ChatGPT

Omnisonic Acceleration Roadmap and Repo Documentation

docs/REPO_MAP.md

Monorepo Layout and Components

The Omnisonic repository is organized as a monorepo with separate folders for applications, services, shared packages, infrastructure, and documentation [1]:

- `apps/studio-web` – Next.js 13 "Studio" web application (Phase 1 UI) for real-time creation & collaboration [2]. This includes pages for sessions, uploads, etc., using React 18, Tailwind (shadcn/UI), and NextAuth for auth.
- `apps/insight-web` – Next.js "Insight" web app (Phase 3 analytics dashboard) for trends and news analytics, utilizing ClickHouse via server actions [3] [4].
- `services/graph-api` – GraphQL Yoga service (Phase 2 Core) exposing the Open Music Graph (models like Work, Recording, License, etc.) on HTTP port 4000 (with websockets on 4001) [5] [6].
- `services/realtime-gateway` – Node WebSocket server for real-time presence and messaging within studio sessions (Phase 1). Runs on port 8080 and manages room membership, using Redis for presence state [7] [8].
- `services/studio-api` – FastAPI (Python) service stub that mirrors some Next.js API functionality (for parity/testing), e.g. basic session CRUD (Phase 1) [9] [10].
- `services/export-worker` – Node worker service using BullMQ + FFmpeg for background audio mixdown exports. Takes export jobs (queued in Redis) and produces rendered audio files (WAV/MP3) stored via the storage module [11] [12].
- `services/upload-cleaner` – Node cron-style worker that periodically deletes orphaned uploads (files not attached to any session) older than 30 days [13] [14].
- `services/alerts` – Node Fastify service for alerts/notifications (Phase 3). It polls analytics (ClickHouse) and triggers notifications according to AlertRules, via email, webhooks, or Slack [15] [16].
- `services/ingest` – FastAPI service for ingesting external data. It provides endpoints like `/ingest/isrc` for batch track metadata import (to Graph API) and `/ingest/rss` for ingesting news feeds into ClickHouse (Phase 3) [17]. It also performs entity tagging using heuristics/embeddings, with optional DuckDB for backup storage [18].
- `packages/db` – Shared database module. Contains the Prisma schema and client for PostgreSQL (all services use this). The Prisma schema (`packages/db/prisma/schema.prisma`) defines models for **User**, **StudioSession**, **Work**, **Recording**, **Upload**, **Export**, **License**, etc. [19] [20]. It also includes NextAuth models (Account, Session, VerificationToken) integrated into the same schema [21] [22].
- `packages/storage` – Shared storage abstraction for file uploads. Provides utilities to generate storage keys, put objects in storage (local filesystem, MinIO, or S3), generate signed download URLs, and delete objects [23] [24]. This is used by the Studio app for user uploads and by the export worker for output files.

- `packages/telemetry` – Shared OpenTelemetry tracing setup. Exports helpers to instrument requests (e.g. `withSpan`) so that API routes and services emit trace logs to stdout (no external APM needed) [25] [26] .
- `packages/schemas` – Shared schemas/validation (e.g. Zod schemas for ISRC, ISWC identifiers) used by Graph API to validate inputs [27] .
- `infra/docker` – Docker Compose files and environment for local development (services such as Postgres, Redis, MinIO, ClickHouse) [28] [29] . For example, `docker-compose.dev.yml` defines local services for testing all features.
- `docs` – Documentation, specs, and ADRs (Architecture Decision Records). Notable docs include the Phase 1 Studio spec [30] , Phase 3 Insight spec [31] , and ADRs like monorepo structure and database choice.
- `prompts` – Contains prompt files used during development (e.g. `prompts/core.md`, `prompts/studio.md`), which are not part of the running system but document how features were incrementally added.

Each app/service is published as a workspace package (via PNPM workspaces) with its own dependencies and scripts. For instance, the `@omnisonic/studio-web` package uses Next.js and has scripts for dev/build/lint, whereas `@omnisonic/graph-api` uses a Node server script. All share common dev dependencies via the root config.

Running the Full Stack Locally

Prerequisites: Ensure Node 20+, PNPM, and Python (for FastAPI services) are installed. Also run supporting services (PostgreSQL, Redis, etc.) – the easiest way is using Docker Compose. From the repo root, you can spin up the database and others with:

```
docker compose -f infra/docker/docker-compose.dev.yml up -d postgres redis
minio clickhouse
```

This starts Postgres (on port 5432), Redis (6379), MinIO (9000 with console on 9001), and ClickHouse (8123 for HTTP) for local development. Create a bucket in MinIO named `omnisonic-dev` (matching `MINIO_BUCKET_NAME`) for uploads [13] .

Environment Setup: Copy or create `.env.local` files for the apps/services with required variables. Key env vars include:

- `DATABASE_URL` – Postgres connection string (e.g. `postgres://postgres:postgres@localhost:5432/postgres`) [32] .
- `REDIS_URL` – Redis URL for realtime presence, queues, etc. (e.g. `redis://localhost:6379`) [33] .
- **Auth:** `NEXTAUTH_URL` (typically `http://localhost:3000` for dev), `NEXTAUTH_SECRET` (random string for session signing), and OAuth client IDs/secrets if using Google/GitHub logins [34] . In dev, you can set `AUTH_ENABLE_CREDENTIALS=true` to allow email/password login for convenience [35] .
- **Storage:** `STORAGE_TYPE` (`local` for filesystem, or `minio`/`s3`), and corresponding credentials: `MINIO_ENDPOINT`, `MINIO_ACCESS_KEY`, `MINIO_SECRET_KEY` (for MinIO) or

`AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY` (for AWS S3) [29] [36]. Also set `MINIO_BUCKET_NAME` / `S3_BUCKET_NAME` (e.g. `omnisonic-dev`). Optionally, `STORAGE_CDN_URL` if serving files via a CDN or public proxy, and `STORAGE_LOCAL_DIR` if using local disk storage path [37] [38].
- **Other:** For ClickHouse analytics, `CLICKHOUSE_HOST` (default `http://localhost:8123`), and for LiveKit (if running in milestone 2), `LIVEKIT_API_KEY` and `LIVEKIT_API_SECRET` plus any LiveKit URL if not using default (`ws://localhost:7880` in dev), etc.

Each service can be started in its own terminal. A typical local run sequence:

1. **Install dependencies and build:**

```
pnpm install
pnpm db:generate   # Generate Prisma client
pnpm db:migrate    # Apply database migrations to Postgres
```

2. **Start the core services:**

3. GraphQL API (Core): `pnpm dev --filter @omnisonic/graph-api` (by default serves on http://localhost:4000) [6].
4. Realtime Gateway: `pnpm dev --filter @omnisonic/realtime-gateway` (WebSocket server on ws://localhost:8080) [2].
5. Export Worker: `pnpm dev --filter @omnisonic/export-worker` (starts the BullMQ worker, which connects to Redis and awaits jobs) [14].
6. (Optional) FastAPI Studio API:

```
cd services/studio-api
uvicorn main:app --reload --port 8000
```

This runs the Python API stub on http://localhost:8000 [10]. (The Next.js app does not heavily depend on this stub, but it mirrors certain endpoints.)
7. (Optional) Ingest Service:

```
cd services/ingest
uvicorn ingest.main:app --reload --port 8100
```

Used for ISRC or RSS ingestion if testing those Phase 3 features [39].

8. (Optional) Alerts Service: `pnpm dev --filter @omnisonic/alerts` (runs on default port 8200 if used) [16].

9. **Start the Studio web app:**
In another shell, run:

```
pnpm dev --filter @omnisonic/studio-web
```

This starts the Next.js development server on http://localhost:3000 [40] . The Studio app will connect to the other services (Graph API, realtime, etc.) via environment-configured URLs or defaults. For example, it calls GraphQL queries to the Graph API (assuming it's on localhost:4000) and opens a WebSocket to the realtime gateway (ws://localhost:8080).

10. **(Optional) Start the Insight web app:**
    If testing analytics (Phase 3), run:

```
pnpm dev --filter @omnisonic/insight-web
```

This runs another Next.js app on a different port (default 3001). Ensure ClickHouse is up (as per docker-compose) before using Insight.

Once the above are running, you can open the Studio app at `http://localhost:3000`. After signing in (the app will redirect to `/signin` if not authenticated), you should see the sessions dashboard. You can create a session, upload audio files, render an export, etc., verifying that each service works together.

Initial Data: The system uses an empty database by default. Upon first sign-in via OAuth or credentials, a new user record is created in Postgres (NextAuth handles this). No other seed data is required for core features, but for certain Phase 2/3 features (Work/Recording, news items) you might need to run ingest processes to populate data. Those can be done later as needed. The repository includes a Playwright end-to-end test that can serve as a smoke test for basic functionality (it creates a session, uploads a sample file, and triggers an export) [41] .

Authentication Model and User Identity

Omnisonic uses **NextAuth.js** for authentication in the Studio web app. The authentication model is centered on the **User** model in Postgres (via Prisma) and supports both OAuth providers and credentials:

- **User Accounts:** The `User` model (defined in Prisma) includes fields for name, email (unique), image, and a hashed password (nullable) [42] . NextAuth creates a User entry on first login if none exists for the email.
- **Providers:** GitHub OAuth is required for production (as per spec), with Google and Discord optionally configured [43] [44] . In development, a **Credentials** provider is enabled (unless `AUTH_ENABLE_CREDENTIALS=false` is set) to allow email/password login for testing [45] . The credentials flow will automatically create a new user with a hashed password if an email is not found (in non-production mode) [46] , or authenticate against the stored hash if the user exists [47] .
- **NextAuth Sessions:** NextAuth is configured to use the **PrismaAdapter**, storing sessions and accounts in the database [48] . This means persistent sessions are saved in the `Session` table (with session token, userId, and expiration) [22] . By default, session cookies are also used by NextAuth. The session strategy is set to `"database"` in this app [49] , so the client gets a session cookie and server verifies it against the DB.

- **Auth Routes:** The Next.js app uses the built-in NextAuth API routes under `/api/auth/[...nextauth]` (no custom code needed aside from the configuration in `lib/auth.ts`). The sign-in page is overridden to a custom React page at `/signin` [50].
- **Authorization & Identity Propagation:** Once logged in, the NextAuth session includes the user's ID (we see the callback ensuring `session.user.id = user.id`) [51]. This ID is used throughout the app to scope data. For instance, API routes like upload and sessions ensure the current session's userId matches the resource's owner before allowing actions [52] [53]. There is no complex RBAC implemented yet – effectively all authenticated users are standard users, and any "admin" concept would be an extension in future.
- **User Identity Source:** Ultimately, the source of truth for user identity is the `user` **table in Postgres**, managed via Prisma and NextAuth. All services that need to associate data with a user (uploads, sessions, alerts, etc.) use the user's UUID from this table as a foreign key. For example, the `StudioSession.userId` field links a session to its owner [54], and each `Upload.userId` links an uploaded file to the uploading user [20].
- **Session Protection:** The Studio web app uses NextAuth's middleware to protect pages that require login. According to Phase 1 spec, the sessions pages and APIs are behind auth checks [55]. Indeed, the API route implementations call `auth()` (NextAuth getter) and return 401 if no session or no user id is present [56] [57]. This ensures non-logged-in users cannot access those resources.

In summary, authentication is handled in the front-end via NextAuth (with multiple providers and DB-backed sessions), and the identity is persisted in the Omnisonic Postgres DB. All subsequent requests (GraphQL or REST) trust the session and use the user's ID from it for authorization checks.

File Storage & Upload Pipeline

Omnisonic supports user file uploads (audio, images, etc.) through a unified storage interface. The design is **library-first and local-first**: files are stored either on the local filesystem or an S3-compatible service (like MinIO or AWS S3), and never leave the user's control (no external transcoding service). The **upload pipeline** involves the Studio web app (Next.js API route), the storage package, and the database:

- **Uploading Files (Studio Web API):** The Next.js app has a REST endpoint `/api/upload` that handles file uploads via `POST` form-data [58]. When a user selects a file (or multiple files) in the UI, the front-end calls this endpoint (see `UploadPanel` component using `fetch("/api/upload", ...)` with the file data [59]). The handler does the following:
- Authenticates the request (`requireUserId()` ensures the user is logged in) [56] [60].
- Parses the multipart form and gets the file(s) and optional `sessionId` (if the file is being attached to a studio session) [61].
- Validates the file (size limit 100MB, and allowed MIME types: audio/*, image/*, video/*) [62].
- If a `sessionId` is provided, it verifies that session exists and belongs to the user (preventing uploading to someone else's session) [63].
- **Stores the file:** It reads the file into a Buffer and generates a unique storage key via `generateStorageKey(userId, file.name)` [64]. The storage key format is `uploads/<userId>/<timestamp>-<filename>` ensuring user scoping and uniqueness [65]. Then it calls `putObject({ key, contentType, body })` to save the file via the storage module [64]. Depending on `STORAGE_TYPE`, this either writes to disk (`.uploads/` dir) or to the configured S3/MinIO bucket [23] [66]. The `putObject` function returns a URL or URI for the stored file:
    - For local, it returns a URI like `local://<key>` (or a CDN URL if one is configured) [38].

    - For S3/MinIO, it returns either a CDN URL (if configured) or the direct bucket URL (e.g. `https://<bucket>.s3.<region>.amazonaws.com/<key>`) [24].
- Creates a database record in the `Upload` table via Prisma, with fields: `userId`, `sessionId` (if any), original file name, size, MIME type, the `storageKey` generated, and the `storageUrl` returned [67]. This record's ID is used to reference the upload later.
- Computes a download URL for immediate use. The API tries `getDownloadUrl(key)` which will return:
    - A public URL if available (e.g. a signed URL for S3, or the CDN link) [68] [69].
    - If it returns `null` (which is the case for local files with no CDN), the API falls back to constructing an internal URL: `<origin>/api/upload/<uploadId>/file` [70]. This is a route to stream the file through Next.js if direct file access is needed.

- Responds with JSON containing the new `upload` record and the `downloadUrl` [71].

- **Storage Abstraction:** The logic for saving and retrieving files is in the storage package. It automatically handles different backends:

- **Local:** Files are saved under `STORAGE_LOCAL_DIR` (default `./.uploads` in project root) with directory structure mirroring the key (it creates subfolders as needed) [72]. Reading the file later is done via `readLocalFile(key)` which just reads from disk [73].
- **MinIO / S3:** The app uses AWS SDK. A single S3 client is initialized for both S3 and MinIO (with `forcePathStyle: true` for MinIO) using env credentials [74]. `putObject` streams the buffer to the bucket [24]. `getDownloadUrl` will generate a signed URL (valid for 1 hour by default) via AWS SDK if no CDN is set [75].

- **CDN Option:** If `STORAGE_CDN_URL` is configured (say you front MinIO with an Nginx or CloudFront), the storage module will simply return and compose URLs based on that (for both put and get), avoiding any need for signed URLs [76] [77].

