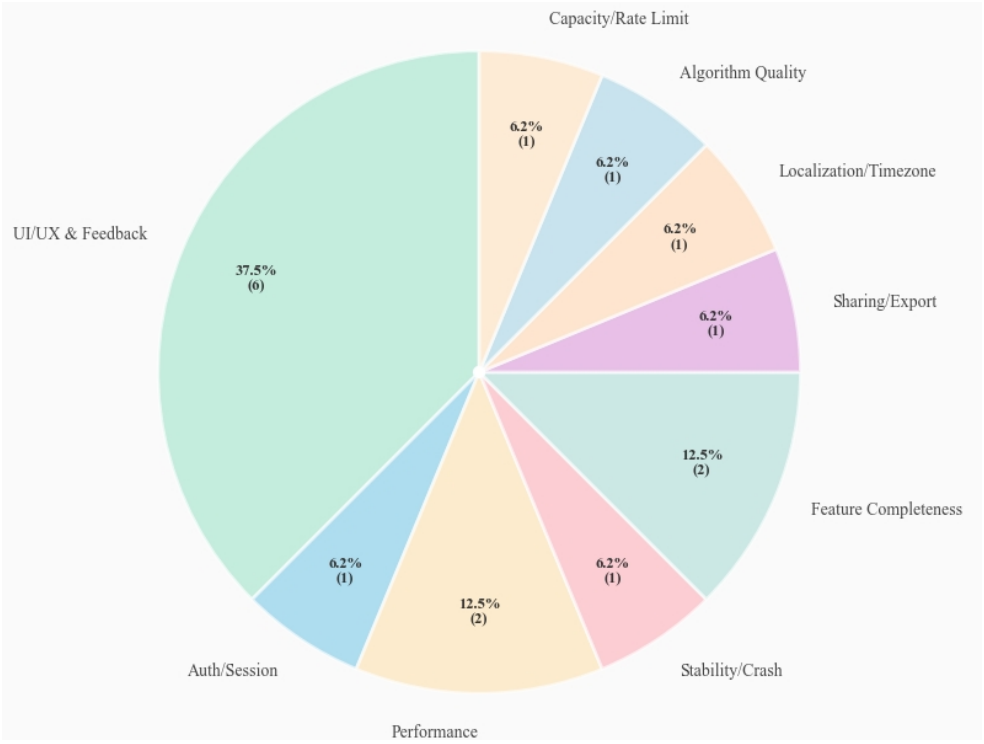

Figure 6-7, User issue category breakdown

This pie chart aggregates user-reported issues by category and shows where problems concentrate. UI/UX and interaction feedback dominate, followed by performance and feature completeness—consistent with the heatmap evidence of dense interactions and repeated tapping in some screens.

Based on user feedback, we made several iterative improvements that significantly enhanced the user experience. Some of the feedback and improvements are shown in

the table below:

Table 6-1, Summary of feedback-driven iterative improvements

User Feedback	Problem Analysis	Improvement & Impact
U1: Processing progress is not perceivable.	Progress bar lacked stage percentages, making the wait uncertain.	Added estimated time display and timeout fallback. Estimated processing time calculated based on file size, enhancing user expectation. Added timeout polling to prevent "hanging".
U1/U6: Incorrect chat identity after account switch; crash when entering chat via share link.	Login state cache not synchronized; concurrent HTTP and WebSocket messages caused UI crash.	Unified login state refresh; fixed concurrency crash. Synchronized user identity, cleared old cache, and fixed duplicate key issue in message list.
U2: List label colors clash; time display not in local timezone.	Color generation strategy resulted in low contrast; time storage didn't account for timezone.	Optimized color algorithm; unified timezone handling. Adopted cyclic list assignment, expanded color gamut, and ensured UTC storage with timezone conversion on the frontend.
U3: Conflict between mandatory description for model sharing and homepage display; waterfall layout jumps on refresh.	Process design caused information gap; unsaved card heights triggered layout reflow.	Added description fallback strategy; persisted card heights. Fallback to basic description if extra description is empty. Cached card heights to reduce visual jitter.
U4: Chat history loading stutters; model detail page lacks edit/feedback options.	Synchronous loading of many messages blocked rendering; missing functional entries.	Optimized paginated loading; added edit and report entries. Pre-fetches historical messages before reaching the bottom. Added permission-controlled edit function and issue reporting channel.
U5: Unstable experience with model external links; 3D preview interactions are not responsive.	Heavy reliance on dynamic pages led to poor performance; preview interaction logic and performance needed optimization.	Changed external links to static pages; optimized preview interaction. Improved loading speed and stability of shared links. Optimized rendering frame rate and gesture response logic.
U6/U7: No direct chat entry from follower pages; unstable image recognition; lack of shooting guidance for modeling.	Disconnected social pathways; room for improvement in matching algorithm; users lacked operational guidance.	Added "Enter Chat" entry; optimized matching algorithm; enhanced operational guidance. Shortened social paths; backend adjusted matching strategy; added clear guidance like "circle the object".
U8: No limit on video uploads; logout is prone to	Large files caused processing failures and resource waste;	Added file size limit; added confirmation dialog for logout. Limited uploads to ≤50MB or 1

accidental taps.	critical action lacked confirmation.	minute with clear prompts. Added confirmation dialog before executing logout.

