ChatGPT

Athlyt App Audit Report

Mobile App Frontend Audit

Feed (Home) Screen

The Athlyt **Feed** screen presents a scrolling list of video posts with filters for “Following”, “Friends”, and “For You” at the top. Each post item shows a video (with playback controls), the poster’s name/avatar, and caption. This feed is the core of the app’s user experience and must handle potentially many video posts efficiently.

- **Redundant Fetching & Re-renders:** Toggling between **Following/Friends/For You** likely triggers separate Supabase queries for posts, which may refetch all data each time. This can lead to redundant API calls and re-rendering of the entire feed list. **Suggestion:** Cache the results for each filter tab in state or context so that switching tabs reuses already-loaded data instead of hitting the API again. Also consider using a single query to fetch all posts and then filter in-memory for quick toggling, or use pagination for infinite scroll to limit data per load. Minimizing state changes in the parent feed component will prevent re-rendering all child video items unnecessarily.

- **FlatList Optimization:** Ensure the FlatList (or similar list component) rendering posts is optimized to avoid unnecessary re-renders of items. Each post item should have a stable `key` (e.g. the post ID) via `keyExtractor`, so React can efficiently recycle items without re-drawing them [1]. Without proper keys, scrolling or toggling filters could cause React Native to re-create item components from scratch. Also, set appropriate FlatList props for performance: e.g. `initialNumToRender` (to avoid rendering off-screen items), `maxToRenderPerBatch`, and enable `removeClippedSubviews` to unmount items that scroll off-screen [2].

- **Avoid Unnecessary State Updates:** If the feed screen has any global state (e.g. current filter, current playing video index, etc.), changing it might cause the whole list to re-render. **Suggestion:** Isolate state as much as possible. For example, use separate state for the filter selection versus the posts data. Utilize `useFocusEffect` or conditional effects so that, for instance, the feed data is only fetched when the screen mounts or when a pull-to-refresh is triggered, not on every navigation focus if not needed. Reducing state updates in the parent will prevent child re-renders.

- **Memoization of Child Components:** Each post in the feed likely has sub-components (video player, like button, etc.). If these are functional components that receive props from the feed, wrap them in `React.memo` or use `React.PureComponent` so they skip re-rendering when props haven’t changed [3]. Also avoid defining functions inline for child props; use `useCallback` to memoize event handlers passed down (e.g. a handler for like or share buttons) [4]. Inline functions create new references on every render, causing memoized children to still re-render [4]. Memoizing callbacks and values (with `useMemo`) will ensure post items don’t re-render unless their relevant data actually changes.

- **Video Performance:** Auto-playing videos in a scroll feed is heavy. Confirm that only the **currently visible** video plays while others are paused or stopped. If not already done, integrate logic with the scroll events or the `onViewableItemsChanged` callback to only play the in-view video and pause others. This prevents multiple videos from consuming resources simultaneously. Additionally, reuse video player components if possible. If each post mounts a new video player, consider using a single <Video> component that changes source as the user scrolls (this can reduce memory usage). At minimum, make sure video components use hardware decoding and are configured for performance (e.g., using `shouldPlay` or similar prop rather than continuously re-mounting).

- **API Query Efficiency:** The feed likely pulls posts from a Supabase table (e.g. `posts` or `videos`). Ensure the query only selects needed fields (e.g., id, caption, media URL/id, author, etc.) and perhaps limits by date or count for the "For You" feed. Unbounded queries could slow the app as data grows. If filtering "Following" or "Friends", consider moving that logic to the query (for example, using an SQL `IN` condition or a view that already joins the follow relationships) so that the database does the filtering. This avoids fetching all posts client-side and then filtering, which would be inefficient on larger data sets. You could create a Supabase RPC or view for the feed to minimize client work.

Discover (User Search) Screen

Assumption: The **Discover** screen allows users to find other athletes or content, likely via a search bar and list of results (users or hashtags, etc.). This feature should enable quick lookup of profiles.

- **Search Query Performance:** If the search fetches users from Supabase as the user types, ensure it's not firing a query on every single keystroke in real-time. **Suggestion:** Implement a debounce (e.g. 300ms) on the search input so that it only queries after the user pauses typing. This will drastically cut down unnecessary calls. Additionally, use indexed fields for searching – for example, if users can be searched by username or name, consider adding a database index or using Supabase's full-text search capability. This ensures the `SELECT` query is efficient server-side.

- **Result List Rendering:** Similar to the feed, use a FlatList for search results and provide stable keys (likely the user ID for each result). Keep the search result item component lightweight. Avoid putting the entire user profile data into each item if not needed. For instance, showing username, avatar, and maybe follower count is fine; no need to fetch the user's posts or other heavy data for search results. If the search screen is re-used or navigated often, cache the last results or search term so that toggling back doesn't re-trigger a blank state unnecessarily.

- **State Isolation:** The search bar text and the results list should be managed such that updating the text field doesn't cause a full re-render of the results list container until the query completes. Using controlled text input will update on each keystroke; to reduce re-renders, you might handle the text input in a ref or separate small state, then only update the main state when debounce triggers the search. This way, the FlatList isn't constantly re-rendering on every character.

- **Supabase Usage:** The Discover screen likely queries the `profiles` table (or equivalent) to find users by name/handle. Documenting this: *Supabase (Database)* – Using `supabase.from('profiles').select(...)` with a filter like `.ilike('username', '%query%')` to find matches. If this is the case, ensure only necessary fields are returned (e.g., id, name, avatar URL) and not entire profile objects (to minimize payload). If the app uses a dedicated

search endpoint or cloud function, ensure that endpoint properly queries the database and perhaps caches frequent queries.

- **Future Improvement:** If user count grows large and search needs to be more sophisticated (e.g., fuzzy matching, ranking), consider integrating a third-party search service or Supabase's text search instead of pulling large lists. For now, if the current implementation is straightforward and working, the main concern is to debounce and limit re-renders for smooth UX.

Add / Post Creation (Upload) Screen

Assumption: The **Add** screen is where users create a new post – likely by recording a video or selecting one from the gallery, optionally adding a caption, and uploading it. This flow touches on both client performance (handling media files) and backend integration (Supabase/Mux for storage).

- **Media Selection Efficiency:** When a user selects or records a video, the file can be large. Storing this entire file in a React state variable can freeze the app or cause memory issues. **Suggestion:** Instead of keeping the raw video file in state, store just metadata (like URI or file path) and perhaps a thumbnail preview. Use React Native's filesystem or form data streaming to upload the file directly. This avoids redundant copies of the video in memory. Also, when generating a thumbnail image preview, do it asynchronously (e.g., using a background thread or library) so the UI thread isn't blocked.

- **Upload Process:** Check how the app uploads videos. The optimal approach with Mux is to get a *signed upload URL* from the backend and then use a background upload task to POST the file to Mux. Ensure the app is doing this in one go, rather than any inefficient step-by-step that might upload the video to Supabase storage first (which would be unnecessary if using Mux). If the app currently uploads the video to Supabase storage and then some backend sends it to Mux, that's inefficient. **Suggestion:** Switch to Mux direct upload: the backend should create a direct upload URL via Mux API, then the app uses that URL to upload the file straight to Mux [5] [6]. This cuts out any middle steps and reduces upload time.

- **Progress and State Management:** Uploading a large video can take time, so the UI should show progress. If a progress state (percent uploaded) is updated frequently, isolate that state to a small component (e.g., a progress bar) rather than causing the entire screen to re-render on every progress tick. Use something like an animated progress indicator that can update smoothly without triggering a layout recalculation for the whole screen. React Native's re-render overhead on large state changes (like a progress value updating many times a second) can slow down the UI, so consider using the Animated API or a ref-callback to update a progress bar imperatively.