- **Downloading Files:** For files stored on S3, the client gets a signed URL (or CDN link) and can download directly. For local files, since the returned `downloadUrl` is an API route (e.g. `/api/upload/<id>/file`), the Next.js app has a handler for that. The route `/api/upload/[id]/file` checks auth and ownership similar to upload:

- It ensures the requesting user owns the upload record [78].
- It only handles local storage case – if the storage type is not local, it returns an error (because for S3 we expect the signed URL to be used directly) [79].

- It reads the file from disk (`readLocalFile(storageKey)`) and returns the raw file bytes with appropriate `Content-Type` and `Content-Disposition` headers [80]. This allows the browser to download the file or play it.
  In practice, the Studio UI provides a download link (opening in a new tab) for each upload, which hits this route for local files [81].

- **Lifecycle and Cleanup:** Each upload is tied either to a session or (in the future) to a user's library. The `Upload` records have a foreign key `sessionId` which can be null for library imports.

Unattached uploads might be cleaned by the `upload-cleaner` service to reclaim space [82]. When a user deletes an upload via the UI, the API `/api/upload/[id]` (DELETE) is called:

- It validates the user owns the upload, then calls `deleteObject(storageKey)` to remove it from storage (deleting the file) [83] and then removes the DB record [84].
- Telemetry logs any failure to delete the file but proceeds to delete the DB entry regardless, to avoid ghost records [85].

- **Data Flow Summary:** From upload to playback:

- **User action (UI):** User selects a file and initiates upload.
- **Next.js API:** The file is received and stored (local disk or MinIO/S3) and an `Upload` DB record is created [67].
- **URL Returned:** The client gets a download URL (either directly to storage or an API link) [71].
- **UI uses URL:** The Studio UI can immediately use this `downloadUrl` to allow the user to download or play the file (for audio, an HTML `<audio>` tag could use the URL to stream playback).
- **Playback:** The app currently provides a simple download link. Playback within the browser can be achieved by pointing an audio element to the same URL. (A future library player will likely use the stored URL to stream audio).

Data Flow Diagram: Below is a high-level sequence of how an upload moves through the system:

```
sequenceDiagram;
  participant User;
  participant StudioAPI as Studio Web API (/api/upload);
  participant Storage as Storage (Local/MinIO/S3);
  participant DB as Postgres (Uploads table);
  User->>StudioAPI: POST /api/upload (file, sessionId);
  StudioAPI->>StudioAPI: Authenticate user;
  StudioAPI->>Storage: putObject(key, file buffer) [64] [24] ;
  StudioAPI->>DB: prisma.upload.create(...) [86] ;
  Storage-->>StudioAPI: stored file URL [87] ;
  StudioAPI->>User: JSON { upload record, downloadUrl } [71] ;
  User->>StudioAPI: GET /api/upload/list?sessionId=...;
  StudioAPI->>DB: fetch Uploads for user/session [88] ;
  StudioAPI->>Storage: getDownloadUrl(key) for each [89] ;
  Storage-->>StudioAPI: signed URL or null;
  StudioAPI->>User: JSON { uploads: [ {upload, downloadUrl}, ... ] } [90] ;
  User->>Browser: Clicks "Download" (opens downloadUrl) [81] ;
  opt If local storage (downloadUrl is /api/upload/:id/file):
    User->>StudioAPI: GET /api/upload/123/file;
    StudioAPI->>DB: find upload 123, verify owner [91] ;
    StudioAPI->>Storage: readLocalFile(key) [80] ;
    StudioAPI-->>User: file bytes (Content-Type: audio/*) [92] ;
  end;
  User-->>Browser: Audio file plays or saves locally;
```

- **Relation to Tracks:** As of now, uploads are primarily associated with sessions (stems or assets uploaded in a recording session). There isn't yet a separate concept of a "Library Track" entity in the schema – this will be introduced in Milestone 1. Currently, if a user wants to import a song into the system, it would still create an `Upload` entry (with no sessionId) and nothing in the UI references it. The upcoming roadmap will add a first-class **Track** model and library UI to manage such uploads outside of sessions.

Database Schema and Notable Models

Omnisonic uses **PostgreSQL 16** with Prisma as the ORM/ODM. The Prisma schema is located at `packages/db/prisma/schema.prisma` [1]. Key models relevant to the music library and collaboration are:

- **User** – Represents a user account (as described above in auth). Each user can have many StudioSessions, Uploads, Exports, etc. [19].
- **StudioSession** – A collaborative session (Phase 1) where a user can record or mix. Fields: `id`, `name`, `userId` (owner), timestamps [54]. Relations: one **User** (owner) and many **Upload**s and **Export**s. This is essentially a "project" or "song session".
- **Upload** – An uploaded file (audio asset or otherwise). Fields: `id`, `userId` (uploader), `sessionId` (nullable), `fileName`, `fileSize`, `mimeType`, `storageKey`, `storageUrl`, `createdAt` [20]. An Upload may belong to a StudioSession (e.g. a guitar stem in a session) or have no session (intended for the general music library). In the current state, uploads without a session are not surfaced anywhere – this is a gap to address for the Library feature. **Note:** There is **no explicit "Track" model yet** for library songs – we will likely extend or mirror Upload for that purpose.
- **Export** – An exported mixdown of a session. Fields: `id`, `sessionId`, `userId`, `status` (pending/processing/completed/failed) [93], `format` (wav/mp3/flac), `fileUrl`, `storageKey`, `fileSize`, `progress`, `errorMessage`, and timestamps [94]. An Export is produced by the export-worker. When complete, it has a `fileUrl` (which is similar to storageUrl but specifically for the final mix file) and can be downloaded via the Insight UI or Studio UI (the Studio session page has a section listing exports).
- **Work** – A musical work (song composition). Fields: `id`, `title`, `description`, `iswc` (unique code for works), `genres` (string array), timestamps [95]. Relations: many Recordings, Contributions, Licenses, and RoyaltyEvents. This is part of the Phase 2 "Open Music Graph" – it represents the composition (the underlying song).
- **Recording** – A recording of a Work (typically a released track). Fields: `id`, `title`, `workId` (the Work it is a recording of), `isrc` (unique code for recordings), `durationSeconds`, `primaryArtist`, `releasedAt`, timestamps [96]. Relations: belongs to a Work, has many RoyaltyEvents. This is the entity closest to a "Track" in a music library sense, but currently it's used for industry metadata rather than personal uploads. (Recordings come into play via the ingest service or manual entry.)
- **Contributor** and **Contribution** – These model relationships between people (contributors) and works (songwriters, etc.) along with their share (pctShare) [97] [98]. They are relevant for the royalty ledger but not needed for basic library functionality.

- **RoyaltyEvent** and **LedgerEntry** – These support the Royalty Integrity Network (Phase 2). A RoyaltyEvent might represent a usage of a Recording (like a stream or play), and LedgerEntry represents an entry in the ledger for payouts to contributors [99] [100]. These tie into License and CycleCheckpoint for periodic settlements [101] [102].
- **License** – Represents a license of a work (mechanical, performance, etc.). Fields: `id`, `workId`, `licensee`, `territory`, `rightsType` (enum), dates, terms (JSON), status (draft/active/etc.) [103] [104]. Part of Phase 2 rights management.
- **EntityTag** – A tag linking a news item to an entity (artist/work/recording) for the Insight features. Fields: `id`, `newsItemId`, `entityType` (enum: artist/work/recording), `entityId`, `confidence`, `method` (heuristic/fuzzy/embedding), `matchedText` [105] [106]. These are filled by the ingest tagging process and can be queried via GraphQL for analytics.
- **AlertRule, AlertChannel, AlertEvent** – Models for the Alerts system (Phase 3). They allow configuration of alerting conditions and logging of alert events (e.g., if a certain artist is mentioned 10 times in an hour, send a Slack notification). The details include channel types (email/webhook/slack) and thresholds [107] [108].

Track/Library Gap: Notably **missing** in the current schema are models for personal music library management. There is no `Track` or `Playlist` model yet – the expectation is to introduce a Track (or reuse Recording) to represent imported songs, and a Playlist (or Collection) for user-curated lists. The current closest proxies are: - `Recording` – which has some fields a track would need (title, artist, duration). However, Recording is tied to Work and assumes industry context (ISRC, primaryArtist). - `Upload` – which has the file info and owner, but lacks metadata like title/artist beyond the filename. - There is no Playlist or Tag model for user libraries (the only tags are the EntityTag for news/AI, not for tagging tracks with genres etc.).

We will address these gaps by extending the schema in the roadmap (Milestone 1 will add a **Track** model, possibly a join table for Playlist tracks, and tags for tracks).

The Prisma setup uses migrations to evolve the schema. Running `pnpm db:migrate` will apply any pending migrations. The dev and test environment use the same Postgres (with a separate schema or database for tests as configured in the environment). The codebase likely includes migration files under `packages/db/prisma/migrations/` (not individually cited here, but present in the repo).

API Surface Overview

Omnisonic exposes both GraphQL and RESTful APIs:

- **GraphQL API (Core):** The GraphQL service (`services/graph-api`) provides the **Open Music Graph** – a set of types and operations largely focused on works, recordings, licensing, and royalty ledger. It uses GraphQL Yoga. The schema is defined programmatically in `src/index.ts` using `createSchema` with type definitions and resolvers in-line [109] [110]. Notable aspects:
- **Types:** Work, Recording, Contributor, Split, License, LedgerEntry, CycleCheckpoint, EntityTag, etc., each corresponding to a Prisma model [111] [112]. There is also a custom scalar for JSON and some enums (LicenseStatus, LicenseRightsType, TaggedEntityType, TaggingMethod) [113] [114].
- **Queries:** Current Query fields include: `work(id)`, `recording(id)`, `cycleCheckpoint(id)`, `cycleCheckpoints(limit, offset)`, `license(id)`, `licenses(workId)`, `activeLicenses(workId, territory, rightsType)`, and `entityTags(newsItemId)` [115]

[116]. These allow fetching works and recordings by ID, listing licenses, etc. For example, the resolver for `Query.work` finds a Work by ID including its related recordings, contributions, licenses, etc. [117]. **There is currently no GraphQL query for listing all recordings or searching by text** – that will be added with the Library feature.

- **Mutations:** Implemented mutations include an `upsertRecording(input)` which either updates an existing recording by ISRC or creates a new Work+Recording [118] [119], and mutations for creating or updating Licenses (`createLicense`, `updateLicense`) [120] [121]. The `upsertRecording` mutation is used by the ingest service to add music data: it normalizes the ISRC, ensures required fields, checks if the recording exists, then either updates it (and triggers pubsub events) or creates a new Work & Recording [122] [123] [124].
- **Subscriptions:** The GraphQL layer is set up to support subscriptions (GraphQL over WebSocket). In the code, we see `useServer` from `graphql-ws` is used with the Yoga server [125], and pub-sub events are defined (e.g. `publishWorkUpdated`, `publishRecordingUpdated`, etc.) [126]. These likely publish to an in-memory or Redis-based pubsub (the code suggests using the `subscriptions` object and Redis for cross-instance events). For now, subscriptions include `workUpdated`, `recordingUpdated`, `ledgerEntryCreated`, `cycleCheckpointClosed` as per the code hint [127]. This means the GraphQL API can push real-time updates to clients (e.g. if a Recording is updated via upsert, it calls `publishRecordingUpdated` which will notify subscribers).
- **GraphQL Client usage:** The Studio front-end does not yet heavily use GraphQL (Phase 1 mostly used REST endpoints). But future library features will likely use GraphQL for queries like `libraryTracks` etc. The repo includes a mention of tRPC for Studio, but currently the implemented pattern is Next.js API routes and direct fetch calls for Studio needs. We anticipate adding GraphQL queries for the library and hooking them up via something like Apollo or a fetch.

- **Authentication:** The GraphQL service does not integrate with NextAuth directly; it likely expects to be behind an authenticated boundary or uses JWTs. In `createYoga`, it could be configured with an authentication check (not clearly shown in snippet). Possibly, the Studio app would connect to GraphQL with a token. (The environment mentions `PUBSUB_URL` and GraphQL WS port, but not an explicit auth mechanism) [5]. For now, consider GraphQL open or using API secret if needed for certain mutations (like ingest).

- **Next.js REST API (Studio):** The Studio web app defines several API routes under `apps/studio-web/app/api/`. These cover functionalities needed in Phase 1:

- `GET /api/sessions` – list sessions or fetch one by ID [128].
- `POST /api/sessions` – create a new studio session [129].
- (Likely implemented in `app/api/sessions/route.ts` – not shown above, but spec defines it.)
- `GET /api/presence?roomId=` – get the list of members in a realtime room (reads from Redis) [130].
- `POST /api/presence` – update presence (join/heartbeat) – though this might have been simplified by using the WebSocket directly for presence.
- `DELETE /api/presence` – remove a presence entry (if not implicitly handled by WS disconnect).
- `POST /api/upload` – handle an upload (as described in detail above) [58].
- `GET /api/upload/list?sessionId=` – list uploads for the user (optionally filtered to a session) [88]. This returns an array of uploads with their download URLs computed [131].
- `GET /api/upload/[id]` – fetch a single upload's metadata and a fresh download URL [132].
- `DELETE /api/upload/[id]` – delete an upload (and its file) [133].

- `GET /api/upload/[id]/file` – download the raw file content (for local storage cases) [79] [92].
- `POST /api/export` – initiate an export (likely creating an Export record and enqueuing a job). The spec shows it returning `{ url, exportId }` as a stub [134]. In implementation, it probably creates an Export with status "pending" and triggers a BullMQ job via Redis. The actual export process is handled by `export-worker`, which updates the Export record and publishes progress to a Redis channel. The Studio UI polls or subscribes to updates (the realtime gateway subscribes to the `export:progress` channel and broadcasts to WS clients [135]; the Studio app has a `useExportEvents` hook likely to receive those).
- `GET /api/export?id=` – get status of an export by ID (returns URL if ready, etc.) [136].
- `GET /api/export/[id]/download` – download the exported file (similar to upload file route). We saw this implemented: it checks the export record, if completed and local storage, reads the file and responds, or if S3, redirects to a signed URL [137] [138].
- The NextAuth routes under `/api/auth/[...nextauth]` handle login callbacks, etc., as part of NextAuth; these were set up via the `[...nextauth].ts` file using the `auth` configuration from `lib/auth.ts` [139].

These REST endpoints are used directly by the Studio React components (via `fetch` or router actions). For example, the session page uses the presence API and opens a WebSocket, the upload panel calls the upload APIs, and the export panel calls the export API and listens on WS for progress.

- **Insight API:** The Insight (Phase 3) involves:
- The ingest FastAPI service provides some HTTP endpoints for data ingestion (`/ingest/rss`, etc.).
- The alerts service provides a Fastify API for managing alert channels and rules (`/channels`, `/rules`, etc.) [140].
- The Insight Next.js app uses server-side queries to ClickHouse (via the ClickHouse JS client) for analytics; it might not expose its own API routes but uses Next server actions to get data from ClickHouse [141].

- There may also be a GraphQL query on the main Graph API to fetch entity tags or similar for insight.