In addition to UI- and interaction-level issues, we also identified a **system-level accessibility bottleneck** caused by the **deployment environment**. Since the modeling server was hosted inside a campus network, public Internet users could not directly reach the server to preview or interact with reconstructed models. Our initial low-latency approach relied on **UDP transport**; however, UDP introduced dynamically allocated ports, which invalidated **FRP port-forwarding rules** and made stable public access nearly impossible. Moreover, the original **RTSP-based streaming pipeline** did not handle **NAT traversal** reliably in this scenario. To resolve this, we migrated the streaming protocol from RTSP to **WebRTC**. With WebRTC, we only need two fixed ports while benefiting from better NAT traversal and real-time interaction support. This change significantly simplified the **FRP configuration** and, more importantly, enabled **stable public access**: users can now preview models remotely and interact with them smoothly over the public network.

6.4 User Experience Evaluation Results

After iteration and user feedback, the user experience evaluation can be summarized as follows:

- **Stable Core Performance:** The app performed well across most devices, with an **average FPS of 51.81.** CPU and memory usage were reasonable, battery temperature and data consumption were well-controlled, ensuring smooth operation.

- **Complete Functional Implementation:** 100% pass rate for 92 functional test cases, indicating correct core business logic and reasonable error prompts.

- **Strong Foundational Stability:** During testing, **UI freeze and crash counts were 0**, demonstrating good foundational code robustness.

- **Rapid Iteration Response:** We were able to quickly identify and implement effective fixes for user-reported issues, establishing a positive development-feedback loop.

6.5 Summary

This user experience testing, combining automated tools, expert evaluation, and real-user feedback, comprehensively assessed the application’s quality. The results show that the app performs well in **core functionality, basic performance, and stability**, meeting users' fundamental needs. Through rapid iterative improvements, we successfully resolved several key issues affecting user flow and perception, improving usability and friendliness.

However, there is still room for optimization in **accessibility compliance, performance on extreme devices, and interaction smoothness**. Some security risks also need immediate attention.

Future Improvement Directions:

1. **Systematically Address Accessibility Issues:** Implement a process to ensure that color contrast and touch target sizes meet WCAG guidelines.

2. **Deepen Performance Optimization:** Focus on optimizing the performance of the slowest devices and investigating the root causes of lag spikes. Enhance testing and strategies for weak-network and abnormal scenarios.

3. **Granular Interaction Analysis:** Use behavior analytics tools to investigate the specific reasons behind user drop-off and "repeated tapping" behavior in pages like MainActivity, optimizing interaction feedback and page logic.

4. **Complete Security Hardening:** Address the identified APK security risks and third-party library vulnerabilities as planned.

5. **Refine Testing System:** Continue enhancing the automated regression testing for core journeys and the scenario matrix to improve testing efficiency and depth, ensuring quality assurance for continuous iteration.

7. Maintenance and Support

To ensure the long-term stability and usability of the Δ3D platform, a structured maintenance lifecycle and support system have been established. This section outlines the strategies for version control, issue resolution, and the incorporation of user feedback into future iterations.

7.1 Maintenance and Support

The maintenance strategy follows a **Semantic Versioning (SemVer)** protocol (Major.Minor.Patch) to manage releases via the GitHub repository.

7.1.1 Version Control and Release Strategy

- **Repository Management:** The project source code and release artifacts are hosted at **android-app-development-course/2025-Autumn-Aberdeen-10-Delta3D**.
- **Release Cycle:**
  - **Patch Releases (x.x.1):** Weekly or bi-weekly updates focused on bug fixes, performance optimization, and minor UI tweaks.
  - **Minor Releases (x.1.x):** Monthly updates introducing new features or significant improvements to existing modules.
- **Distribution:** Compiled APK files (Delta3D.apk) are distributed via the **GitHub Releases** page, ensuring users have access to a verified and consistent installation source.

7.1.2 Future Roadmap and Feature Integration

The development team has identified critical technical debt and feature requests to be addressed in the upcoming maintenance cycles (Versions v1.1.0 - v1.2.0):

1) **Data Management Enhancement:**
   - **Model Deletion:** Implementation of a "Delete" endpoint in *assets.py* and corresponding UI actions in the *Profile* and *Asset Detail* screens. This will allow users to remove unwanted models to free up server storage and manage their portfolio.
   - **Post Deletion:** Enabling users to retract community posts via the *Post Detail* interface, ensuring user control over their shared content.
2) **Authentication Security Upgrade:**

- **Password Recovery:** Integration of a "Forgot Password" flow using email verification (SMTP) or security questions to restore account access, addressing the current limitation of the login module.

7.2 Technical Support

Support channels are designed to categorize issues effectively, distinguishing between software bugs, account issues, and general inquiries.

- **Issue Tracking System (GitHub Issues):**
  - This is the primary channel for reporting technical bugs (e.g., crash logs, rendering errors) and feature requests.
  - Users and developers can track the status of reported issues (Open, In Progress, Resolved) directly within the repository.
- **In-App Reporting:**
  - For content-specific issues (e.g., inappropriate models or copyright violations), users can utilize the "Report Issue" feature located in the *Asset Detail* menu.
  - These reports are logged server-side (*asset_reports.log*) and reviewed by administrators for moderation.