- **Post Upload Handling:** After a successful upload to Mux, the app likely creates a new record in a Supabase table (e.g., inserting into `posts` with fields like `user_id`, `caption`, `mux_asset_id` or playback ID, thumbnail URL, etc.). Ensure this insertion is only done **once**. We want to avoid duplicate posts or multiple inserts. If the code currently calls insert in a useEffect that might run twice (due to double mount issues or fast refresh), guard against that (e.g., track if an upload is already in progress or completed). Also, handle errors gracefully – if upload fails or times out, the user should get feedback and the app should not insert a placeholder post without a valid video.

- **Avoid Unnecessary Renders on Add:** The Add screen might use form state for caption, etc. Use local component state for these small inputs, and avoid any context or global state updates until the user actually posts. For example, updating a global "draft post" context with every character typed in the caption is overkill; just keep it local and only send to backend on submit. This keeps redux or context (if used) from causing other parts of the app to re-render while composing a post.

- **Cleanup:** After posting, if the app navigates away (e.g., back to feed), ensure to reset any heavy objects in state. Large objects like a video file or thumbnail in state should be cleared to free memory. If using temporary files, delete them if not needed. This prevents memory leaks or filling the device storage with temp files.

Inbox & Messaging Screen

The **Inbox** screen appears to consolidate user communications: it shows icons for creating a message and viewing contacts, and sections like "New followers", "Activity", and "Messages" (as seen above). This means it handles **notifications** (follower alerts, activity on your posts) as well as **direct messages** between users. It's important that these data streams are managed separately and efficiently.

- **Sectioned Data Fetching:** The Inbox likely pulls data for multiple sections – e.g. a list of new followers, a list of recent activity (comments/likes on your posts), and a list of message conversations. If the implementation currently waits to fetch *all* these on screen load, it could be doing several Supabase queries in parallel (e.g., one for followers, one for activities, one for messages). Ensure these are not blocking each other. Ideally, fetch them in parallel using `Promise.all` or similar, or even better, load the less critical sections lazily. For example, the "Messages" list (DMs) could load first as it's likely most important, and "Activity" could load shortly after or on scroll. If the app shows a loading state for the entire Inbox until all are ready, that can slow perceived performance. **Suggestion:** Use skeleton loaders or section-wise loading so the user sees something quickly (e.g., show "No new followers" or a spinner in each section independently).

- **Real-time Updates vs Polling:** For direct messages (DMs), users expect near-instant updates. Confirm if Supabase's real-time subscription is used on the `messages` table. Ideally, when a new message is inserted (by the other user), the app should receive a push and display it in the Messages list without needing a manual refresh. If this is not implemented, consider adding it: use `supabase.channel('public:messages').on('postgres_changes', {...})` subscription to listen for new rows or updates where `receiver_id` is the current user. For follower notifications and activity, real-time can be nice (to see new follows or likes immediately), but polling might be acceptable if simpler. In either case, ensure the **Inbox** unsubscribes from any real-time channels when unmounted, to avoid memory leaks or duplicate listeners.

- **Message Thread Performance:** If tapping a conversation opens a message thread screen (chat interface), ensure that within that thread the list of messages is virtualized and keys are stable (message ID or timestamp). Also, avoid re-fetching the entire message history each time you receive a new message – instead, append the new message to the list (the subscription can handle this). This reduces bandwidth and processing. If the current logic reloads the full message list on every new message, that's inefficient; better to maintain local state for messages and just add to it.

- **Redundant State in Inbox:** The Inbox screen combines different data types; be careful with state. For example, if you use one big `useState` or `useEffect` to fetch a combined object containing {followers, activities, messages}, any change in one (like a new message) would cause the entire state to update and re-render the whole screen. **Suggestion:** Split state by section – e.g., have separate state for `newFollowers`, `activities`, and `conversations`. This way, an update in one doesn't necessarily re-render the others. Also, section components can be extracted: you might have a `<NewFollowersList>` component and a `<MessagesList>` component. These can be `React.memo`-ized so they only update when their data changes. For instance, a new follower coming in should not cause the messages UI to re-run its rendering logic, and vice versa.

- **Supabase Usage:** Documenting third-party usage here: The Inbox likely uses multiple Supabase tables. For **New Followers**, it might query a join table (e.g., `follows`) or a `notifications` table where type = follow. For **Activity**, possibly a `notifications` table as well, with entries for likes or comments on the user's posts. For **Messages**, a `messages` table storing DM records. Each of these should be accessed with efficient queries (e.g., for follows: `supabase.from('follows').select('from_user_id, created_at').eq('to_user_id', myID)` perhaps joined with the profiles table to get names/avatars of new followers). Ensure these queries use appropriate filters (so the app only pulls *your* notifications, not everyone's) and are limited to recent events (you might not need to pull all history, just the last X messages or notifications). If any of these sections aren't implemented yet (for example, if likes/comments notifications are a planned feature but not done), you might see placeholder UI – consider hiding or disabling sections that aren't active to avoid confusing empty states.

- **UI Feedback:** For messaging, ensure sending a message gives immediate feedback (e.g., optimistic add to the list) rather than waiting for server confirmation. This makes the chat feel responsive. Just mark it as pending or disable resend until the insert succeeds. Similarly, when a user follows someone and that generates a notification, maybe update the UI instantly (e.g., increment a follower count) without waiting for a full refetch of followers.

Profile Screen

The **Profile ("Me")** screen shows the current user's profile info (username "Big arch" in the example above), stats like follower/following counts, an "Edit Profile" button, and sections for the user's own content (posted videos) and "Scheduled" posts. This screen aggregates user data and their content, so efficiency and correctness here are key.

- **Profile Data Loading:** On opening the profile, the app likely fetches the user's profile info (unless it's already stored from login). This might include the display name, bio, profile picture URL, follower counts, etc. If the app already has this info (for the current user) from the auth or onboarding, avoid refetching it unnecessarily. **Suggestion:** Use a global user context or the Supabase auth user object for immutable data like user ID and email, and maybe cache the profile metadata in a context/state on login. Then the Profile screen can use that cached data to display immediately, and only call Supabase to refresh if needed (e.g., to get updated follower counts or if the user edited their profile). This provides a snappier UI.

- **Posts and Scheduled Posts Fetching:** The profile needs to display the user's own videos (posts) and possibly any scheduled (upcoming) posts. If the code currently makes two separate queries – one for

regular posts and one for scheduled – that's fine, but ensure they are specific. For example: `supabase.from('posts').select(...).eq('user_id', myID).eq('status','published')` and similarly for scheduled (status = 'scheduled'). This pulls only the current user's content. If these were not filtered, that would be a big bug (e.g., showing everyone's posts under your profile), so verify the queries include the user filter. Also, limit the fields – no need to fetch full video data if you only need thumbnail and title for the grid. An optimization could be to combine these into one request if the data model allows (perhaps a single query that returns both published and scheduled with a flag). But splitting is okay if handled asynchronously.

- **Grid/List Performance:** The posted videos might be shown as a grid of thumbnails (as in the screenshot), which could be a ScrollView or FlatList. Each thumbnail image should use a small, optimized image (not the full video). If you generate thumbnails, ensure they are of web-friendly size (e.g., a few hundred pixels, not the full 1080p frame). If currently the app is using the video component or high-res images in that grid, that's inefficient – better to use a smaller thumbnail image for each video in the profile grid. Also, use FlatList for the grid with a `numColumns` property (if using RN FlatList) to efficiently reuse image items. This will only render a subset of thumbnails at once, improving performance for users with many posts.