- **Search**: At present, **there is no dedicated search API implemented**. Neither GraphQL nor REST offers a text search over recordings or uploads. The only search-like operations are filtering by IDs or fields (e.g. `licenses(workId)` or listing sessions by user). There is also no full-text index yet. This is identified as a gap – the roadmap's Milestone 1 will add a search query for library tracks (likely using Postgres full-text or `ILIKE`). Similarly, the news/Insight search (if any) would be handled via ClickHouse or specific endpoints in ingest, but that is also nascent.

- **Client integration:**

- The Studio web app currently uses built-in fetch calls (see `usePresence.ts` using fetch for presence listing [142], and direct anchor links for download). There is mention of tRPC in the stack, but at least up to now, the implemented pattern is REST. The introduction of a library GraphQL may involve using Apollo Client or URQL in the Next.js app, or continuing with REST endpoints for simplicity.
- The Insight web app likely uses either direct ClickHouse queries or minimal APIs to fetch analytics, as it imports `@clickhouse/client` in its package [143].

Summary: The API surface is currently split between a **GraphQL Core** service (for music metadata and ledger) and a **Next.js REST API** (for user-driven operations like sessions, uploads, and real-time collab). Each has clear responsibilities, but they will be extended. We plan to introduce new GraphQL operations for the Music Library (Track queries, mutations for playlists, etc.) and possibly unify the client access through GraphQL for those features. The GraphQL service is already part of the monorepo and can be queried by the Studio app (just need to configure the Apollo client endpoint).

Additionally, as we integrate LiveKit (milestone 2), we'll add a small API endpoint for generating access tokens for LiveKit (since LiveKit uses JWT tokens for clients to join rooms). This will live in the Studio API routes (e.g. `POST /api/livekit/token`) and will be protected by an admin secret to sign the token.

All API additions will follow the established patterns: use Prisma for DB changes, respect the NextAuth user session for auth, and utilize the storage/telemetry packages for consistency.

---

docs/STATUS_GAPS.md

Below is a checklist of key capability areas, indicating whether they are **implemented** in the current Omnisonic repo or **missing/incomplete**, with references to the code or config. Each item includes evidence from the repository (file paths or code) supporting the status, or is marked **UNKNOWN** if the code was searched but not found.

- [ ] **Music Library UI (track listing & management): Missing.** The repo currently has no page or component for browsing a user's imported music outside of a session. There is no `/library` route in the Next.js app, and the Phase 1 spec lists only landing, about, sessions pages [144] with no mention of a library UI. Uploads not tied to a session are not surfaced anywhere in the UI (no "My Tracks" page exists). This is a major gap: users cannot view or play the songs they've uploaded outside the session context.
- [ ] **Music Search (filtering and finding tracks): Missing.** There is no search bar or search API for music content in the current implementation. The GraphQL API has no query for searching tracks (only specific ID lookups like `work(id)` or listing all works isn't present) [115]. The Next.js app also has no search field except for session name filter. We searched the code for any full-text search or use of Postgres `ILIKE` and found nothing relevant. Thus, users cannot search songs by title/artist in the current system.
- [ ] **Playlists / Collections: Missing.** There is no model or UI for playlists. The Prisma schema has no `Playlist` or similar model [145], and no code references to playlists. Users cannot group tracks into playlists or sets – an anticipated feature that is not yet implemented at all.
- [x] **Realtime Presence & Text Collaboration: Partially Implemented.** Users can see who is in a session and share some basic state, but it's limited. The **presence** system is in place using WebSockets and Redis: the `realtime-gateway` tracks `rooms` and broadcasts join/leave events [146] [147], and the Studio UI uses `usePresence` to maintain a member list [148] [149]. However, **text chat or collaborative editing** features are not present (no chat messages or shared cursors). The presence is currently just showing active participants.
- [ ] **Realtime Audio Collaboration (WebRTC/LiveKit): Not Implemented.** There is no support for streaming audio between users in real time yet. The Phase 1 implementation uses a custom WS for presence, but audio/video is not integrated – no use of WebRTC or LiveKit in the code (searching the

repo for "livekit" yields nothing). The spec explicitly lists LiveKit integration as a future item [150]. So, currently, multiple users in a session cannot hear each other or jam live; they can only upload tracks and share presence. This is a critical missing piece for true real-time collaboration.

- [x] **File Upload & Storage: Implemented (basic).** Users can upload files (e.g. audio stems) to sessions. The upload pipeline is present as described (Next.js route + storage module). The system supports local, MinIO, or S3 storage and returns download links. We verified the code: `/api/upload` route saves files and a record [64] [67], and files can be downloaded via generated URLs or the file API [89] [92]. However, **metadata extraction** from audio files is not done yet – uploads are stored with filename only, no ID3 tags are parsed (the code does not attempt to read audio metadata). That will be addressed in the library import feature. Also, **upload UI** is rudimentary (one file at a time via an input; no progress bar aside from "Uploading..." text).
- [x] **Mixdown Export (offline rendering): Implemented (mock).** Users can request an "Export mixdown" of a session. The infrastructure exists: clicking "Render mixdown" in the UI triggers an API call (we see in the E2E test it clicks a "Render mixdown" button [151]). The backend sets up an `Export` record and the `export-worker` service fakes the render (generating a silent audio file of fixed length, since FFmpeg is used to create a dummy sine wave) [152]. The worker updates progress and final file URL in the DB [12] [153], and progress is broadcast over WebSocket (via Redis pubsub) [135]. The UI updates the status. **Gaps:** This is a stub implementation (no actual mixing of session tracks occurs, it's a dummy output). Also, no real-time audio mixing in-browser; this is an offline process.
- [ ] **Music Library Metadata & Tags: Not Implemented.** There's no system to store rich metadata for tracks in the user's library. The current data model lacks fields like album, genre, BPM, key for user-uploaded tracks. The `Recording` model has some of these (genre, isrc, etc.) but is not used for user imports yet. Tagging of tracks by mood or custom tags is not present (the only tags in DB are for news items via `EntityTag`, which is unrelated to library tracks). So, features like genre classification, BPM detection, or user tagging of tracks are missing. (The roadmap's AI milestone will address some of these with an analysis service.)
- [ ] **AI/ML Features (auto-tagging, recommendations): Not Implemented.** The repository contains placeholders and planning for AI but no functional AI features yet:
- **Auto-tagging:** The Phase 3 spec outlines an ingest tagging module and mentions embeddings [154], and environment variables for enabling embeddings exist [155]. However, actual code implementing ML inference is absent – `services/ingest` likely has stubs (and requires installing `torch` for embeddings, which is optional) [156]. We did not find any code that computes audio features (BPM, key) or content-based tags for tracks.
- **Recommendations:** No recommendation engine is active. There's no code for "users who liked X also like Y" or similar. The Phase 3 plan mentions using metadata and possibly vector search (pgvector/Faiss) for "Similar Tracks", but that is future work [157].
- **AI-assisted creation:** No AI mixing or mastering tools are integrated at this time. In summary, AI/ML capabilities are **planned but currently absent** – the codebase has structures to accommodate tags and embeddings (e.g., `TaggingMethod` enum [158]), but no actual ML pipelines run.
- [ ] **Observability (Monitoring & Logging): Basic, Incomplete.** Some observability is in place, but more for debugging than full monitoring:
- **Tracing:** OpenTelemetry is set up in the code. Each Next.js API route uses a tracer (`withSpan("studio-web-api", "<routeName>", ...)`) to trace the request [58], and the realtime gateway does similar for WS connections [159] [160]. These spans are logged to stdout (since no exporter is configured, by design) [26]. This helps during dev, but in production one might pipe these to a log aggregator.

- **Logging:** The backend services use console logs or simple loggers (the export-worker uses `pino` for structured logging [161] [162]). There's no central log collection, and no user-facing monitoring dashboard for system health.
- **Metrics:** No explicit metrics (like Prometheus or statsd) are emitted by the code. Also, no uptime or performance monitoring service is integrated.
- **Audit Logging:** There is no audit trail for user actions (e.g., who edited what metadata when). If needed, we'd have to add explicit logging or DB records.
- **Unknown:** No evidence of alerting on errors aside from the console and the developer's own observation. The Alerts service is meant for domain alerts (like news thresholds), not system health. So observability is minimal right now – sufficient for development (with logs and traces to console) but not a comprehensive production monitoring solution.
- [ ] **Security Features (beyond basic auth): Minimal.** Outside of authentication (NextAuth), there are few security-related features:
- **Authorization:** As mentioned, resource access is enforced mostly by checking ownership (userId matches) in API routes [53] [163]. There are no role-based restrictions coded (no admin vs user distinctions yet).
- **Input Validation:** Some input validation is present (e.g., zod schema for credentials [164], or ISRC/ISWC format validation in GraphQL [165]). But there is likely more to harden (no extensive validation on text fields or files beyond type/size).
- **Protection:** No explicit rate limiting on API endpoints, no mention of sanitization for preventing XSS/SQL injection (though Prisma and Next mitigate most SQL injection and XSS by design). CSRF protection is handled by NextAuth for its routes, and Next.js API routes by default require same-site or proper CORS (the app is same origin).
- **File security:** Uploaded files are named safely (spaces replaced by dashes in keys) [65]. However, currently if someone knows a storage URL they might access another's file if permissions aren't checked – for local storage, the file API route checks userId, and for S3, we use signed URLs (so that's secure) [53] [166].
- **Secrets Management:** Secrets (DB password, OAuth secrets, etc.) are provided via env. The repo has a doc guiding to not commit secrets and use GitHub secrets for CI/CD [167] [168].
- **Missing:** Two-factor auth, account lockout, and other advanced security features are not present. Also, no encryption of user data at rest beyond what Postgres or S3 inherently do. These might not be critical at the current stage but would be considerations later.
- [ ] **Admin Tools: Missing.** There is no admin interface or admin-specific functionality in the current state:
- No admin web UI to manage users, content, or system settings.
- No moderation tools (e.g., removing a user's content, since this is mostly single-user at the moment).
- No admin APIs except what a regular user can do. The concept of an admin user isn't implemented (the `User` model has no role or flags for admin).
- The planned Phase 5 aims to introduce admin pages (e.g. content management, monitoring) [169], but as of now, this is entirely absent.
- [x] **Analytics & Insights: Partial (Phase 3 in progress).** Some groundwork is laid, but not yet delivering end-user value:
- ClickHouse is configured and an Insight web app exists [170]. The Phase 3 spec outlines tables like `news_items` and `entity_links` [171]. We anticipate the ingest service populating those. However, it's unclear if any data is actually in ClickHouse yet or if Insight pages are completed. The `apps/insight-web` has dependencies like Chart.js and Recharts [143], suggesting graphs to be shown. This is a work in progress.

- No analytics on user behavior or system usage is present (no tracking of plays, or active users, aside from what might be inferred from presence).
- Developer-oriented analytics (like performance traces) exist, but user-facing analytics (like how many times a track was played) are not implemented.

Each of the above **missing capabilities** is identified in project planning (see the Omnisonic roadmap PDF and internal prompts). The next sections (roadmap and PR prompts) describe how we will fill these gaps.

ROADMAP.md

Omnisonic "Library-First" Acceleration Roadmap
(This roadmap is milestone-driven and feature-focused, not date-based. It prioritizes delivering a robust personal music library and real-time collaboration features before moving on to AI enhancements and admin tools. Each milestone has clear acceptance criteria and will be delivered via small, reviewable pull requests. Feature flags will be used to maintain stability.)

Milestone 0: Baseline & Guardrails

- **Goal & Importance:** Ensure the development environment and codebase are well-understood and stable before adding major features. This milestone sets up the "ground truth" – mapping out the repo, verifying the stack runs end-to-end, and adding any needed tooling so subsequent work can proceed with confidence. Essentially, this is about **knowledge ramp-up and safety checks** so that we know what we have (and don't have) in the current Omnisonic implementation [172] [173].
- **Scope:** Documentation and minor config updates only. No new user-facing features. Scope includes writing a **Repo Map** document, confirming the dev setup (all services running locally), and adding any Architecture Decision Records (ADRs) for context. Also encompasses setting up continuous integration (if not already) for tests and linting as a guardrail.
- **Key Tasks:**
- **Repository Map & Plan:** Document the monorepo structure, identify where key pieces are (Next.js app, GraphQL service, Prisma schema, storage logic, etc.) [174]. Also outline the upcoming Milestone 1 implementation plan (to validate our understanding).
- **CI Pipeline (if missing):** Add GitHub Actions workflows for linting, type-checking, running tests, etc. (This was mentioned in earlier roadmap as a to-do) [175]. Ensures code quality for future PRs.
- **Local Run Verification:** Run the full stack (Postgres, Redis, Minio, all services) and fix any .env or script issues. Specifically verify that Studio web loads, sign-in works, file upload works, and an export can be queued and completed [176]. Fix any baseline bugs encountered.
- **ADR setup:** Create a `docs/decisions/` folder and add initial ADRs as needed (e.g., one explaining the monorepo structure choice, if not already present) [177].
- **Definition of Done:** We have a high-level understanding of the current system (documented in `docs/REPO_MAP.md`) and a concrete plan for Milestone 1 in place [178]. The entire stack can be started locally with documented instructions, and baseline functionality (upload, export, etc.) works without regression. CI pipelines are passing, giving us confidence to proceed. *No feature changes* are introduced in Milestone 0 – it is purely preparatory.
- **Deliverables:** `docs/REPO_MAP.md` (repo structure and data flow map) [178]; `docs/plans/M1_LIBRARY_PLAN.md` (a detailed implementation plan for the library features) [179]; initial ADRs; a passing CI build.

- **Notes/Risks:** This step mainly mitigates the risk of misunderstanding the existing code. By reading and running everything now, we reduce surprises in later milestones. There is little risk to user functionality since this milestone doesn't alter features. The main risk is time spent—must ensure we don't get stuck in analysis and delay actual feature work beyond a reasonable duration (should be time-boxed).