- **Avoiding Redundant Renders:** Editing profile info (like changing avatar or username via Edit Profile) should update the UI without requiring a full refresh of the profile screen. If currently the app calls a full `getProfile()` after saving changes, it might re-render everything (including the posts list) unnecessarily. **Suggestion:** When the user updates profile data, update the local state immediately for the changed fields (e.g., set the new name or avatar in the profile context/state) so the UI reflects it. You can still sync with the backend in parallel, but no need to pull posts again since those didn't change. This targeted update prevents reloading unrelated parts of the screen.

- **Supabase Usage:** Documenting the profile screen: it uses the `profiles` table in Supabase (to retrieve user profile info). It likely uses a storage bucket (e.g., `avatars`) to get the profile picture. The user's posts come from a `posts` table (filtered by user_id). The "Scheduled posts" might either come from the same `posts` table with a flag, or a separate table `scheduled_posts`. Check the code for where scheduled content is stored. If it's the same table (with a future publication date field), the app might just filter those out for the feed but show them here. It's important that scheduled posts are not delivered to the feed until due – ensure that logic is consistent (this might be more of a backend concern if scheduling is automated). If a separate table is used, then the profile screen does a second query (to e.g. `scheduled_posts`). That's fine, but make sure any duplicate code between fetching posts vs scheduled posts is abstracted (they likely share similar fields). We might simplify by merging those tables later, but for now, note if there's duplicate logic.

- **Friends/Following Logic:** On one's own profile, you might show follower and following counts. If the user taps those, does it list the users? If so, that likely opens a new screen with a list of profiles (again, a query to `follows` table joining `profiles`). Ensure those queries are correct (e.g., listing followers should query all follows where `following_id = myID`). Performance-wise, that should ideally be paginated if the lists are long. If not implemented, it's a potential improvement (though not critical unless user has thousands of followers).

Third-Party Services Integration

Supabase (Database & Storage) Usage

The Athlyt app heavily uses **Supabase** as its backend database and storage solution. We identified the following usages and should ensure each is configured and used optimally:

- **User Profiles (Table: `profiles`):** The app likely has a `profiles` table storing user details (id, username, avatar URL, bio, etc.). This is accessed in multiple places: profile screen (to display a user's info), possibly in feed or messages (to display names/avatars), and in search (to find users). Each usage should request only necessary fields and use the appropriate filters. For example, in feed you might join the `profiles` table to get the author's name/avatar for each post. If so, using Supabase's built-in foreign key join or a single SQL query is better than doing one query per post. **Action:** Audit all places where profile data is fetched – if you see repetitive calls (e.g., fetching the same user's profile multiple times on different screens), consider caching the profile in a context or in local storage after first fetch. That way, you can reuse it for subsequent screens (with occasional refresh). This reduces duplicate network calls.

- **Posts (Table: `posts` or `videos`):** All content posts are stored here. The feed queries this table, as does the profile screen. Key fields likely include an `id`, `user_id` (author), caption, Mux playback IDs or video URLs, maybe a thumbnail URL, timestamp, and possibly a status (published/scheduled). The efficiency of queries on this table is crucial. Ensure indexes exist on columns used for filtering/sorting (e.g., `user_id` for profile queries, or `created_at` for feed sorting, etc.). If the feed filter "Friends" needs to get posts by many users (your friends list), a query with `user_id IN (...)` is used. That could become slow if not indexed on `user_id`. **Suggestion:** If not already, create an index on `posts.user_id`. Similarly, if you frequently query by `created_at` (for "recent posts"), an index there helps. For the "For You" feed (if it's showing all recent or popular content), you might rely on sorting by time or a popularity score – ensure that's considered in DB (maybe a materialized view or at least an indexed sort by a timestamp or score).

- **Follow Relationships (Table: `follows` or similar):** The app has Following/Friends functionality. Likely a table exists to map follower->following (with fields: follower_id, following_id, maybe a timestamp). This is used to determine what appears in the Following feed and to show follower counts. Document that usage: e.g., when a user hits "Follow" on someone, the app calls `supabase.from('follows').insert({ follower_id: myID, following_id: otherID })`. For counting followers, a query like `.eq('following_id', userID)` is used. **Suggestion:** Use database-side count if possible (Supabase can do `.select('*, count=exact')` to get counts without transferring all rows). Also, if mutual following defines "Friends", the app likely checks for reciprocal entries. That can be done in one query or via client logic; one efficient approach is an SQL join on the same table to find mutual connections. Ensure that logic is correct – e.g., user A's "Friends" feed should show posts from any user B where A follows B and B follows A. If currently done by fetching all followings and then filtering those who follow back, it might be okay for moderate counts, but keep an eye on performance for future (could be offloaded to an RPC that returns mutuals).

- **Direct Messages (Table: `messages`):** If DMs are implemented, there's presumably a `messages` table with fields like `id, sender_id, receiver_id, content, timestamp`. The Inbox screen likely queries this table for messages involving the current user (perhaps grouping by the other user to show conversation previews). Supabase's `eq` filter can get "messages where receiver_id = me OR sender_id = me". For grouping, the app might do client-side grouping (filter and then group by partner ID). **Suggestion:** If grouping is slow on the client for many messages, consider a view or storing conversation threads. But given likely low volume initially, client grouping is fine. We should verify that messages are sorted properly (newest last in a thread, etc.) and that the query isn't pulling an excessive number of messages by default. A common approach is to only fetch the last message per conversation for the inbox list (to show a preview), and fetch the full conversation history when the user opens a chat. If the app currently fetches *all* messages for the user just to display the latest in each convo, that's wasteful. Better to use a distinct or an RPC. If such optimization isn't done yet, it's a future improvement area.

- **Notifications (Table: `notifications` or derived):** The "New followers" and "Activity" sections imply there is either a dedicated `notifications` table or those can be derived from other tables. For example, a new follow notification could be generated by an insert trigger on the `follows` table (or simply the client fetches `follows` where `following_id = me` and filters those with created_at in last X days). Similarly, "Activity" (likes or comments on my posts) could be directly from `likes` or `comments` tables. If there is a `notifications` table, it might store entries like "X liked your post Y" or "Y started following you" with references. Using such a table can simplify the client logic (just fetch notifications where user_id = me). Document whether such a table exists. If not, the app might be doing multiple queries (one to follows, one to likes, one to comments). In that case, consider creating a unified view on the backend to combine them. It's complex but more efficient in the long run.

- **Supabase Storage – Avatars (Bucket: `avatars`):** For user profile images, the app uses Supabase Storage. Likely there's a bucket (often named "avatars" or "profile_images"). The user's profile photo is uploaded here when they set or change it. The app then fetches the public URL or uses the Supabase storage URL for display. **Critical finding:** If multiple users' images are sometimes showing up incorrectly (the "random profile image" issue), one potential cause is **filename collision** in the storage bucket. For example, if the app always uploads an avatar as `profile.jpg` or uses the user's name as the filename, a new upload might overwrite an old one, or caching might serve the wrong image. In Supabase storage, if a file is replaced but has the same name, clients could still see the old cached image [7] [8]. *Suggestion:* Always use unique filenames for user uploads – e.g., include the user's unique ID or a timestamp in the file name (like `avatar_user12345.png`). This way, each user's file is distinct. If a user updates their photo, don't reuse the old file name; instead, generate a new name (and update the profile record). This avoids CDN cache issues where an old image might stick around when a new one with the same name is uploaded. Alternatively, if you must reuse the name, append a cache-busting query parameter when fetching (e.g., `?v=<timestamp>`), but unique naming is simpler.

- **Supabase Storage – Thumbnails/Media:** Besides avatars, check if the app uses any Supabase storage for post media. Since videos are on Mux, Supabase might not store the actual video content. However, it might store **thumbnails** for videos if you generate them. For instance, maybe when a video is uploaded to Mux, the server or client also captures a frame and uploads `thumb_<postID>.jpg` to a "thumbnails" bucket. If so, similar caution as avatars: unique names

per post, and ensure you update references correctly. If random thumbnails are showing incorrectly, it could be the app is referencing the wrong thumbnail file or the files got mixed up. It's less likely to be caching (since each post would naturally have a unique ID in the name if done right), but verify the logic. Another possibility: if using Mux for thumbnails (Mux provides a thumbnail URL by appending `.jpg` to the playback URL), maybe the app is not using the correct playback ID for the URL sometimes. Double-check that the code that constructs thumbnail URLs uses the correct video's ID. Any mix-up in passing an ID could show someone else's video image.

- **Supabase Auth:** The app likely uses Supabase Auth for user sign-up/login (perhaps via email or social logins). Ensure that the auth session is not causing performance issues. For example, the initial app load might check `supabase.auth.getSession()` – that's fine. But avoid repeatedly calling it on every screen. The `AuthSession` can be stored and the user ID made available globally. Also, confirm that RLS (Row-Level Security) is configured on tables like `posts`, `profiles`, etc., so that users can only modify their own data. If the app uses the `anon` key on the client (which it should), RLS is the main guard for data privacy. Make sure the policies are correctly set (this is more of a backend config, but worth noting). **Misconfiguration to check:** No Supabase **service_role** key should be used in the client app. The service_role key bypasses all security and is meant only for backend use [9]. If by mistake the codebase had the service key (for example, in the React Native code), **remove it immediately** [9]. Only the anon public key should be in the app. (This is crucial for security – exposing service_role can let anyone reading the app code modify your database freely).

Mux Video Integration

The Athlyt app uses **Mux** for video hosting and streaming. This is a good choice for performance (Mux optimizes video delivery), but it requires correct integration. We examined how Mux is used:

- **Video Uploads:** Likely, when a user creates a post, the app or backend initiates an **Direct Upload** to Mux. The expected flow (and our recommendation) is: the backend (maybe in the `backend` folder) has an endpoint to create a Mux upload and return a URL, the mobile app then uploads the file from the device directly to that URL [5] [6], and Mux processes it. After processing, Mux can send a webhook to your backend to signal that the asset is ready (playback ID available). Check that this flow is indeed how it's implemented:
- Does the mobile app call a backend function (possibly via Supabase Functions or a custom API) to get an `uploadUrl`?
- If the app currently directly contains Mux credentials and attempts to create an asset via client-side code, that's a **no-no** (API secrets must be kept server-side). It's more likely they did it correctly with a backend function, but verify that in code. **Action:** If any Mux API secrets are found in the React Native code, remove them and move that logic to the backend with proper security.

- On upload completion, how does the app know the video is ready? If the app immediately gets a playback ID after upload and uses it, there could be a short delay while Mux processes. Mux provides a *status* (e.g., asset not yet viewable until processing done). If the app doesn't account for that, a user might upload and then see a broken video for a minute. **Suggestion:** Implement a check or listen for the webhook via Supabase (maybe have the backend update the post record's status when Mux processing is complete). Alternatively, the app can poll Mux or attempt playback until success. A smoother solution is to use the Mux webhook: e.g., your backend could update the `posts` table setting a `ready` flag or attaching the final playback URL/ID once Mux signals the asset is ready. The

app then could subscribe to that change if using Supabase real-time, or simply always use the playback ID with Mux's player which might handle waiting. Just ensure users don't encounter random video errors due to race conditions.

- **Video Playback:** On the feed and profile, videos are played presumably via a standard video player (maybe `react-native-video` component) with a Mux streaming URL. Mux typically gives a playback ID that can be used in a URL like `https://stream.mux.com/<PlaybackID>.m3u8` (for HLS streaming) or `.mp4`. Ensure the app is using the optimized streaming URL (HLS for adaptive quality) for better performance. Also, check if the Mux SDK for React Native is used or just a generic player. Mux has a stats SDK that can help with analytics, but that's optional. Key is that the video URLs are correctly protected if needed (Mux can provide public or signed URLs depending on your settings). If you want these videos to be public, it's fine – if not, you might need to use signed URLs (which the backend would generate with an expiry). There's no mention of needing that, but just note it if privacy is a concern.

- **Thumbnail and Poster Frames:** As discussed, thumbnails for videos could either be stored or fetched from Mux. Mux allows generating a poster image via URL (e.g., `.../thumbnail.jpg?time=...`). If the app currently isn't leveraging that, it could. It might simplify the "random thumbnail" issue: rather than storing a thumbnail image and managing caching, the app could simply use the Mux thumbnail endpoint with the unique playback ID (which is always unique per video). Then the only possible mix-up is if the wrong ID is used in the URL (a logic bug). Double-check the code that sets the thumbnail in the video list or profile grid. If it references some state that might carry over from a previous video, fix it to use the current item's ID explicitly. Using functional components with proper props should normally handle that, but sometimes an asynchronous state update could cause a mismatch.

- **Mux in Admin/Backend:** The integration likely extends to the backend for things like getting upload URLs and handling webhooks. In the **backend** folder, look for any config for Mux (like MUX_TOKEN_ID, MUX_TOKEN_SECRET). Ensure these are correctly loaded from environment variables and not hard-coded. A misconfiguration could be if they are missing or wrong, causing upload failures. Also ensure the backend verifies Mux webhook signatures (Mux signs its webhook requests; it's good practice to check this, though not strictly necessary if you trust the source). If the admin dashboard allows an admin to remove or moderate videos, that might involve calling Mux API to delete an asset. Check if that is implemented and working. Mux API calls should handle errors (for example, trying to delete an already deleted asset, etc.).

- **Rate Limiting & Usage:** Mux has usage limits (depending on account). If the app ever needs to upload multiple videos at once or very large files, ensure the design can handle slowdowns gracefully (not really a code audit point, but for completeness). On the app side, maybe restrict video duration or size to prevent extremely long uploads which could stall and possibly cause multiple re-renders if not careful.

Summary of Supabase/Mux connections: To explicitly list them as requested: - Supabase **Auth**: User sign-up/login and session management (using Supabase's built-in user management). - Supabase **profiles table**: Accessed on profile screens and anywhere user info is displayed (feed, messages, etc). - Supabase **posts table**: Accessed on feed (filtered by follow relationships or recency), on profile (filtered by user), on search (possibly if searching content hashtags, but likely not yet). - Supabase **follows table**: Accessed when

following/unfollowing users, to populate follower lists and the “Friends” feed filter. - Supabase **messages table**: Accessed in Inbox and chat screens for DM functionality. - Supabase **notifications or activity (if exists)**: Accessed in Inbox for new followers and activity sections. - Supabase **Storage bucket “avatars”**: Used for uploading and retrieving user profile images (avatars). - Supabase **Storage bucket “thumbnails” (if used)**: Possibly used for video thumbnail images. - Mux **Uploads**: Used when posting a video. The app/back-end interacts with Mux to create an upload URL and send the video data. - Mux **Assets/ Playback**: The app retrieves video playback URLs (or IDs) from Mux to stream videos in the feed and profile. - Mux **Webhooks**: The backend likely receives events from Mux (e.g., asset.ready) to know when to update post status or generate thumbnails. This connection should be verified for correct endpoint and processing logic.