Milestone 1: Music Library & Smart Search

- **Goal & Importance:** Deliver a **personal music library** experience within Omnisonic Studio. This is the highest priority because the core value of Omnisonic (local-first music management) is not fully realized without a library. Currently, users cannot manage or search their own tracks outside of sessions – this milestone enables that. We will introduce a **Track** entity for user-imported songs, allow users to import audio files as library tracks with automatic metadata extraction, provide UI to browse and play these tracks, and implement a basic but effective search and filtering mechanism [180]. This makes Omnisonic immediately more useful as a standalone music app, not just a collaborative DAW.
- **Scope:** This milestone covers the end-to-end vertical slice of library management:
- **Database:** Add new models (or extend existing ones) for tracks, playlists, and tags as needed for a minimal library.
- **Backend:** Implement GraphQL queries/mutations and/or REST endpoints to manage tracks and playlists (create/read/update/delete track metadata, create playlists, add/remove tracks, search tracks).
- **File Import:** Enhance the upload pipeline to import files as library tracks with metadata. Possibly a new API route or GraphQL mutation that handles uploading multiple files and extracting metadata (ID3 tags).
- **Frontend:** New pages in the Studio app: a Library view (`/library`) listing tracks with search & filters, a detail page for a track (`/library/[id]`), and possibly a basic playlist UI. Also a modal or form to edit track metadata and a way to play tracks (e.g., using an HTML5 audio element or existing player component).
- **Search:** Implement a "smart search v1" on the library – likely using Postgres full-text search or `ILIKE` queries for title/artist, plus some simple ranking by relevance. Include filters (e.g., by genre, BPM range if metadata is there).
- **Auth & Permissions:** Ensure that tracks are private to the owning user by default – the queries should scope to `ownerUserId = currentUser`. Also enforce that only the owner (or an admin, when that exists) can mutate a track or playlist.
- **Non-goals (deferred):** Waveform generation for tracks (visual waveforms), advanced audio analysis (AI will come later), collaborative sharing of libraries (all tracks are just user-owned for now), cloud backup (still local-first).
- **Key Tasks & Implementation Plan:**
- **Schema Changes (Prisma):** Introduce a **Track** model (or reuse the Recording model by repurposing it for user tracks). Proposed fields for Track: `id`, `userId` (owner), `title`, `artist`, `album`, `year`, `genre`, `durationMs`, `bpm` (optional), `key` (optional musical key), `artworkUrl` (optional), `source` (enum: IMPORTED vs SESSION_STEM vs RECORDING) to indicate origin, `fileId` or storage fields linking to the file (we might use the existing `Upload` for file info) [181]

[182]. Alternatively, extend the existing `Upload` model with metadata fields, but better to separate concerns and use a new Track model referencing Upload or sharing the storageKey.
  - Add a **Playlist** model with at least: `id`, `userId`, `name`, plus a join table **PlaylistItem** (`playlistId`, `trackId`, `position`). This allows ordered lists of tracks [183].
  - Add a simple tagging solution: either a string array field `tags` on Track [184], or a separate Tag model and a TrackTag join table. Initially, we might choose a text array for simplicity.
  - Ensure migrations are created for these changes.

- **GraphQL API & Backend Logic:** In the GraphQL schema (services/graph-api):
  - Add GraphQL types for Track and Playlist (and PlaylistItem if needed). Define Query fields like `libraryTracks(query, filters, limit, offset)` to search current user's tracks [185], `libraryTrack(id)` to fetch one track, `playlists` (current user's playlists), `playlist(id)` [186].
  - Add Mutations: `updateTrackMetadata(id, input)` for editing track fields [187], `createPlaylist(name)`, `addTrackToPlaylist(playlistId, trackId, position?)`, `removeTrackFromPlaylist`, `reorderPlaylistItems`, and `deletePlaylist` [187].
  - Implement resolvers for these using Prisma. The resolvers must enforce that the user can only access their own tracks/playlists (e.g., via `where: { userId: context.user.id }` in queries) [188]. Use NextAuth JWT or session token to get userId in GraphQL context (or call a common auth check).
  - For search, implement the `libraryTracks` query to accept a text `q` and possibly filters like `artist`, `genre`, BPM range, key, etc. [185]. Use Prisma/SQL to filter: e.g., if Postgres full-text is enabled, use `contains` or `searchVector`; otherwise use case-insensitive partial match (`ILIKE`). We can rank results by matching title vs artist (maybe fetch and sort in memory if small scale).
  - Leverage existing patterns: the codebase already uses Prisma via `@db/client`. We continue that. Also, for any complex logic (like metadata parsing), consider writing a helper in `packages/core` or directly in the API route.
  - Consider whether to implement file import as a GraphQL mutation vs a REST endpoint. Since the rest of Studio uses REST for file upload, we might add a **REST endpoint** `/api/library/import` that accepts multiple files, uses the same storage logic as `/api/upload`, but then also extracts metadata and creates Track entries. Alternatively, implement a GraphQL mutation like `importTracks(files: [Upload!]!)` if GraphQL file uploads are configured. Given the existing infrastructure, a REST endpoint might be simpler (Next.js can handle FormData easily). We will do whichever aligns better with current patterns (likely REST).
  - **Metadata Extraction:** Integrate an ID3 parsing library (e.g. `music-metadata` for Node) to extract metadata from audio files upon import [189]. Fields to extract: title, artist, album, year, genre, track duration. Also, extract embedded album art image if present: store it via the storage module (similar to how exports are stored) and save the artwork URL in the Track. If a tag is missing (e.g., no title in the file), use the filename as fallback for title [190] [191]. If multiple files are uploaded, process each and return all created tracks.
  - Ensure large files are handled: perhaps stream to storage in chunks rather than Buffering the entire file (the `music-metadata` library can parse from a stream). Given Node's capacity and 100MB limit, buffering might be okay, but we note it as an area to watch (memory usage) [190].

- **Studio Web UI – Library Pages:** Build the front-end components for the library:
    - **Library List Page (`/library`):** A page that lists the user's tracks in a table or list format [192] [193]. Include a search bar at top and filters for common fields (e.g., dropdowns or multi-select for genre, maybe a BPM range slider if we have BPM data) [194]. The list should be paginated or infinite-scroll if the library is large (initially, offset/limit pagination is fine). Use the new GraphQL query (or REST API) to fetch tracks. This page also provides an “Import” button for user to add new files to the library (could open a file picker and call the import API).
    - **Track Detail Page (`/library/[id]`):** Shows details of a single track [195]. Display all metadata (title, artist, album, etc.), and potentially a waveform placeholder (we won't implement actual waveform generation now) [195]. Include controls: a Play/Pause button to play the track (use an HTML5 `<audio>` element or a simple custom audio player), an “Edit” button to edit metadata, and “Add to Playlist” to put the track into a playlist [195].
    - **Playback integration:** If there's an existing audio player component or hooks in the app, use that. If not, a minimal solution is to use an `<audio>` tag for each track detail (or an audio element in a fixed player component) that loads the track’s `downloadUrl`. Ensure that if the track is private, the URL is either a signed URL or goes through our auth-protected route (since our `/api/upload/[id]/file` is protected, the Next.js session cookie will authorize it). Initially, playing one track at a time is enough (no continuous queue).
    - **Edit Metadata:** Implement a modal dialog or separate page to edit track metadata [196]. This would utilize the mutation `updateTrackMetadata`. For now, assume simple text fields for title, artist, etc. Possibly reuse or adapt the design system components (shadcn/UI) for forms and modal. On save, update the UI optimistically.
    - **Playlist UI:** Create a basic Playlist management interface [197]. Perhaps a section on the library page or a separate page `/playlists`. At minimum:
    - A way to create a new playlist (a modal or inline form to name it).
    - A list of playlists (with track counts maybe).
    - On a track's detail or in the list, an option "Add to Playlist" which pops up a list of playlists or creates a new one.
    - A Playlist detail page or modal: showing all tracks in the playlist in order, with ability to remove or reorder (drag-and-drop or up/down controls). This can be somewhat minimal due to time – the main focus is track management, but having at least one playlist ensures the data model is exercised. If time is short, we might implement the backend for playlists and a simple way to add a track to a default playlist (like "Favorites") as a placeholder, deferring a full UI.
- **Integration & Search:** Wire the search bar to call the search API (GraphQL query with `q` and filters) as the user types (debounce input) [194]. Implement basic ranking on the backend: e.g. if search term matches title start, rank higher than if it matches in the middle or matches artist. If possible, implement a "did you mean" suggestion for minor typos: this could be done by using the trigram similarity in Postgres (`pg_trgm` extension) or a simple Levenshtein distance on small string sets. However, given time, we might skip “did you mean” in v1 or implement a very rudimentary version (like if no results, check for any similar titles by Levenshtein <= 2).
    - Also consider implementing **Saved Searches** as suggested (store a query by name) [198] [199]. This is a nice-to-have. If time allows, add a `SavedSearch` model (id, userId, name, query JSON blob) and allow user to save the current filter criteria with a name, and later load it. This would be extra – not critical for v1 – so it could be feature-flagged or deferred.

- **Testing:** Write tests to cover the new functionality:
    - **Backend tests:** If there's a testing framework for the GraphQL API or REST routes, add integration tests. For example, using Jest to call the GraphQL server with a sample context:
    - Test creating a playlist and adding a track, then querying the playlist returns that track.
    - Test updating track metadata actually persists changes in DB.
    - Test that searching by title returns the correct track and that unauthorized access (user A cannot see user B's track) is enforced [200].
    - If no infrastructure for backend tests is in place, at least ensure the E2E test covers basic library use (maybe extend Playwright to import a track and then search for it).
    - **Frontend tests:** Add React component tests for the new pages if possible (the repo uses Playwright for E2E, and likely React Testing Library or similar for unit tests). Write tests for:
    - Library page renders a list of tracks given a mocked query result.
    - Editing a track updates the display.
    - (If time) an E2E scenario: import a track via the UI, then find it on the library page and play it.
- **Docs:** Update or create documentation:
    - `docs/features/library-api.md` – document the new GraphQL API endpoints (types, queries, mutations with examples) [201] [202].
    - `docs/features/library-ui.md` – document how to use the library UI, perhaps with screenshots or at least a description of the pages and how search works [203] [204].
    - Update the main README or Phase 1 spec to note that the library feature now exists, and how to enable it if behind a feature flag.
    - Possibly an ADR about choosing to unify Track with Upload vs separate (record design decisions about the library schema).
- **Acceptance Criteria (Definition of Done):**
- A user can upload audio files as library tracks (not tied to a session) and the system stores them with metadata. Specifically, after import, the new tracks appear in `/library` with correct titles (filename or ID3 title) and basic tags.
- The user can view a list of all their imported tracks, with sorting or pagination working for a reasonably large set (e.g. test with 50+ tracks).
- The user can play a track from the library through the UI (hear audio).
- The user can edit a track’s metadata (e.g., correct the title or add an artist) and see the change persist.
- The user can create a playlist and add tracks to it; then view that playlist with those tracks in the specified order.
- The search bar in the library allows finding tracks by title or artist (e.g., search “beat” filters to tracks with “beat” in name or artist). Results come back quickly (under a second for 1000 tracks). Basic filters (like by genre or BPM if applicable) work.
- All existing functionality remains unaffected: session uploads still work as before (session file uploads should perhaps also create Tracks with sourceType=SESSION_STEM if we unify, but that might be a later enhancement – at minimum ensure that nothing breaks in session flow). Backwards compatibility: old Uploads tied to sessions still accessible in those sessions.
- Tests covering these scenarios pass. No critical bugs (e.g., SQL errors, crashes, or security holes like accessing someone else’s tracks) are present.
- **PR Plan:** We will break this milestone into **4 PRs** (at least) for easier review:
- **1A: Database & GraphQL foundation** – Add Track/Playlist models and migrations, plus GraphQL schema/types and resolvers for basic CRUD (no file upload yet) [181] [185]. This PR sets up the data

layer and ensures the app compiles with the new schema. It will include unit tests for the resolvers (using an in-memory SQLite or test DB).

- **1B: File Import API & Metadata Extraction** – Implement the new import route or mutation that handles file upload and creates Track entries [205] [206]. This includes integrating the metadata parsing library. We'll also update the storage logic if needed. Tests for importing multiple files and verifying metadata fallback will be included [207]. This might touch both the Next.js API and GraphQL (depending on approach). Feature flag (if any) for enabling library could be introduced here (e.g., an env `ENABLE_LIBRARY` to wrap the UI/routes).
- **1C: Library UI pages** – Add the `/library` and `/library/[id]` pages and associated components (track list, search bar, track detail, edit modal, playlist UI) [192] [195]. Use React hooks to fetch data from the new APIs. This PR will focus on the frontend, with stubbed backend calls if necessary. Include component-level tests.
- **1D: Search & Polish** – Improve the search ranking and add "did you mean" suggestions if possible [208] [209]. Implement saved searches if we decided to. Tweak any UI issues, add loading states, etc. Also, documentation updates and final testing (Playwright E2E scenario for library) happen here.

Each PR will be around a <300-line change set, aiming for <30 min review. They will be feature-flagged as needed (for instance, we could hide the Library menu link behind a config flag until fully ready). - **Dependencies:** This milestone depends on a stable baseline (Milestone 0). It introduces a dependency on an ID3 parsing library (NodeJS), which we'll add. Ensure ffmpeg is available if we try to get duration via ffprobe – or we skip that if not easily accessible (since we have duration from metadata or we can approximate by audio element once loaded). - No external cloud services needed; everything remains local or uses existing S3 if configured. Ensure Postgres has the `pg_trgm` extension enabled if we do similarity search – if not, we either enable it via a migration or adjust plan (maybe just do simpler search). - **Migration Strategy:** Adding new models and fields is backwards-compatible; existing data (sessions, uploads) remains unaffected. We are not renaming or removing any existing fields in this milestone, just extending the schema. Thus, running the new code with migrated DB should not break old functionality. For existing uploaded files that were imported outside a session (currently none through UI), we might later migrate them into Track records manually (but since there's no prior library, nothing to migrate now). Session-related flows remain intact: e.g., we will not yet unify session stems into the Track table in this milestone to avoid interfering with session use. If we choose to treat session uploads as Tracks with sourceType, we'll do so carefully such that all code using Upload.sessionId continues to work (likely by leaving the Upload mechanism as-is for sessions and possibly linking it to Track optionally). In short, no breaking changes for users or data. - **Risks & Mitigations:** - *Complexity:* This milestone touches many parts of the stack (DB, backend, frontend). Risk of something not integrating cleanly (e.g., GraphQL changes might conflict with Next API auth). **Mitigation:** Develop and test in small PRs, behind a feature flag (so we can merge partial functionality without exposing to end users until complete). - *Performance:* Loading a large library or doing search could be slow if not indexed. **Mitigation:** Ensure DB indexes on key fields (title, artist) or use Postgres full-text for efficiency. Test with ~1000 dummy tracks. We can add an index on Track.title for ILIKE queries, and consider full-text index if needed. - *Metadata accuracy:* Relying on user file tags might yield messy data (missing or inconsistent metadata). **Mitigation:** Provide sensible fallbacks (filename for title, "Unknown Artist" if artist missing). Log warnings for files with no readable tags. Possibly allow user to edit metadata post-import (we do). - *File sizes:* Importing many large files could be heavy on memory if we buffer them. **Mitigation:** We have a 100MB limit already. We can stream to storage. If performance issues arise, we might implement chunked upload or background processing for imports (not in v1). - *Scope creep:* There are many nice-to-haves (saved searches, waveform, etc.). **Mitigation:** Stick to essentials first (list tracks, play, edit, basic search). Defer advanced features to future (noting them but not doing now). - *Security:* Opening a new API surface means new potential auth issues. **Mitigation:** Thoroughly enforce user scoping in queries and mutations. Write tests for cross-user access attempt (should be forbidden) [188] [200]. - *UI/UX:* The library UI needs to integrate with the existing design (tailwind + shadcn components). Risk of inconsistency or minor bugs. **Mitigation:** Follow existing UI patterns (e.g., see how sessions list is implemented) and test in multiple browsers. We'll also document any UI limitations (like no mobile optimization initially, if that's out of scope).

Milestone 2: LiveKit-Powered Realtime Collaboration

- **Goal & Importance:** Introduce **low-latency audio/video collaboration** so that multiple users can jam together or communicate in a session. This milestone integrates **LiveKit**, an open source WebRTC SFU (Selective Forwarding Unit), to handle real-time audio streams among users. Currently, Omnisonic users can collaborate asynchronously (share tracks) but cannot truly play or sing together live – adding this will fulfill the real-time collaboration promise of "Studio". LiveKit provides the building blocks for rooms, participant management, and media streaming, which significantly upgrades the collaboration experience.
- **Scope:**
- Setup a **LiveKit server** (for dev and production) and integrate its client SDK into the Studio app.
- Provide a backend service or endpoint to issue access tokens for LiveKit (since clients need a token with a secret to join rooms).
- **Audio Rooms:** On the Studio session page, allow users to join a LiveKit room corresponding to that session. Once joined, users can broadcast their microphone audio to others and hear others' streams. Also display which users are connected and who's speaking.
- Basic controls: mute/unmute microphone, adjust volume or output device, and leave room. Possibly also allow an "always on" vs "push-to-talk" mode as a setting.
- Use **LiveKit's JS SDK** and possibly their pre-built React components to expedite UI integration [210].
- All of this behind a **feature flag** (e.g., `ENABLE_LIVEKIT`), so we can disable it easily or load LiveKit scripts only when enabled [211].
- Non-goal: We won't implement musical time synchronization or network MIDI in this milestone (LiveKit will handle audio latency but not ensure beats align – that's complex and out of scope now). Also, we'll focus on audio; video or screen-sharing could be added later but are not priority.
- **Key Tasks:**
- **LiveKit Server Deployment:** Decide how to run LiveKit for development and prod:
    - For dev, we can use `livekit-server --dev` which is a single binary that runs an SFU with no auth required (it auto-accepts any token). This is simplest for local testing.
    - For production, we likely self-host LiveKit. We should prepare Docker Compose configs or helm charts. Possibly we add a `infra/livekit` with a docker-compose that sets up LiveKit server and a TURN server, etc. However, since this is an app roadmap, we document it rather than fully automate it now (maybe as an optional deploy script) [212].
    - Acquire or generate an API key and secret for LiveKit (for token signing). In dev mode, LiveKit might provide a default key/secret (`devkey`/`devsecret`). In production, the operator will set real secrets.
    - **ENV Config:** Add env vars: `LIVEKIT_URL` (e.g., `ws://localhost:7880` for dev server), `LIVEKIT_API_KEY`, `LIVEKIT_API_SECRET` for token generation, and possibly `LIVEKIT_TLS` or domain if needed. Document where to store these (likely in `.env.local` for dev, and GitHub secrets for prod).
- **Token Endpoint:** Implement a secure endpoint to mint LiveKit access tokens. Likely a Next.js API route: `POST /api/livekit/token` that takes a room name (e.g., `session-<sessionId>`) and

optionally participant identity, and returns a token string [213]. Use a LiveKit server SDK or a JWT library to create the token signed with our secret. (LiveKit tokens are JWTs with embedded grant info, specifying room join permissions.)

  - This endpoint must ensure the caller is an authenticated user and allowed to join that room (e.g., if we name rooms by sessionId, ensure the user has access to that session). In practice, since any session member can join audio, we check that the user either owns the session or is listed as a collaborator. Right now, collaboration is informal (no invitation system), so perhaps we allow any authenticated user to attempt join? Better: restrict to the session owner for now, until we have a notion of invited collaborators. Alternatively, if we consider presence or a future invite list, enforce those. For milestone 2, it might be simplest to let anyone with the link join – but that could be abused, so maybe require them to be in the presence list (meaning they opened the session page). If the session page itself is protected and not sharable, it's implicitly only the owner that can access currently. We might allow multi-user sessions by letting other users discover session IDs – but there's no UI for that now. We can expand later with invites.
  - The token should grant publish and subscribe permissions for that room.
  - Feature flagging: if `ENABLE_LIVEKIT` is false, this endpoint can return an error or be disabled.

- **Frontend Integration (Live Jam UI):**
  - Add LiveKit's SDK: e.g., install `livekit-client` npm package [210]. Optionally, use `@livekit/components-react` for prebuilt UI widgets (participant list, etc.) [210] – these come with a React context for LiveKit.
  - On the session detail page (`/sessions/[id]`), introduce a **Live Jam** panel (could be a sidebar or a section) [214]. If `ENABLE_LIVEKIT` is off, this UI is hidden entirely [211].
  - The panel shows a **Join Live Jam** button when not connected [215]. When clicked:
  - It calls our `POST /api/livekit/token` to get a token (sending room = `session-<id>`).
  - Initializes a LiveKit `Room` instance and connects to the LiveKit server via its WS URL with that token.
  - After connecting, it will have the local participant and any remote participants.
  - It then immediately captures the user's microphone audio track (prompt for permission via browser; we decide if we do this on join or have a separate "mic on" toggle – probably join and unmute are separate steps).
  - Publish the audio track to the LiveKit room (so others can hear).
  - Subscribe to remote tracks: for each participant that joins, play their audio. The LiveKit SDK will provide events for participant connected, track published, etc., which we handle. We likely just rely on LiveKit's built-in handling or attach each remote track to an <audio> element.
  - The panel updates to show **controls**:
  - A "Leave" button to disconnect from the room.
  - A microphone mute/unmute toggle [216]. If using LiveKit's React components, they have pre-made mic controls; or we manually call `room.localParticipant.setMicrophoneEnabled(false)` etc.
  - A list of participants currently in the jam, with an indicator if they are speaking (LiveKit provides audio level or speaking status).
  - Possibly volume control for the remote audio (the browser combined output or per track).
  - If "push-to-talk" mode is preferred, we might implement that: e.g., only capture audio while a key is held or button is pressed. But likely start with a simple toggle (always on vs muted).

    - UI/UX details: LiveKit's component library can style these controls; if not, we can craft minimal UI (some icons for mic on/off and a list of names). We'll use the user's existing name (from NextAuth profile) or some display name (the presence system currently uses displayName in handshake, which is basically their name or email prefix) [217]. We can pass that to LiveKit as the participant identity or name.
    - Show an indicator if connection quality is poor or if LiveKit server isn't reachable (e.g., if token fetch fails or connect times out, show an error suggesting to check server or flag) [218].
- **Back-end LiveKit Worker (Optional):** Not strictly needed, but in production we might run LiveKit as a separate server. We will provide documentation or config:
    - Provide a `deploy/livekit/README.md` with instructions for running LiveKit in prod (like using their Docker image, and opening ports, TLS via Caddy, etc.) [219].
    - Possibly a `docker-compose` snippet for LiveKit + Caddy (for TLS) for those self-hosting, as suggested [220].
    - Document environment variables needed for Omnisonic to integrate (like the `LIVEKIT_URL` that clients should use, which could be a public wss URL).
- **Testing:**
    - This is tricky to automate fully. We can write a basic integration test for the token endpoint (call it with a test user session, check we get a JWT). We can verify the JWT decodes to expected room name and identity (without verifying signature, since we generate it).
    - For the client side, unit tests might be limited (perhaps we can mock the LiveKit client). Instead, do a manual two-browser test:
    - Launch two instances (or two different browsers) of the dev server, log in as two different users (since currently there's no share, one hack is to have both join the same session by ID if we know it).
    - Click "Join Live Jam" on both and audibly verify they can hear each other. We'll document this as a verification step.
    - Possibly use LiveKit's test utilities or a headless mode to simulate participants. But given this is real-time media, automated test might be out-of-scope. We ensure at least that connecting doesn't throw errors and UI updates state accordingly (that we can test by mocking `Room.connect` to resolve).

- **Docs & Feature Flag:**

    - Write `docs/features/livekit.md` explaining how to test the live jamming with two users, how to configure LiveKit server, etc. [221].
    - Add notes in README about the `ENABLE_LIVEKIT` flag and default off in .env.example, plus mention any special setup (like running `livekit-server` locally).
    - Clearly mark this feature as experimental for now.

- **Acceptance Criteria:**

- When LiveKit is enabled and configured, two or more users in the same session can hear each other's live audio. For example, User A and User B open session X, they both click "Join Live Jam", allow mic access, and then User A speaking or playing instrument into their mic is heard by User B within ~200ms latency (typical local).
- The UI shows who is in the jam (at least their names or a count). When someone disconnects or mutes, the UI updates appropriately (e.g., "User B left the jam" or their status icon goes off).

- Users can mute/unmute themselves and see their own status. Muting stops their audio from being sent (confirmed by the other user not hearing them when muted).
- If the LiveKit server is not running or env not set, the "Live Jam" UI should be hidden or disabled so as not to confuse users. Or if shown and clicked, it should show a meaningful error ("Realtime server unavailable").
- The rest of session functionality (presence, uploads) still works as before alongside LiveKit.
- The solution should scale to at least ~5-10 users in a room without major issues (we can't easily test that many locally, but LiveKit is built for scale).
- Security: Only authorized users get tokens to join. If an unauthorized request is made to /api/livekit/token (e.g., no session cookie or wrong sessionId), it should be denied.

- All new code is covered by unit tests where possible, and there are no regressions in existing tests.

- **PR Plan:** Break down into roughly **3 PRs**:

- **2A: LiveKit Backend Integration** – Add the LiveKit token generation endpoint and related config (env vars, server keys) [213]. Possibly include a basic dev script for running LiveKit. Write tests for token issuance. This PR does not affect UI yet (except maybe a very minor stub).
- **2B: Live Jam UI** – Integrate LiveKit client in the session page with join/leave, audio streaming, and participant list UI [215] [214]. This will be a larger PR involving new React components and hooking into the SDK. Testing might include a dummy Room object for unit tests.
- **2C: Production Deployment Docs** – Add the `deploy/livekit` documentation and any config files needed for production use (docker compose templates, etc.) [212] [219]. Also final tweaks to error handling and feature flag gating.

Each PR will be kept as focused as possible. Feature-flagging the UI will allow merging 2A even if 2B is not done, without breaking anything (the UI remains hidden). - **Dependencies:** This depends on acquiring the LiveKit server binary or container. We add a dependency on the `livekit-client` JS library. Make sure to include it in `apps/studio-web/package.json`. We should also ensure the dev environment (perhaps via README) instructs how to install the LiveKit server. Possibly treat LiveKit as an external service like we do for Postgres/Redis (not part of our PNPM workspaces, just a binary to run). - **Migration Strategy:** No DB changes in this milestone (unless we decide to track something like "who is live now" in DB, which isn't necessary due to LiveKit's own tracking). So no migrations. - The introduction of LiveKit does not break existing usage; it's additive and hidden behind a flag. So existing users can continue without noticing anything if they don't enable it. - Rolling back would simply involve disabling the feature flag. - **Risks & Mitigations:** - *Complexity (WebRTC):* Working with live media and an external system (LiveKit) can be complex (issues of NAT, browser compatibility, echo, etc.). **Mitigation:** We leverage LiveKit's established solutions rather than building our own SFU. We'll use their recommended settings and test in a simple environment. The docs and community around LiveKit can help if we encounter issues. - *Dev environment burden:* Running another server (LiveKit) is needed. **Mitigation:** Provide clear instructions and perhaps a script to run it. Use dev mode to avoid needing TURN for local tests. For production, plan using their Docker which includes TURN. - *Latency / Quality:* If not configured right, audio quality could suffer or echo might happen. **Mitigation:** Use audio constraints (maybe disable echo cancellation if music is being played along) – but probably leave defaults which do echo cancel (since likely people will talk or sing). Document best practices (use headphones to avoid feedback). - *Security:* The token endpoint must not be misused to allow arbitrary room access or exhausting our system. **Mitigation:** Use short-lived tokens (LiveKit tokens by default might only last a few hours). Only allow joining specific room that matches session user has open. Rate-limit the endpoint if needed (though usage is low). - *Coordination with presence:* We have two parallel systems – our custom presence and LiveKit's internal presence (who's in the room). They might not always be in sync (e.g., a user could join LiveKit audio but not be in our presence if something went wrong, or vice versa). **Mitigation:** We can integrate them loosely by e.g., when a user joins LiveKit, we broadcast presence.join as well if needed, or rely on LiveKit's participant events for our UI. Since the Live Jam UI will show participants via LiveKit, and our presence indicator is already on session page (for general presence), we might end up with duplicate listings. We may hide or merge them: possibly treat LiveKit join as superseding presence. For now, we might keep them separate (presence list vs jam list) to avoid confusion, or integrate by using presence for text and LiveKit for audio. This is mostly a UX consideration; we'll clarify in UI or docs that Live Jam participant list is separate. - *Out of Scope Creep:* There's temptation to also add text chat or screen share. We explicitly defer those. **Mitigation:** Focus on audio only. If we have extra time, a basic text chat could be simple (we have presence WS to piggyback or we use LiveKit's data channels). But we likely skip it now to avoid delay.

Milestone 3: AI-Assisted Features (Tagging & Recommendations)

- **Goal:** Enhance Omnisonic with foundational **AI/ML capabilities** to auto-enrich the music library and provide smart recommendations. This includes automatic tag generation for tracks (genre/mood/instrument tags via audio analysis), computing track similarity (for "recommended tracks" or "similar songs" lists), and laying groundwork for future AI-driven features. This milestone helps users organize and discover music by leveraging AI, fulfilling the roadmap item of "genre fusion and recommendations" and making the library "smart."
- **Scope:**
- Create a new microservice or module for running AI analysis on tracks (could be called `ai-engine`).
- When a track is imported or on-demand, analyze the audio file to extract features: BPM (tempo), musical key, possibly a low-dimensional embedding (vector) for similarity, and high-level tags (e.g., genre, mood).
- Store the results in the database (extend Track model with fields like bpm, key, and maybe an `embedding` vector or a separate table).
- Provide a GraphQL API to get recommendations: e.g., `query similarTracks(trackId)` that returns a list of tracks that are similar to the given track [157].
- Optionally provide an interactive feature like a "genre fusion helper" which is more experimental (taking two tracks and suggesting how to mix them) [222] – this might be a small UI tool using the metadata we have (BPM, key) to compute suggestions (like "Track B is 5 BPM slower and 2 semitones lower than Track A; to mix, adjust accordingly").
- Keep AI purely optional/local: no calls to external AI APIs. If using any heavy ML models (like for embeddings), allow disabling via feature flag `ENABLE_AI` and ensure the app runs without them.
- Non-goal: We are not training new ML models here, just using existing libraries or small pretrained models for feature extraction (e.g., librosa via Python or some Node library for BPM detection, etc., or in the simplest case, use metadata / heuristics).
- **Key Tasks:**
- **AI Service Skeleton:** Set up `services/ai-engine` (if using Python, maybe use FastAPI; if Node, maybe a small Express server or extend our GraphQL?). Given we already have Python in the project (FastAPI ingest), we might lean on Python for AI (more libraries available). But to minimize new

components, we could attempt a Node solution if suitable libraries exist. Considering audio analysis, Python (with librosa or madmom, etc.) is powerful. Perhaps do Python FastAPI:

  - Endpoint: `POST /analyze` that accepts a track ID (or direct audio file URL) and returns analysis results [223] .
  - It would retrieve the audio file (e.g., the AI service could fetch from our storage using a signed URL or local path if shared volume).
  - Compute BPM, key, basic genre (maybe using a pre-made model or a very naive classifier based on tempo + instrument detection).
  - Return data like `{ bpm: 128, key: "C#m", tags: ["house","energetic"], embedding: [0.12, ...] }`.
  - We likely start with simple: BPM and key via a library (there are Python libraries for both; key detection could use e.g., librosa's chroma features; BPM detection from onset patterns).
  - For genre/mood, perhaps skip or do a simple guess (if we had a pre-trained model or an API, but since we avoid external, maybe not do genre classification now, or use some open dataset model).
  - Alternatively, integrate an existing small model like Spotify's Annoy indices or others – but keep it simple in v1 (maybe just tag "fast" if BPM>120, "slow" if <80, etc., as placeholder).
  - Important: **Feature flag** this whole pipeline with `ENABLE_AI` [223] . If off, nothing changes (the system doesn't auto-call analysis).
  - Also ensure this runs asynchronously, not blocking the main thread of Next.js or GraphQL.