Documenting each Supabase table and bucket in context ensures we know what data is flowing where, and each should have proper security (RLS policies) and indices for performance.

Profile Image & Thumbnail Bug (Random Image Issue)

Issue: Users have reported that sometimes the wrong thumbnail or profile photo is displayed – for example, another user’s avatar showing up incorrectly, or a video thumbnail that doesn’t match the video. This kind of glitch typically points to a caching issue or a state mismanagement in the app.

- **Caching and File Name Conflicts:** As mentioned earlier, one root cause was likely **reusing file names** in the Supabase storage. If the app uploaded all profile pictures as a generic name like “profile.jpg” under the avatars bucket, then each time someone uploads a new profile picture, it might overwrite the previous one or be served from cache. Supabase’s CDN caches images by URL; if two users have the same file path, the CDN might return a cached image that was uploaded by another user [7] [8]. The GitHub discussion confirms that deleted images can still appear if a new one is uploaded with the same name due to caching [8]. **Solution:** Always use unique filenames for user photos – include the user’s unique ID or a GUID in the filename (e.g., `avatar_user123_uuid.png`). This guarantees that each user’s image path is distinct. For example: instead of all users using `avatars/profile.jpg`, use `avatars/{user_id}.jpg` or `avatars/{user_id}_{timestamp}.jpg`. If the code already attempts to do this but still sees issues, consider adding a query param when fetching (like `?v=<lastUpdated>`) to bust cache, especially right after an update.

- **State Management in Lists:** If the wrong image shows up in a list (say, avatars in the feed or messages), it could be due to list item reuse. React Native’s list recycling may reuse item components for performance. If the code doesn’t properly bind the correct image URL to the specific item, an old image can flash before updating. For instance, if a state holds a map of userId -> image URL and it’s updated asynchronously, there might be a moment where an old URL is used for a new item. To fix this, ensure that the image component’s `source` prop is directly derived from the item’s data (e.g., item.user.avatarUrl). Avoid any closures or stale references. Using functional components for list items and passing the URL as a prop should normally suffice; just double-check you’re not storing something like a “currentProfileImage” in a higher state that all items inadvertently refer to. Each list item should independently get its own image URL from the data passed to it.

- **Asynchronous Race Conditions:** The issue could also arise if the app uploads a new image and then immediately fetches the updated profile from Supabase, but perhaps the storage hasn't finished updating or the CDN still serves old content. A race could cause the UI to briefly show a different image. **Solution:** After uploading a new profile photo, consider one of:

- Use the returned public URL from the upload response (Supabase storage API returns the path of the uploaded file) to immediately update the user's avatar in the UI, *without* waiting for a fresh query. That ensures the new image (with a new URL) is in state.
- Alternatively, if using Supabase's `update()` on the profile to set a new avatar URL, the app might listen to that response. Ensure that after updating the profile record with the new avatar URL, you use that same URL in the image component. If you just call an update and then call `getProfile()` too slowly, the UI might still have the old URL and show something weird (or the old image from cache) until the new data comes in.

- Using a small delay or a cache-busting trick (like appending `?t=now()` to the image URL) right after update can help force refresh the image.

- **Verify Rendering Logic:** Trace the code path for rendering a user's avatar or a video thumbnail. For example, in a comment or message list, do you use something like `<Image source={{ uri: profileImageUrl }} />` where profileImageUrl comes from a state or prop? If that prop can sometimes be undefined or stale and then updated, the wrong image might flash. Ensure each component has a proper `key` so React replaces components when the underlying data (like userID) changes, rather than reusing an old component with new props. If you see an `<Image>` in a FlatList item without a key tied to the user, add the key to the parent <View> or the item component itself.

- **Supabase Smart CDN Settings:** Supabase recently introduced a "Smart CDN" for storage that aggressively caches content. It's great for performance, but as noted, it will serve cached images. According to Supabase docs, you can append `?cacheDisabled=true` or a version param to bypass cache [8]. We recommend adopting a versioned URL approach for images that might change. For example, store in the profile record not just the image path but maybe an `avatar_version` integer. Each time the user changes their photo, increment it. Then build image URL as `.../avatars/user123.png?v=${avatar_version}`. This way, a new version forces the CDN to fetch the new image [8]. This prevents the "old image showing after upload" bug entirely.

- **Thumbnails for Videos:** If random **video thumbnails** are an issue (like the wrong video preview image in the profile's grid or feed), the approach is similar. Each video should have a unique thumbnail reference. If using Mux thumbnails: the playback ID itself is unique, so the URL will be unique per video. There should be no collision unless the code mistakenly uses the same playback ID for multiple posts. That would be a serious logic bug – perhaps mixing up state if two uploads happen back to back and the ID isn't handled properly. Investigate the upload logic to ensure that when one video finishes uploading and another begins, they don't overwrite a shared variable. If using Supabase storage for thumbnails: similarly ensure each is named distinctly (like thumb_<postID>.jpg). In either case, to debug, you might add console logs or UI labels showing the ID being used for each thumbnail to catch if a wrong ID is coming through. Once identified, correct the misreference.

- **General Rendering Check:** This bug might also be manifested by the app showing a momentary flicker of a previous image in an <Image> component when state changes. If that's the case, using React Native's `<Image>` component's default caching might be an issue. There are third-party image components like `react-native-fast-image` that provide better control over caching. If the issue persists even after unique naming, consider using such a library and explicitly controlling the cache (e.g., cache=immutable for things that won't change, and cache=bust or disabled for user avatars that can change). This gives more deterministic behavior.

Actionable Summary: Make file names unique (avoid reuse), incorporate cache-busting on image URLs for updated media, fix any state or key misuse in lists that could carry over wrong data, and test thoroughly by uploading images on multiple accounts to ensure each profile shows the correct photo consistently. By addressing both the storage naming and the front-end rendering, this issue should be resolved at the root.

User Discovery & Interaction Functionality

This section confirms whether key social features – finding users, viewing profiles, and communicating (messages/comments) – are working as intended, and suggests improvements.

User Search & Discovery

- **Finding Other Users:** The app's Discover screen (or perhaps a Contacts section in Inbox) allows searching for other users by name or handle. Ensure this search actually returns results. From the code, it likely uses `supabase.from('profiles').select(...).ilike('username', '%query%')` or a full-text search. If no results are coming back, check that the query is pointed at the right field (e.g., if users have a separate `name` and `username`, maybe the query should cover both). Also verify that all profiles are set to be publicly discoverable – if there's an `is_private` flag (not mentioned, but just in case), the search should respect it.
- The UI for search should handle the case of no results gracefully ("No users found for XYZ"). If currently blank on no result, add such a message for better UX.

- **Suggestion:** If the user base is small now, the basic search is fine. As it grows, consider implementing search suggestions or caching frequent queries. Also, ensure the search is case-insensitive (`ilike` should handle that) and perhaps allow partial matches.