- **Triggering Analysis:** Determine when to analyze a track:
  - On track import, after the track is saved, enqueue a job for analysis if AI is enabled [224] . This could be done by the Next.js API calling the AI service asynchronously (fire-and-forget) or by adding a job to a queue (maybe reuse BullMQ via Redis). We already have BullMQ for exports – we could add an "analysis" queue and run workers (maybe the ai-engine service itself could be a worker listening on Redis for tasks).
  - Simpler: Next.js API after saving track, makes an HTTP request to `ai-engine/analyze` for that track. The AI service responds when done (this could take a couple of seconds per track). This synchronous approach might slow the import response – better to decouple. So we prefer background: perhaps store a pending analysis record or just rely on the AI service to callback or update DB directly.
  - We can integrate AI service with DB too (give it DB access so it can update the Track row with results).
  - Possibly use Prisma in the AI service (it can import `@omnisonic/db` if we make the service Node, but if Python, we'd do raw DB writes or call a GraphQL mutation to update track).
  - Given complexity, maybe easiest: Next.js returns track immediately (so user sees it), then in background the analysis happens and updates the DB, and the UI could later show new tags/bpm when ready (maybe on page refresh or via a subscription).

- **DB & GraphQL Changes:** Add fields to `Track` for the analysis results:
  - Add `bpm` (Int, nullable), `key` (String, nullable), possibly `tags` (string[] or a separate Tag relation as earlier).
  - If we do vector similarity, we might add a column for `embedding` (vector). But Postgres can support a vector type via pgvector extension. Alternatively, store in a separate table or not at all (we can compute similarity by simpler heuristics in v1).
  - Extend GraphQL `Track` type to include `bpm`, `key`, `tags`.
  - Add `Query.similarTracks(trackId, limit)` [225] which returns a list of tracks. Implementation: find the track's metadata, then:

    - If we have embeddings and vector search: use a PG vector similarity or in-memory distance to find nearest tracks.
    - If not, use a heuristic: e.g., same genre or overlapping tags, plus BPM within ~5% and same or relative musical key (e.g., match or fifth).
    - Or even simpler: just list tracks sorted by difference in BPM + some string similarity on genre – enough to demonstrate concept.
    - If `ENABLE_AI` is off, `similarTracks` could still work using just basic metadata (so that feature can exist without actual ML).
- **UI Enhancements:**
    - Display any new metadata on the Track detail page: show BPM and Key if available, show tags (maybe as chips).
    - Possibly indicate if a track is analyzed or not yet (if analysis is asynchronous, maybe a small "Analyzing…" indicator if the BPM is null).
    - Add a "Similar Tracks" section on the track detail page listing a few tracks that the system thinks are similar [157]. This leverages the GraphQL query. The UI could just show a small list (track title + artist) and perhaps allow playing them or clicking to go to that track.
    - (Optional) Add a button "Analyze Track" on a track page, in case a track wasn't analyzed or to re-run analysis (perhaps for older tracks imported before enabling AI, or if user updates audio). This would call the AI service again.
    - *Genre Fusion Tool:* As a stretch goal, implement a simple UI where a user picks two tracks from their library and the system outputs a suggestion on how to transition between them [226]. This could be on the Library page or a separate route `/fusion`. For now, we might implement it as:
    - User selects Track A and Track B (dropdowns or autocomplete).
    - Backend (or frontend logic) compares BPM and key: if BPMs differ, calculate percentage change to match tempos; if keys differ, calculate the interval between them.
    - Output a suggestion like: "Speed up Track B by 4% and pitch up 2 semitones to blend with Track A" or "Tracks are in compatible keys (both Am) at similar tempo (~128 BPM), should mix well." This is not ML-based, just using known music theory.
    - This is a lower priority sub-feature but would be a neat demo of using our metadata.
- **Testing:**
    - Unit test the analysis logic if possible (e.g., give it a known audio file or signal to see if BPM detection roughly works).
    - If using Python, perhaps test it in isolation with a short audio clip.
    - GraphQL tests: ensure that after analysis, the Track fields are populated. Possibly simulate an analysis by writing to the DB and then calling `similarTracks` to verify output.
    - The similarTracks logic can be tested with some dummy tracks (e.g., one with BPM 120, another with 121, ensure they come out as similar).
    - E2E: It's hard to test audio analysis in E2E deterministically. But we could do a scenario: import a track, then (if analysis is quick) check a minute later that BPM shows up on the UI. Might be flaky. Alternatively, just test that similarTracks list appears (using known dummy metadata).

- **Documentation:**

    - `docs/features/ai-engine.md`: Explain how the AI service works, how to enable it, what it uses (mention if it uses any particular model or library) [227].

    - Document that AI features are optional and may require additional dependencies (like if using Python, need to install certain packages like `librosa`, or if Node, maybe need `sox` or other).
    - Update `docs/STATUS_GAPS.md` to mark AI as now addressed partially.
    - Possibly an ADR on "Local AI vs Cloud AI decision" if needed.

- **Acceptance Criteria:**

- After importing tracks, the system automatically adds enriched data for them: For example, within a short time (a few seconds to a minute), a track's BPM and key appear in the UI without user input. If a track's metadata had no genre, maybe the system assigned one or at least a mood tag.
- The "Similar Tracks" feature produces a reasonable list. If a user views a rock song, the recommendations should be other rock songs or songs with similar tempo/key from their library (not random). Since our logic might be simple, we define "reasonable" loosely: at least it's not listing completely dissimilar tracks. We can test with known differences: if we have one EDM track and many rock tracks, searching similar for the EDM should ideally bring EDM (if any) or none. Our baseline could be BPM-based grouping, which should yield some clustering.
- The AI processing should not crash the app nor block user actions. If `ENABLE_AI=false`, everything still works (just no BPM displayed, etc.).
- Security: the AI service, if separate, should only be accessible internally (we might not expose it publicly at all, or protect it). If an external request is needed, ensure auth or secret.
- Performance: analyzing one track should be reasonably fast (preferably under ~5 seconds for a 3-minute song on modest hardware). If it's slower, ensure it's async so user isn't stuck. The system should handle analyzing multiple tracks sequentially or concurrently without issues (maybe queue them).
- The feature flag works: turning off AI prevents any analysis or AI suggestions.

- The genre fusion demo (if implemented) gives logically correct suggestions given two tracks' metadata, though it's an experimental feature so exact output isn't critical.

- **PR Plan:** Possibly **2-3 PRs**:

- **3A: AI Engine service and Track analysis fields** – Add the new service skeleton (e.g., a FastAPI with a dummy `/analyze` that returns fixed data initially), add Track.bpm/key/tags fields in the DB and code. Ensure the pipeline from import to analysis trigger is wired (maybe initially call analysis with a stub to verify the flow). Feature flag gating. [228]
- **3B: Implement analysis logic and integrate results** – Actually implement BPM/key detection, etc., in the AI service. Update the Track records with real results. Extend GraphQL queries for similarTracks, and implement the recommendation logic [157]. Possibly also add the Saved Search feature or an extension to search by these new fields (e.g., allow filter by BPM range).

- **3C: Frontend UI for recommendations and fusion** – Update UI to display BPM/key/tags on tracks, add Similar Tracks section, and add the Fusion tool if doing it [222]. This final PR focuses on user-facing changes once the backend is solid. Include documentation updates.

- **Dependencies:** We'll need certain libraries:

- If Python AI: `librosa` (for audio analysis), which in turn needs `numPy`, etc. Possibly `essentia` or `madmom` for BPM if we want accuracy. Installing these might require system packages (e.g., FFmpeg for reading audio). Alternatively, use `pydub` with FFT for a crude BPM. We'll document any needed packages (maybe add to `requirements.txt` in the service).
- If Node AI: there are some Node packages for BPM like `music-tempo` or using Web Audio API offline – but Node might be less accurate. Could try `fft-bpm-detector`. Node for key detection is even rarer. So likely Python is better.
- Another path: since we already produce a waveform for exports via FFmpeg, we could have FFmpeg compute RMS or spectrum for BPM – but complex.
- We might use the `@tensorflow/tfjs` if there's a small model for audio classification we can run in Node, but that can be heavy.
- So likely dependency: Python with certain libs.
- Also Postgres `pgvector` extension if we attempt vector similarity in DB. If our hosting environment allows, we could enable it in a migration. Or we do vector similarity in memory in Node/Python (for library sizes likely okay).
- Ensure adding these doesn't break CI (we might skip running AI in CI or mark tests accordingly).
- **Migration Strategy:** Add new nullable columns for BPM, key, etc. This does not break anything existing – old tracks just have null (and UI can hide or show as blank). The code should handle nulls (e.g., if BPM unknown, maybe show "–").
- If we later computed for existing tracks, those will fill in.
- There is no removal of fields, only additions, so safe to apply.
- If enabling pgvector, that's a database extension installation; might require a manual step on the DB (we can attempt via migration but needs superuser). If that's troublesome, skip vector usage.
- **Risks & Mitigations:**
- *Accuracy of AI:* Our simple approach might mis-tag or mis-estimate BPM. **Mitigation:** We set user expectations that it's a "beta" feature. They can correct metadata themselves if needed. Also choose robust libraries – e.g., librosa's BPM detection is decent if parameters tuned, but not perfect. Possibly allow user to override BPM if wrong.
- *Performance:* Running analysis on many tracks or long tracks could be slow or CPU intensive. **Mitigation:** We could limit concurrency (one analysis at a time). Use short audio snippet for analysis (e.g., take first 30 seconds for BPM which often suffices, or downmix to mono low sample rate for speed). If extremely needed, allow analysis to be cancelled or skipped for large files.
- *Integration complexity:* Adding a new service (AI) in Python adds complexity in deployment and dev environment. **Mitigation:** We containerize it or allow it to be optional. For dev, if user doesn't want to set up Python, they can keep `ENABLE_AI=false`. We document steps to get it running (like `pip install -r requirements.txt && uvicorn`). Possibly in CI we won't run it fully (just basic tests).
- *Data volume:* If we store audio embeddings, those can be large vectors. But we likely won't at scale or can limit dimension (like 32-D or 128-D). Not a big issue for now.
- *Scope creep:* We must be careful to implement core features (BPM, key, similar tracks) first and not dive too deep into perfecting a genre classifier or training models. **Mitigation:** Use simple heuristics and existing known values. For example, we might initially skip genre classification entirely (just leave it for user or future ML model).
- *Legal/Contention:* If using any pretrained model, ensure license is compatible. We likely stick to MIT/BSD libraries and avoid big proprietary ones.

Milestone 4: Scheduling & Automation

- **Goal:** Introduce a basic scheduling system to automate actions in Omnisonic – e.g., scheduling a playlist to play at a certain time or sending reminders to start a session. This brings a new dimension of utility, allowing Omnisonic to act as a "set it and forget it" music planner (for DJs or home audio) and to demonstrate automation capabilities. It addresses the concept of time-based triggers which was not present yet.
- **Scope:**
- Implement a scheduler service or background job that can execute tasks at scheduled times. This could be built on top of BullMQ (since we have Redis) using delayed jobs, or use a Node cron library.
- Define a **Schedule** model in DB: with fields `id`, `userId`, `actionType` (play_playlist, open_session, etc.), `actionData` (e.g., playlistId or sessionId), `scheduledFor` datetime, `status` (pending/executed) [229].
- Provide UI for users to create and view schedules. For example: "Play playlist X at 7:00 AM daily" might be a future feature, but we can start with one-off scheduling: pick a date/time and an action.
- Implement execution of at least one type of action: the simplest is maybe to queue up a playlist to play at a time (assuming the app is open/running), or to send a notification. Given this is tricky (if user's browser isn't open, nothing happens, unless we integrate some push notification or email – which is beyond scope likely). Alternatively, an action could be to start a mixdown export at a time or some back-end action.
- Perhaps focus on something self-contained: e.g., schedule an export for off-peak hours or schedule a session to auto-close or archive at a time (just examples).
- Or simpler, treat it as a reminder system: at scheduled time, create a notification or email (but email requires SMTP integration, which we do have placeholders for) [230].
- Possibly tie with Alerts: but alerts are more about data thresholds, whereas scheduling is user-defined.
- Non-goal: We won't implement complex recurrence (maybe allow daily/weekly if easy, but can skip recurrence for now except maybe by allowing user to create multiple schedules).
- **Key Tasks:**
- **Schedule Model & API:** Add `Schedule` model as described [229]. Fields: if we want recurrence, add `cronExpr` or `repeatInterval`, but likely skip for now. Use simple one-time `DateTime` for `scheduledAt`.
  - GraphQL: perhaps add mutation `createSchedule(actionType, actionData, scheduledFor)` and query `schedules` for user's upcoming schedules.
  - Or REST: `POST /api/schedule` to create, etc.
- **Scheduler Worker:** Could create a new service `services/scheduler` or integrate into an existing worker. E.g., we could augment `upload-cleaner` or create `@omnisonic/scheduler` worker that:
  - Periodically (every minute) checks for due schedules (WHERE scheduledAt <= now AND status = pending) and triggers them.
  - Or uses BullMQ delayed jobs: when schedule is created, enqueue a job delayed until that time to execute action. That might be easiest: use Redis delayed jobs; on job run, perform the action then mark schedule executed.
  - The action execution: if "play playlist", how to do that server-side? If user's app isn't open, we can't actually play sound. Perhaps the action is to send a notification or set a flag so that next time user opens the app, it auto-plays? We might simplify: implement a stub where at