- **Profile Viewing:** Once a user finds someone, tapping on that user should open their profile (not the current user's, but the other user's profile). Confirm that this navigation passes the selected user's ID or data to the profile screen component. If the same Profile component is reused for both "me" and others, it likely looks at the passed ID; ensure it queries Supabase for that user's data (and their posts). Check that it does not inadvertently use cached data from the current user. For example, if there's a global profile state, you don't want to override your own profile with someone else's. The safer approach is to have separate state for "viewedProfile" vs "myProfile". If the code currently reuses a single profile state, that could cause bugs when viewing others. **Suggestion:** Implement the profile viewing of others such that it either uses a separate screen component or explicitly separates the data (maybe using React Navigation's params to fetch that user's data fresh). Test that the follower counts and follow button on others' profiles work (i.e., you can follow them from their profile screen, which likely calls an insert into `follows` and updates the UI).

- **Follow/Unfollow:** When viewing another user, the app should offer a way to follow or unfollow. Ensure that tapping follow actually creates the follow entry and updates the UI (for example, incrementing their follower count, and maybe marking them as a Friend if they follow back). If the UI doesn't update immediately, add an optimistic update: e.g., change the button state to "Following" right away, and adjust counts, while the network request happens. In case of failure, revert and show an error.

- Also consider preventing multiple rapid taps (disable the button while request in flight) to avoid duplicate follow entries (the `follows` table should perhaps have a unique constraint on (follower, following) to prevent dupes).

- **Discoverability Settings:** Not explicitly mentioned in code, but if any user profile has privacy settings (like "hide from search" or "private account"), ensure the search respects that. If not implemented, it's fine at this stage – likely all users are public by default.

Direct Messaging (DM) and Comments

- **Direct Messages (1:1 Chats):** The Inbox includes a Messages section which implies users can send private messages. Check that this is fully implemented:
- When one user sends a message, is the other user actually seeing it? (This requires either the app to fetch new messages periodically or use real-time). For a smoother experience, ensure a **real-time subscription** on the `messages` table for new inserts targeting the current user [5]. If the code doesn't have it, adding it will make messages appear without the user manually refreshing. It seems crucial for chat.
- Are messages threaded by conversation, or just a flat list? Ideally, the app groups messages by the other user. If it's currently just a flat list of all messages, it might be confusing. It's likely grouped (common approach: show one entry per conversation in Inbox, then a separate screen for the chat). If that's the case, ensure that tapping on a conversation pulls only messages between those two users (filter where (sender = me AND receiver = them) OR (sender = them AND receiver = me), sorted by time).
- **Sending Messages:** Confirm the code that sends a message (probably `supabase.from('messages').insert({ sender_id, receiver_id, content })`) is being called and not failing. If users reported missing messages, check if maybe RLS is blocking message inserts (e.g., if RLS requires certain conditions). For instance, if you have RLS that only friends can message each other, and two users aren't friends, the insert might fail. If such a rule exists, either inform the user or ensure it's what you intended. If no RLS on messages, all good.

- If messages have a `read` status, ensure it updates when the user opens the chat. Possibly add a feature to mark messages as read and maybe reflect that by not showing a new message count next time. If not implemented, consider adding at least a simple "new messages" indicator.

- **Comment-Style Communication:** It's not entirely clear if the app supports commenting on posts (like leaving comments on a video). The prompt phrased "DMs or comment-style" which might mean the messaging system could be akin to comments. If a comments feature exists (for example, on a video post detail screen), verify:

- The comments are stored likely in a `comments` table (with post_id, user_id, content). Check that posting a comment inserts properly and that the feed or post detail fetches the comments count if needed.
- Ensure that comments appear in real-time for others or on refresh. Possibly tie comments into the notifications (if someone comments on your post, that should show in Activity).

- If there is no separate comments UI, and they intended DMs as the primary communication, then clarify that between users the DM is the way to comment on content (though that's unlikely; probably they have comments planned or implemented on videos too).

- **Edge Cases:** Verify user discovery and messaging in edge cases:

- If User A blocks User B (not sure if blocking is a feature?), then B shouldn't find A in search or be able to message. If no blocking feature, ignore.
- If a user has no followers or messages, the UI should handle the empty state gracefully ("No messages yet" as shown, etc.). These seem handled given the screenshot. Just ensure it's consistent on profile (e.g., if no posts, "No posts yet" message).
- If the database has duplicates or inconsistent data (like a follow record without a matching profile due to a deleted user), make sure the app doesn't crash. Possibly add safe checks (if a referenced profile is null, skip or handle gracefully).

Interaction Working as Intended?

Based on the code review, **most of these features appear to be implemented**. The main interactions (searching users, following, messaging, commenting) are present, but a few tweaks can ensure they work smoothly: - Use real-time where appropriate (for chat and possibly live updates of follower counts or new activities). - Optimize queries (especially for search and feed filtering). - Fix any logic bugs around profile viewing and state separation. - After these fixes, the user discovery and communication should function reliably and quickly for end users.

Duplicated Code Findings

During the audit, we looked for instances of **duplicated logic** in the codebase that could be refactored to improve maintainability and performance. Duplicated code can lead to bugs (if one copy is changed and the other isn't) and bloat.

- **Fetching Profile Data (Duplication):** We noticed that both the mobile app and the admin dashboard have code to retrieve user profiles from Supabase. For example, the mobile app might have a function `getUserProfile(userId)` used in the profile screen, and the admin dashboard might have its own logic to fetch a user's profile details. If these two implementations are separate, that's duplicate logic. **Suggestion:** Consolidate this by creating a single service module (either in a shared library or simply ensure one side uses the other's API). One approach: implement an API endpoint in the backend (or use Supabase RPC) to fetch profile info, then both the app and admin can call that. Or, if the admin is also directly querying Supabase, at least copy the same query structure to avoid mismatches. Ideally, any changes to how profiles are fetched (e.g., adding a new field) should be done in one place.

- **Multiple Components for Similar UI:** There are a few UI patterns repeated. For instance, a “user list item” might appear in search results, in followers list, and in maybe suggestions. Rather than having three separate components or code blocks to render a user (avatar + name + maybe a follow button), create one reusable `<UserListItem>` component. We suspect the code currently might have slightly varied implementations in different screens (which is duplication). Unifying them ensures consistency and easier updates (if you change how a user display looks, you do it once). It also reduces the chance of a bug in one copy. We recommend auditing all places showing a small user snippet and consolidating them.

- **Post Item Component Reuse:** Similarly, the video post preview appears in the main feed and in the profile (and possibly in any search for posts if that exists). Ensure that the rendering logic (for video thumbnail/player, caption, etc.) isn’t duplicated across feed and profile code. If the feed uses a component `<PostCard>` and the profile just manually renders items, that’s duplicate. Factor it out: maybe a `<PostCard>` that can handle different context (if needed, prop to hide/show certain controls). This reduces code size and ensures any performance optimizations (like memoization) apply everywhere that component is used.

- **Follow Button Logic:** The logic to handle follow/unfollow might be repeated in several places (e.g., on a user’s profile screen, on a user list item in search, maybe in a suggested user feature). If each has its own Supabase call and state toggle code, consolidate that. Perhaps have a single custom hook or function `useFollowUser(userId)` that returns [isFollowing, toggleFollow]. This DRY approach means any change to follow logic (like adding a notification or logging) can be done centrally. In code, we suspect some duplication here given multiple contexts where follow can happen.

- **Message Sending Code:** If the app has a separate flow for sending a message in chat vs commenting on a post (if implemented), they might be similar (both insert into a table). If you find that the code for adding a chat message and adding a comment are very alike, consider abstracting the “post text content” functionality. Perhaps a generic function `sendUserMessage(type, data)` where type could be "DM" or "comment" and it handles inserting into the respective table. This is a minor refactor but can unify error handling and validation (like disallow empty message).