scheduled time, we log “Playlist X would play now” and update schedule status, maybe also create an AlertEvent or notification entry.
  - If “open session”, maybe it means alert people to join a session at that time. We could integrate with Alerts service to send an email invite. Actually, we have SMTP configured for alerts. Perhaps piggyback: schedule could send an email “It’s time for your session jam”.
  - Considering time, perhaps easiest meaningful action: send an email reminder. So a schedule could be: at time T, send email to user (or others) with a message “Reminder to practice guitar” or “Playlist X is scheduled to play (please open Omnisonic to play it)”. This leverages the Alerts system or a simple SMTP sending. We have SMTP env placeholders [230] we can use.
  - Implement minimal email via nodemailer or similar if SMTP configured.
- **UI for scheduling:**
  - Maybe on a playlist page, have a “Schedule Play” button that opens a form to pick date/time (and perhaps recurrence or none).
  - Or a general “Schedules” page where user chooses an action from a dropdown (e.g. “Play a playlist” or “Open a session”) and fills details and time.
  - For simplicity, focus on scheduling playlist playback, as that’s straightforward conceptually (everyone understands scheduling music).
  - Use a datetime picker component for selecting time.
  - On submitting, call createSchedule API. Then show the new schedule in a list of upcoming schedules, with ability to cancel (delete).
  - If we can auto-play, we might require the web app to be open at that time and listen for an event (could use WebSocket to notify if user is online). That’s complex, skip auto-play. Instead, maybe we pop a browser notification (if user allowed notifications). Web push is also complex to implement from scratch. So again, likely rely on email or just have it in-app if open.
- **Integration with Alerts (optional):** We might utilize the Alerts microservice concept:
  - That system is more about threshold triggers, but we could cheat by inserting a rule that triggers at time (but it’s not built for time triggers, it's built for counts).
  - Instead, maybe use it to deliver notifications: For sending email, we can use the Alerts service if it had an API. The Phase 3 spec says Alerts service polls ClickHouse; but also has an API for channels and rules [140]. Could piggyback by creating an AlertEvent. This might be overkill. Simpler: directly send email from scheduler worker.
- **Testing:**
  - Unit test scheduling logic (e.g., if we use delayed jobs, ensure a job scheduled for +5sec runs and calls the action function).
  - Possibly simulate a quick schedule (like 1 minute ahead) in a test and ensure it flips status to executed.
  - UI: perhaps test that creating a schedule returns success and that the schedule list shows it.
  - We might not easily test actual email sending in CI; we could abstract email sender to a mock in tests.
- **Docs:**
  - Document how scheduling works and any limitations (e.g., "Omnisonic must be running to execute scheduled tasks" or "emails require SMTP configured").
  - Update user guide to mention how to use scheduling feature (maybe in an “Automation” section).
  - If needed, instruct admin to configure SMTP in env for production to enable email notifications.

- **Acceptance Criteria:**

- A user can schedule an action (e.g., playlist playback reminder) at a chosen time via the UI, and at that time the system executes the action (for instance, user receives an email or sees a notification that the scheduled event occurred).
- The schedule persists in DB and is marked done after execution. The user can view their upcoming and past schedules.
- If a schedule is set in the next few minutes and the app is running, we can observe it trigger (maybe a console log or something for dev).
- If SMTP is configured and action is email, an email is indeed sent to the user's address (we can test with a dummy SMTP in dev).
- Security: Users can only create schedules for themselves (we ensure `userId` is taken from session, not from client input).
- The scheduler does not consume excessive resources – e.g., it wakes up periodically or uses delayed jobs efficiently.
- Feature doesn't break anything else (since it's mostly additive).

- (If not fully functional like auto-play, we document those limitations clearly to manage expectations.)

- **PR Plan:** Possibly 2 PRs:

- **4A: Backend scheduling system** – Add Schedule model, GraphQL/REST endpoints, and scheduler worker (using BullMQ or cron) [231] . Possibly integrate email sending. Basic tests.

- **4B: Scheduling UI** – Add front-end components for creating and listing schedules. Tie into backend. Tests and docs.

- **Dependencies:** Possibly nodemailer for email, or use existing alerts (which might need SMTP config, which we have env for). Also a date-time picker UI component (could use a lightweight one or just input type datetime-local).

- **Migration:** New Schedule table – straightforward, doesn't conflict with others.
- **Risks:**
- *Usefulness:* If the app can't actually auto-play music at the time (due to browser limitations), the feature might feel underwhelming. **Mitigation:** Focus narrative as a “reminder” or “notification” feature, not literal auto-play (unless we get creative with a service worker for web push – out of scope).
- *Email reliability:* If SMTP is not set, scheduling feature might do nothing visible. **Mitigation:** Hide or warn if no notification mechanism. Perhaps we say scheduling requires enabling email in config.
- *Time zones:* Need to clarify schedule times (store in UTC, display in local time). Use user's local time for input but convert to UTC for storing.
- *Complexity:* Cron scheduling for repeated events can be complex. We likely skip recurrence for now to avoid that risk. Single event scheduling is simpler.
- *Collision:* What if two events scheduled same time? Should be fine, both will trigger (if using jobs, they queue).

Milestone 5: Admin & Monitoring Tools

- **Goal:** Provide administrative and monitoring capabilities for Omnisonic to support multi-user scenarios and production maintenance. This includes an **Admin UI** for managing users and content, and enhanced monitoring dashboards for system status and usage analytics. This milestone ensures that as the system grows, administrators can oversee it and troubleshoot issues, addressing the last major gap (lack of admin tools) identified in the project status.
- **Scope:**
- **Admin Web UI:** A protected section of the web app (or a separate app) that only admin users can access. It will allow viewing all users, perhaps editing/deleting users, viewing all sessions or tracks in the system, and removing inappropriate content if needed.
- **Admin Roles/Auth:** We may introduce a simple authorization mechanism to designate certain user(s) as admin. Perhaps an env list of admin emails or a flag in the User model (we'd add `isAdmin` boolean).
- **Monitoring Dashboard:** Could be basic at first – e.g., a page showing system metrics like number of active users, total tracks, storage used, etc. Possibly graphs of usage over time. We have ClickHouse for analytics; we could use it to track usage events. Alternatively, tie into existing traces/logs to display errors or performance.
- Possibly integrate with the Alerts service for system alerts (like if a service goes down or high load – but that might be too much).
- **Security audit improvements:** Add features like audit log viewing, perhaps log of who did what admin action.
- Non-goal: Full APM or external monitoring integration (that's ops side), although we might add something like a /healthz endpoint and instructions for Kubernetes liveness etc.
- **Key Tasks:**
- **Admin Privileges:** Add an `isAdmin` field to User model or maintain a list of admin emails in config. For simplicity, maybe use an env var `ADMIN_EMAILS` that the code checks. Or better, do a migration to add `role` field to User (e.g., enum UserRole { USER, ADMIN }) default to USER. Mark ourselves or a test user as ADMIN in dev for testing.
    - Update NextAuth session callback to include this info (like `session.user.role`). NextAuth allows adding custom fields.
    - Add an authorization check middleware for admin pages (like Next.js middleware or simple runtime check in getServerSideProps) to block non-admin.
- **Admin UI - Users:** Create a page `/admin/users` listing all users [169]. Use server-side data fetching to get user list (via Prisma or GraphQL if we expose it). Columns: name, email, perhaps # of sessions, # of tracks. Possibly a button to toggle admin or to disable a user.
    - If implementing "disable user", we'd need a field (like `disabled: boolean`). That might be too deep, skip unless needed.
    - Provide search on users by email/name if many.
    - Provide detail view for a user (or modal) showing their sessions and tracks. If needed, allow admin to delete a user's track or session (for moderation).
- **Admin UI - Content:** Pages to view all tracks (library tracks) and all sessions in the system. Admin can filter or search them. Possibly integrated into user detail (like see tracks for a specific user).
    - For tracks: maybe `/admin/tracks` shows a table of tracks with columns: title, owner, duration, etc. Admin could click to play (for moderation listening) and possibly delete track if it's violating something.

    - For sessions: `/admin/sessions` with columns: name, owner, createdAt, maybe #uploads. Could allow deletion or transferring ownership.
- **Monitoring Dashboard:**
    - This might be a single page `/admin/metrics` or integrated into an admin home. Show key stats: total users, total tracks, total storage used (we can sum file sizes from Upload table), total plays (if we track, maybe not yet), etc.
    - If possible, show a graph of active users or uploads over time. If we logged these events in ClickHouse or even Postgres, we can fetch count by day.
    - We might use ClickHouse data if Phase 3 had some user event tracking (not explicit, mostly news ingestion). Alternatively, we could instrument usage: e.g., log an event when a track is played or when user logs in, to a new table (but that's a bit late in game to implement fully).
    - Maybe just show static counts and leave advanced analytics for later.
- **System Health Monitoring:** Possibly add a simple `/admin/health` page that shows status of each service (GraphAPI, Ingest, etc.). Could call each service's health endpoint (GraphAPI and Ingest may have `/healthz` returning ok [9]). Summarize results (like a green check if ok).
    - If environment includes something like Grafana, we could embed a frame or link to it, but likely not.
- **Testing:**
    - Test that non-admin cannot access admin routes (simulate by calling the page API as a regular user and expect redirect or 403).
    - Test that admin listing works (maybe create a dummy user and see it listed).
    - If admin deletes a track via admin UI, check that it's removed from DB and storage (we should reuse our deletion logic).
    - Basic test for metrics calculations functions (like if we have a function to sum storage, test with sample data).

- **Documentation:**

    - Document how to designate an admin (via env or DB). Emphasize to secure admin credentials.
    - Explain admin features available.
    - Possibly update `SECURITY.md` if any to reflect admin capabilities.

- **Acceptance Criteria:**

- The system can distinguish admin users. For example, if we mark a user as admin, that user sees an "Admin" section in the app while others do not.
- Admin user can view a list of all user accounts and all content. They can perform administrative actions such as promoting a user to admin (if implemented), or deleting a user's content. For instance, admin can remove an offensive track from someone's library; the track record is deleted and file removed from storage.
- Basic stats are visible: admin sees current counts of users/tracks/sessions and maybe a rough graph of growth (if data available).
- Admin can check if services are up (maybe by seeing health indicators).
- The introduction of admin features does not alter normal user experience except possibly adding `role` in session JSON which is benign. Regular users should not accidentally access admin pages (our checks work).

- All new functionality is behind proper auth checks to prevent escalation (e.g., no one can call an admin API without being admin).
- Monitoring: if a service is down (simulate by stopping one), the admin health page should show it as down (if we implement health checks).
- No sensitive data is exposed to non-admins.

- **PR Plan:** Possibly 2 PRs:

- **5A: Admin backend & role support** – Add user role field, admin check middleware, GraphQL or REST endpoints for listing users/tracks (or reuse Prisma directly in next for simplicity), and health check endpoints. Make sure admin pages are gated [169] . Some initial UI could be scaffolded.
- **5B: Admin UI & metrics** – Implement the actual UI components (tables, etc.), link in navigation for admins, and metrics calculations. Finalize tests and docs.

- **Dependencies:** Maybe a UI table library for convenience, or just use simple HTML tables with Tailwind. Possibly use an icon or library for status indicators.

- **Migration:** Add `role` to User – existing users default to USER. If using env for admin, no migration needed but that's less flexible. Probably best to migrate.
- **Risks:**
- *Security risk:* If we make a mistake in admin check, could expose data. **Mitigation:** Keep it simple – e.g., an environment-defined admin list is straightforward but less dynamic; a DB field requires careful control (ensuring only an admin can change roles).
- *Data volume:* Listing all tracks/users could be heavy if thousands. We can paginate the list. For now, assume manageable numbers.
- *Completeness:* True admin needs more (like ban user, view logs). We implement only core needs due to time. Document that.
- *Scope:* Admin features could balloon – we will implement minimal viable admin capabilities.

Note: Milestones 3, 4, 5 can be re-ordered or combined depending on priorities (e.g., if AI is less urgent than scheduling). The above ordering is based on given roadmap, but can be adjusted by the team. Each milestone is designed to be delivered independently, with feature flags to isolate them if needed.

The roadmap ensures that after Milestone 2, Omnisonic has a solid foundation of user-facing features (library and real-time jamming) and then moves into more innovative/advanced territory (AI and automation), finally rounding out with admin/ops improvements for a production-ready system. Progress should be reviewed at each milestone and plans adjusted based on user feedback and technical findings.

Codex PR Prompts

Below are detailed prompts for each planned Pull Request, to guide an AI pair programmer (like GitHub Copilot or OpenAI Codex) in implementing the features. Each prompt includes the PR title, the goal, key files to examine, specific tasks, tests, documentation updates, telemetry points, verification steps, and what is explicitly out of scope for that PR. These prompts ensure focused, small PRs that collectively achieve the milestone goals.

PR 0A: "Repo Map Documentation and Baseline Plan"

- **Goal:** Compile a comprehensive overview of the Omnisonic repository structure and write down the implementation game plan for the upcoming Music Library features. This PR establishes shared understanding and adds no functional code, only documentation (and possibly CI config). It should map out all apps/services, how to run them, and outline how we'll extend the schema for the library in Milestone 1.
- **Files to Inspect First:**
- `README.md` – Check for any existing repository structure info and ensure consistency [1] [232].
- `docs/specs/phase-1-studio.md` – See what was planned or done in Phase 1, to incorporate into repo map [144] [233].
- `packages/db/prisma/schema.prisma` – Identify models related to tracks/sessions (User, StudioSession, Upload, Recording, etc.) [19] [20].
- `apps/studio-web/app/api/upload/route.ts` – Understand upload flow as part of data flow diagram [64] [67].
- `services/graph-api/src/index.ts` – Note GraphQL types like Work, Recording, and any track-like concept [111] [112].
- **Implementation Tasks:**
- Write `docs/REPO_MAP.md` documenting each part of the monorepo (apps, services, packages) with their purpose and paths [1]. Include how to run the whole stack (PNPM commands, env vars, Docker compose) [40]. Document authentication model (NextAuth, user table) with references to code [48]. Document the storage pipeline (upload to storage to DB to download) with a flow diagram [64] [38].
- In `docs/REPO_MAP.md`, include a **data flow diagram** (can use Mermaid sequence diagram) for "upload -> storage -> DB -> UI playback" [89] [92], based on the upload code.
- Create `docs/plans/M1_LIBRARY_PLAN.md` outlining the design for Music Library (Milestone 1). This should cover proposed database changes (e.g., new Track model, Playlist model) [181], GraphQL additions (queries/mutations for tracks & playlists) [185], Next.js pages to be added (`/library`, etc.) [194], and how we'll ensure backward compatibility (session uploads remain working) [188]. Essentially, this is a technical spec for milestone 1 implementation.
- Add a "Definition of Done" checklist in the plan for each sub-PR of Milestone 1 [200]. Break Milestone 1 into PRs 1A, 1B, 1C, 1D as identified in the roadmap, and list acceptance criteria for each (e.g., for 1A: migration runs, new Track model visible in Prisma client, etc.).
- Ensure no actual code changes besides docs. However, update CI config if needed (for instance, if no CI pipeline yet, include a basic GitHub Actions workflow from earlier templates to lint/test) – based on earlier roadmap mention [175]. If adding CI, also add a badge or note in README.
- Optionally, create a docs/decisions/ directory and add an ADR for "Monorepo vs Polyrepo" if not present (ADR 0001 might exist; check `docs/adr/` for monorepo structure). If it exists, just reference it in Repo Map.
- **Tests to Add:** No code, so no automated tests. Instead, perform a **content review**: ensure all file paths and references in docs are correct. Maybe add a dummy test in `packages/db` to verify Prisma generate works (not necessary, but CI should run `pnpm db:generate` to catch schema errors).