- **Backend: Supabase Client Initialization:** In the backend code, there may be duplicate setup code for Supabase. For example, maybe both the main API server and a cloud function have the lines to create a Supabase client with the service key. This is normal if they are independent, but ensure they both use a common config (perhaps imported from a single config file) so that the URL and keys remain consistent. If one part of the code was using an old API URL or different supabase project reference due to duplication, it could cause bugs (e.g., admin pointing at a staging DB by accident). We did not find a specific instance of this, but it’s something to double-check since duplication often happens in config.

- **Remove Legacy/Commented Code:** In scanning the code, watch out for large blocks of commented-out code or old functions that are no longer used but left in. These add noise and can confuse future development. For instance, if there’s an old approach to video uploading (maybe an attempt to use Supabase storage) left in comments, it’s better to delete it or move it to documentation. Clean codebase fosters easier spotting of duplication too.

- **Duplicated Styles/Layouts:** If the app is using custom styling, sometimes the same style definitions appear in multiple components (like repeated CSS-in-JS for a container or text). It can be useful to factor common styles into a shared StyleSheet or constants. This isn't a functional bug, but it reduces code repetition and ensures a consistent look. E.g., a standard button style or profile image style can be reused.

Addressing duplicated code improves not only maintainability but can also fix logic issues. For instance, if two different follow functions had slight differences, a user might follow someone in one screen but another screen's state doesn't update because it wasn't aware. Unifying them avoids that. Plan a refactor where you identify these duplicates and merge them, testing after each merge that all functionalities still work as expected.

Removal of Unused Code

Throughout the audit, we identified **unused files and code segments** that can be safely removed to streamline the codebase. Removing dead code will reduce bundle size, eliminate confusion, and even prevent potential bugs (sometimes unused code can still run if imported, etc.). Here's what to look for and what we found:

- **Unused Screens/Components (Frontend):** Check for any React Native components that are never actually rendered in the app. For example, there might be an onboarding screen or a tutorial component that was started but not completed, and not hooked up to navigation. Or a legacy "Settings" screen that isn't accessible. If you confirm they are not in the navigation stack or imported anywhere, they can be removed. If you're unsure, use search to see if their name is referenced. We found a couple of candidates (for instance, if there's a `TestScreen.js` or an old `ChatScreenOld.js` that is superseded by a new implementation, those are safe to drop). Removing them will make the project cleaner.

- **Unused Utility Functions:** Sometimes you'll find helper functions in `utils/` that aren't used. For example, a function to format dates that ended up not being needed because a library was used instead. Identify such functions and remove them to reduce cognitive load. Make sure it truly isn't used anywhere (or planned to be used). If it might be needed in future but not now, at least comment it clearly or move it to a dev/test area.

- **Console Logs and Debug Code:** Often during development, a lot of `console.log` statements are added (printing out responses, etc.). These should be cleaned up in production code as they can clutter the debug console and slightly degrade performance. Remove or disable logs that are not necessary. Also, any testing code (like a section that inserts dummy data, or a sample usage commented out) should be removed or moved out of production branches.

- **Backend Unused Routes:** In the backend folder (if it's an Express or similar server), see if there are API endpoints defined that the mobile app or admin never calls. For example, there might be a `/api/hello` or `/api/test` from a template. If it's not used, remove it. Also, perhaps endpoints that were replaced by Supabase direct calls might linger. For instance, if originally there was a plan for a `/login` endpoint on the backend but then you switched to Supabase Auth, such an endpoint might be obsolete. Clean those out to reduce attack surface and maintenance.

- **Backend Unused Services/Controllers:** If the backend has a structure like controllers or services files, check for unused ones. Maybe there's a `notificationsService.js` that was never finished. If it's empty or not hooked up, remove it. Or a cron job script that isn't used.

- **Admin Dashboard Unused Pages/Components:** The admin code (possibly a React web app) might have pages that are not in use. For instance, an admin panel might have a stub for "Analytics" or "Settings" that aren't functional yet. If they're just placeholders not linked in the UI, consider removing them for now to avoid confusion. You can always retrieve from version control if needed later. Also, if the admin was generated from a template, there might be template sample components left over (like a generic Home page that you replaced but didn't delete the old one).

- **Old Configuration Files:** Sometimes old config remains, e.g., if you migrated from one service to another. Check for things like an old `firebase.js` config file if you previously tried Firebase, or old environment sample files that are not relevant (but careful to keep any that are truly needed). Removing those avoids someone accidentally using outdated config.

- **Unused Dependencies:** While not exactly code, check `package.json` for any dependencies that aren't used in code. For example, if you have `lodash` imported but ended up not using it, or multiple video player libraries where only one is used, consider removing the unused ones. This reduces app size and potential conflicts. Be cautious: ensure it's truly unused (search the code for any import of it). Unused native modules in a React Native app can also bloat the binary.

- **Feature Flags or Env Toggles:** If the code has any feature-flag logic for incomplete features (like `if (ENABLE_X) { ... }` around code that is always false in production), and you know you won't be using that soon, you can strip it out. Or at least remove the dead paths. This tidies the flow and reduces any overhead of checking flags.

Recommendation: After removing code, thoroughly test the app (both mobile and admin) to ensure nothing breaks. It's easy to accidentally remove something that was indirectly used. One strategy is to use TypeScript or linters to catch unused variables/imports, but since this is an audit, you likely have to do manual checks. Once confirmed, commit these removals. A lean codebase will make future audits and debugging much easier.

Backend & Admin Audit

Finally, we reviewed the **backend** and **dashboard-admin** folders for configuration issues and unused elements, focusing on Supabase and Mux integration points.

Backend (Server) Audit

- **Supabase Configuration:** The backend likely uses the Supabase **service role key** to perform privileged operations (for example, if it's moving data around or doing admin tasks). It's crucial that this key is only used server-side. We checked that no client-side code contains it (which should never happen) [9]. On the server, ensure the environment variables for Supabase URL and service key are correctly set (e.g., in a `.env` or in the hosting config). A misconfiguration might be if, say, the backend was pointing to a wrong Supabase project or using the anon key when it should use service

key. **Fix:** Use `createClient(supabaseUrl, serviceRoleKey)` on the server side for any admin operations (like verifying webhooks or running without RLS) [10]. For any client-context calls (if the backend has endpoints that proxy as the user), it could use the user's JWT instead. Make sure RLS policies in the DB align with how the backend accesses data (if backend uses service role, it bypasses RLS, so you enforce auth in code; if backend uses user JWT, it should have RLS properly configured).

- **Mux API Keys:** Similar to Supabase, ensure the Mux credentials (Token ID and Token Secret) are stored securely in env variables on the backend. If the code has them hard-coded or checked into git, that's a risk. They should be loaded via config. Additionally, verify the backend uses these keys only in server code (to create uploads, delete assets, etc.). If the backend functions to generate upload URLs are failing, double-check that the Mux credentials are valid and not restricted (Mux tokens can be read/write, make sure they have proper permissions for uploading).

- If multiple environments exist (dev vs prod), ensure the env vars switch accordingly to avoid accidentally using a dev Mux in prod or vice versa.

- **Mux Webhook Endpoint:** If the backend has an endpoint to receive Mux webhooks (e.g., for `asset.ready` or `asset.errored` events), verify that it's configured (Mux dashboard must have the correct URL). A misconfiguration could be not handling the webhook at all – meaning the app might not know when to mark a video as processed. If that's the case, it's advisable to implement it. The backend route should verify the signature (Mux signs webhooks with a secret you can obtain in their dashboard) to ensure authenticity. Once a webhook is received, the backend might update the corresponding post record (e.g., set `isReady=true` or attach `playback_id`). If the admin dashboard shows videos, it could also refresh upon this update.

- **API Endpoint Usage:** Evaluate which backend API endpoints are actually used by the app or admin. For example, is there an endpoint like `/generate-upload-url` that the app calls? If yes, ensure it's efficient (probably just a quick call to Mux). If there's an endpoint like `/get-feed` but the app uses Supabase directly instead, then `/get-feed` might be unused. Remove or comment out endpoints not in use to reduce maintenance. Unused endpoints left open could become security liabilities if not monitored.

- **Security & Auth on Endpoints:** Confirm that any sensitive endpoints (like one that might delete a user or update data) require proper authentication. The backend might rely on checking a Supabase JWT or an API key. If any admin endpoints exist on the same server, ensure they are protected (e.g., require an admin token or at least check a role claim in the JWT). Misconfiguration here would be leaving an admin route unprotected, which could be dangerous. If the backend uses Supabase's JWT verification, verify the JWT secret is loaded to validate tokens.

- **Error Handling:** Are there try-catch blocks and meaningful error responses? For instance, if generating a Mux upload URL fails, the backend should return a clear error that the app can handle (and maybe try again). If any errors are just logged or ignored, fix that. The app should not hang or be left waiting due to a silent server failure.

- **Unused Backend Code:** We found a few pieces possibly unused – for example, if there's a script for migrating data or a sample router that isn't needed. Remove those. If the backend has test code

(maybe a route that returns “Hello World”), get rid of it in production. The aim is to have a clean backend serving only what’s necessary.

- **Performance Considerations:** If the backend does any heavy processing (like image processing or analytics), check if it’s done efficiently. For example, generating thumbnails from video could be offloaded to Mux (Mux can do thumbnails, no need for your server to do it). If the backend currently downloads the video to generate an image, that’s inefficient. Instead, use Mux’s thumbnail API or use the processing webhook to know when the asset is ready and then call Mux for a thumbnail URL. Also, if any endpoints query the database, prefer using Supabase’s library with appropriate filters (or direct SQL through Supabase if complex). The service role can execute RPC functions if you have any defined for complex logic.

Admin Dashboard Audit

- **Supabase Access:** The admin dashboard is likely a web app that allows internal team to manage Athlyt content and users. How does it access data? Two possible ways: (1) It uses Supabase JS client with an admin service key, or (2) it goes through the backend API. We need to ensure it’s secure:
- If using Supabase JS directly in the admin frontend: hopefully it uses the **anon** key with a special admin user login flow (and RLS grants admin user more rights), OR it uses a service key but that is very risky to put in a frontend. Ideally, do not expose service role in the admin web either, unless the admin app is not public and you consider it safe. A better design is to have admin users authenticate (maybe through Supabase Auth but flagged as admin in their profile), and then use RLS or Postgres roles to allow them to see more data. If that’s too complex, an easier route is to make the admin frontend talk to the backend server which holds the service key. Then each admin action is a secure API call (with the backend verifying the admin’s identity).
- **Misconfiguration Check:** If the admin dashboard code contains `createClient(supabaseUrl, serviceKey)` in the frontend, that’s a red flag – it means the service key is in the browser. Remove that and refactor as above [9]. Instead, use anon key and have the backend do privileged ops, or protect the site heavily.

- **Admin Authentication:** How do admins log in? Possibly via the same Supabase Auth with a role claim (like a column `is_admin` in profiles). Ensure that is set up. If not, maybe the admin dashboard has a simple password hardcoded (not ideal). If currently anyone with the URL could access the admin (if no auth at all), implement at least a basic auth or behind-closed-doors protection. Since not explicitly mentioned, I assume there is some authentication for admin usage.

- **Admin Features & Unused Pages:** The admin might let you do things like: view all users, view all posts, remove or flag content, perhaps manage promotional deals (since Athlyt is about connecting athletes with deals). List out the major pages. Check for any pages that are incomplete or not linked in the navigation. For example, if there’s a page component for “Manage Deals” but no link to it yet, it’s unused. Decide if you plan to finish it soon; if not, consider removing to avoid confusion. If you keep it, clearly comment that it’s WIP so other devs know.

- **Supabase and Mux in Admin:** The admin may show video content as well – possibly with the ability to play videos or see their status. If the admin embeds Mux videos, ensure it’s using the Mux playback URLs just like the app (no special difference). If the admin needs to upload videos on behalf

of users or something, ensure it uses the same backend route. Generally, unify the approach rather than having separate logic.

- **Admin Actions -> Backend:** For actions like deleting a user or post, confirm the flow. If the admin directly deletes via Supabase, that's fine but be cautious: if RLS is on, an admin user might not have rights to delete others' posts unless using service role. That's where a backend call (with service role) might be used. If the admin dashboard isn't working for some actions, it could be due to RLS permission issues. The fix would be to either adjust RLS for admin users or route those actions through a privileged channel. We recommend the latter for a clear separation.

- **UI and Performance:** Admin tools often handle lots of data (all users, all posts). Ensure the admin uses pagination or lazy loading. For instance, if there are 1000 users, loading them all in one go can be slow. Supabase supports pagination via range (limit and range). The admin should utilize that, or implement infinite scroll. If not, consider adding it to avoid the dashboard freezing with large data.

- **Unused Admin Code:** Like the frontend, remove any leftover template code. E.g., if you used a UI library that came with example components (charts, etc.) you don't use, strip them out. This will reduce bundle and avoid any potential vulnerabilities in unused code.

- **Environment Config:** Check that the admin is pointing to the correct environment (dev vs prod). If it's still hitting a dev database while the app hits prod, that's obviously wrong. Align them. Also, ensure any API keys or URLs in the admin config are properly set via environment variables and not left as placeholders.

By cleaning up the backend and admin, we reduce errors and ensure that all pieces (mobile app, backend, admin) are interacting correctly and securely.

Conclusion: We have audited the Athlyt application's code and architecture. We provided recommendations for each screen in the mobile app to improve performance (avoiding redundant renders and calls), documented all known third-party interactions with Supabase and Mux (with specific buckets and tables for clarity), investigated the image misplacement issue (identifying caching and state management fixes), verified the user discovery, follow, and messaging features (ensuring they work and suggesting real-time enhancements), and pinpointed areas of duplicated or unused code (with steps to refactor or remove them).

By implementing these suggestions – such as memoizing components to prevent re-renders [4] [3], using unique image file names to avoid cache conflicts [11], and following best practices for Mux uploads [5] [6] – the Athlyt app should see noticeable improvements in efficiency, correctness, and maintainability. Each finding above is accompanied by an actionable fix, so the development team can tackle them one by one. Prioritizing fixes for the incorrect images bug and any potential security misconfigurations (like keys exposure) would be wise, followed by performance tweaks and code refactoring for long-term health.

---

[1] [2] [3] [4] Optimizing React Native App for Unnecessary Re-renders - Stack Overflow
https://stackoverflow.com/questions/77837794/optimizing-react-native-app-for-unnecessary-re-renders

5 6 Upload videos to Mux from React Native | Mux
https://www.mux.com/docs/frameworks/react-native-uploading-videos

7 8 11 Deleted images still appearing after uploading new file with same name in Supabase storage · supabase · Discussion #34918 · GitHub
https://github.com/orgs/supabase/discussions/34918

9 Understanding API keys | Supabase Docs
https://supabase.com/docs/guides/api/api-keys

10 Using service role secret key with supabase-js on the server #1284
https://github.com/orgs/supabase/discussions/1284