- **Docs to Update:** This PR is all docs. Specifically creates/upates:
- `docs/REPO_MAP.md` (new),
- `docs/plans/M1_LIBRARY_PLAN.md` (new),
- possibly updates `README.md` if needed (e.g., add reference to new docs or fix outdated info),
- possibly an ADR in `docs/adr/` if needed.
- **Telemetry:** Not applicable (no runtime code). However, could add a stub idea: e.g., mention in plan that in future we might add telemetry events for track plays. No actual telemetry to emit now.
- **Verify Locally:**
- Read through the new docs in a markdown viewer to ensure formatting is clear (headings, lists).
- Check that all cited file paths exist and line references are accurate (open those files to cross-check content).
- Run `pnpm install && pnpm build` to ensure no build breaks (since no code changes, should be fine).
- If CI config added, run `pnpm lint` and `pnpm test` locally to ensure they pass.
- **Definition of Done Checklist:**
- [x] `docs/REPO_MAP.md` explains all major repo sections with code references.
- [x] `docs/plans/M1_LIBRARY_PLAN.md` details schema changes, API additions, UI plans and testing strategy.
- [x] Data flow diagram for upload->playback included and accurate.
- [x] CI pipeline added or confirmed (lint/test) – all green.
- [x] No functional changes to app behavior.
- [x] Team has reviewed docs and agrees on Milestone 1 approach (this item for human confirmation).
- **Out of Scope:** Do not implement any actual library features here – **no model creation, no pages or API yet** [234] . Do not modify database or code; just planning and documentation. Also, don't include any secret credentials or lengthy environment configurations beyond what's needed in docs. This PR is purely informational and planning.

PR 1A: "Track & Playlist Models, GraphQL Schema"

- **Goal:** Introduce the new **Track** and **Playlist** data models and basic GraphQL CRUD operations for the music library. This PR focuses on the backend schema: updating the Prisma schema with new tables and relationships, generating migration, and extending the GraphQL API with types and resolvers for library tracks and playlists. No front-end changes here, just backend readiness for library data.
- **Files to Inspect First:**
- `packages/db/prisma/schema.prisma` – Review existing models (User, StudioSession, Upload, Recording) to decide where to integrate Track vs reuse Recording [19] [96] .
- `services/graph-api/src/index.ts` – See how current types and resolvers are defined to follow similar patterns (e.g., how Query.work and Mutation.upsertRecording are structured) [117] [118] .
- `apps/studio-web/lib/auth.ts` – Note how user IDs are used, possibly to set default userId in new models (though likely straightforward) [48] .
- `packages/db/src/client.ts` – Confirm that after schema update, running `pnpm db:generate` will expose new Prisma client fields.
- (Optional) `docs/plans/M1_LIBRARY_PLAN.md` – Use the design outlined for fields in Track and Playlist as guidance (the plan from PR 0A) – e.g., fields like title, artist, etc., and relationships.
- **Implementation Tasks:**

- **Extend Prisma Schema:**
    - Add a new model **Track** in `schema.prisma` with fields:
    - `id String @id @default(uuid())`
    - `userId String` (owner) with relation to User [19] ,
    - `title String`,
    - `artist String?`,
    - `album String?`,
    - `year Int?`,
    - `genre String?`,
    - `durationMs Int?`,
    - `bpm Int?`,
    - `key String?` (musical key),
    - `artworkUrl String?`,
    - `source TrackSource` (an enum you'll define e.g. IMPORT, SESSION, RECORDING, OTHER) [235] ,
    - `createdAt DateTime @default(now())`,
    - `updatedAt DateTime @updatedAt`.
    - Possibly `fileId String?` if linking to Upload; but better approach: one-to-one relation with Upload (so add `uploadId String?` and `upload Upload? @relation(...)`). We might want to keep Upload for actual storage info and Track for metadata. Decide and implement accordingly (likely relation Track.upload -> Upload).
    - Add index on userId for quick filtering by owner.
    - Use @@map or naming conventions similar to others if needed.
    - Add `TrackSource` enum in schema (values: IMPORT, SESSION_STEM, RECORDING, OTHER) [235] .
    - Add model **Playlist** with:
    - `id String @id @default(uuid())`,
    - `userId String` (owner, relation to User),
    - `name String`,
    - `createdAt DateTime @default(now())`,
    - possibly `isPublic Boolean @default(false)` if we foresee sharing (optional, can omit now).
    - Add model **PlaylistItem** (join table for tracks in playlist):
    - `playlistId String`,
    - `trackId String`,
    - `position Int`,
    - primary key on (playlistId, trackId) or an `id` if easier,
    - relations: `playlist -> Playlist` and `track -> Track`, with onDelete cascade on both.
    - This allows ordering tracks in a playlist.
    - Ensure to relate these: e.g., `Playlist.items PlaylistItem[]` and `Track.playlistItems PlaylistItem[]`.
    - Add relevant indexes (playlistId, trackId combination unique or with position).
    - Update `User` model relations: add `tracks Track[]` and `playlists Playlist[]` for convenience (like how User has sessions, uploads) [236] .
    - Leave existing models untouched to avoid breaking anything (we're adding new ones only).
    - Run `npx prisma format` if needed to keep formatting.

- **Run Migration:** Create a new Prisma migration (`pnpm db:migrate` which prompts for name, e.g., "add_track_playlist") and include it in version control. Verify the generated SQL looks correct (new tables, FKs).
- **Update GraphQL Schema (Yoga):** In `services/graph-api/src/index.ts`:
  - Define new GraphQL types corresponding to Track, Playlist, PlaylistItem:
  - `type Track { id, title, artist, album, year, genre, durationMs, bpm, key, artworkUrl, source, createdAt, updatedAt, owner: User! }`. Include owner if needed (or at least ownerUserId). We might also include a field `storageUrl` if we want GraphQL to directly expose where to stream the track; but storageUrl is in Upload. Perhaps we'll have Track resolver fetch its Upload to get a URL. For now, omit or add `fileUrl: String` which we resolve from upload if needed.
  - `enum TrackSource { IMPORT, SESSION_STEM, RECORDING, OTHER }`.
  - `type Playlist { id, name, createdAt, tracks: [Track!]! }` – we can resolve tracks via PlaylistItems join.
  - We might not expose PlaylistItem as a GraphQL type publicly; instead, playlist.tracks can be derived.
  - Possibly a `type PlaylistItem { track: Track!, position: Int! }` if ordering needed.
  - Add Query fields:
  - `libraryTracks(query: String, filter: TrackFilterInput, limit: Int, offset: Int): [Track!]!` – This will be used for search; but in this PR, implement a basic version (just return all tracks for the current user for now, ignoring search logic until PR 1D). We will parse `query` and maybe do a title/artist contains if provided (simple).
  - `libraryTrack(id: ID!): Track` – fetch one track (ensure user owns it).
  - `playlists: [Playlist!]!` – current user's playlists.
  - `playlist(id: ID!): Playlist`.
  - Add Mutation fields:
  - `updateTrackMetadata(id: ID!, input: TrackInput!): Track` – to edit title/artist/etc. [187].
  - `createPlaylist(name: String!): Playlist` [187].
  - `addTrackToPlaylist(playlistId: ID!, trackId: ID!, position: Int): Playlist` – returns the updated playlist or something.
  - `removeTrackFromPlaylist(playlistId: ID!, trackId: ID!): Playlist`.
  - `reorderPlaylistItems(playlistId: ID!, trackIds: [ID!]!): Playlist` – allows sending a new order.
  - `deletePlaylist(id: ID!): Boolean` (or Playlist). (We might not implement all in this PR fully, but at least stub them out.)
  - Define input types if needed: e.g., `input TrackInput { title: String, artist: String, album: String, year: Int, genre: String, bpm: Int, key: String }`. This will be used in updateTrackMetadata.
  - For filters: maybe define `input TrackFilterInput { source: TrackSource, genre: String }` etc., to use later; but could skip detailed filter now for simplicity.
  - Implement Resolvers:
  - For Query.libraryTracks: use Prisma `prisma.track.findMany` filtering by `userId = context.user.id` (the currently authenticated user). If `query` arg provided, filter title OR artist ILIKE `%query%` (we can use Prisma contains or mode insensitive) – or leave the filtering for PR 1D and just return all for now.

    - For libraryTrack(id): `prisma.track.findUnique` (or findFirst with userId condition) to ensure ownership.
    - For playlists: `prisma.playlist.findMany({ where: { userId: ctx.user.id }})` including items/tracks via relations.
    - For playlist(id): similar find by id and userId.
    - For Mutation.updateTrackMetadata: First, find track ensure it belongs to user. Then `prisma.track.update({ data: inputFields, where: { id } })`. If input has fields undefined, ensure to not overwrite existing (so perhaps we only include provided fields). Prisma can accept partial data. Return the updated track.
    - For createPlaylist: `prisma.playlist.create({ data: { name, userId: ctx.user.id } })`.
    - For addTrackToPlaylist: We need to create a PlaylistItem. Could do `prisma.playlistItem.create({ data: { playlistId, trackId, position } })`. But ensure the track and playlist belong to this user (validate by querying playlist.userId and track.userId). Return updated playlist with tracks.
    - For removeTrackFromPlaylist: `prisma.playlistItem.delete` or deleteMany (since PK is composite) for that playlistId & trackId. Return something (maybe the playlist).
    - For reorderPlaylistItems: loop through provided trackIds and update PlaylistItem.position accordingly (this might be multiple updates). Alternatively, delete all PlaylistItems for that playlist and re-create in new order; but better just update positions.
    - For deletePlaylist: `prisma.playlist.delete` (which should cascade PlaylistItems if foreign key onDelete cascade). Or deleteMany items then playlist. Return true/false.
    - We should also consider cascade delete of Track if user deletes a track (not in scope of this PR perhaps, but maybe add a mutation for deleteTrack as well).
    - Hook these resolvers into the schema creation (the Yoga server probably has an object with Query, Mutation resolvers defined; extend those).
    - Include relations in resolvers if needed (e.g., Playlist.tracks: implement it by fetching PlaylistItems join or via Prisma include).
    - Possibly easier: In Query for playlist, do `include: { items: { include: { track: true }, orderBy: { position: 'asc' } } }` then return playlist with an `items` array. But our GraphQL type was tracks, not items, so we can map items to tracks in resolver.
    - Or define Playlist.items field of type [PlaylistItem], which then has a resolver to fetch track. But that's more work on client. Simpler for client: Playlist.tracks directly. Implement resolver for Playlist.tracks: do a findMany PlaylistItem for that playlistId include track, return just track list. That keeps GraphQL schema simpler for consumption.
    - Add authorization checks in resolvers: ensure the current user's ID matches the track's owner or playlist's owner for all modifications and queries [188]. Throw GraphQLError if not authorized.
- **Prisma Client Generation:** Run `pnpm db:generate` to update the Prisma client TypeScript with new models. Ensure no type errors in code after updating resolvers (use the new `prisma.track`, `prisma.playlist`, etc.).
- **Update GraphQL Context/Auth:** If needed, ensure that we pass user info (id) into GraphQL Yoga context. Possibly check how `auth()` is used in Next API vs Graph API. The Graph API might not have built-in auth middleware. Perhaps in this scenario, we restrict GraphQL usage to logged-in clients by requiring a Bearer token or relying on the Studio app to proxy with session cookie. If context.user is not present, all our resolvers should handle that by throwing unauthorized. We'll assume context provides userId via some means (to simulate, we might extract from an

Authorization header or integrate with NextAuth JWT). For now, possibly add a placeholder in Yoga startup: if using NextAuth JWT, maybe the GraphQL requests include it. This detail might be out-of-scope for now, but mention it or implement minimal: e.g., read `Authorization: Bearer <token>` header, verify it with NextAuth secret to get user. This could be complex, and since GraphAPI might currently be open or expecting local dev usage, maybe skip heavy auth integration but still protect our new resolvers by expecting a userId in context.

- **Tests to Add:**
- Add a **unit test** in `services/graph-api` for new resolvers if possible (we may simulate context with a fake user and use an in-memory SQLite). If testing GraphQL in-memory is too involved, at least do some simple invocation:
    - Test that Query.libraryTracks returns only tracks for the user. Prepare a few tracks in DB (via Prisma) for two different users, ensure query returns correct ones.
    - Test that createPlaylist creates a playlist and that addTrackToPlaylist then can be queried.
    - These tests might operate at the Prisma level with direct calls to resolvers.
- Alternatively, write a smaller scope **integration test**: e.g., using Prisma client directly to verify relationships:
    - After migration, insert a Track and PlaylistItem via Prisma and ensure relations link (like track.userId and playlist.userId).
    - Ensure cascade behavior: if playlist deleted, playlistitems removed (for manual check).
- If time, a test for updateTrackMetadata: create a track, call resolver to update it, verify fields changed.
- Ensure to run `pnpm test` and that all existing tests still pass (especially e2e).
- **Docs to Update:**
- Update `docs/REPO_MAP.md` if needed to add the new models in the schema listing (optional, since Repo Map might not list every model but could mention Track/Playlist now exist).
- Update `docs/plans/M1_LIBRARY_PLAN.md` to mark schema tasks done, or update any changes from plan (if plan said X but we did Y, note the change).
- Create `docs/schema.graphql` (if repository desires a dumped schema) – optional.
- Possibly add `docs/features/library-api.md`: document the GraphQL schema for library (types and sample queries) [201]. This could be done in a later PR (1D) when everything is finished, but starting it now with at least type definitions and simple examples might be good.
- **Telemetry:** Instrument key parts:
- In the Next.js REST routes we often use `withSpan`. For GraphQL Yoga, we can manually add logging. Perhaps include a console log or a span when a mutation occurs. Since we have `ensureTracer("graph-api")`, we might wrap certain resolver logic in `tracer.startActiveSpan` (similar to how presence does in realtime gateway) [160].
- Add minimal telemetry: e.g., after creating a playlist, do `console.info("Playlist created", { playlistId, userId })` or using OpenTelemetry if set up (maybe not trivial here).
- Telemetry stub example: In `createPlaylist` resolver, add `span.setAttribute("playlist.id", newPlaylist.id)` if we had a span.
- At least, mark TODO comments like `// TODO: add telemetry event for track search or track play`.
- **Verify Locally:**
- Run `pnpm db:migrate` to apply migrations to dev DB. Check in Postgres that new tables (Track, Playlist, PlaylistItem) are created with correct schema.
- Run `pnpm db:generate` then start `pnpm dev --filter @omnisonic/graph-api` and see that it boots without errors.

- Manually test via GraphQL playground (if Yoga's playground is enabled):
    - Create a playlist:

      ```
      mutation { createPlaylist(name: "Test List") { id name } }
      ```

      (You might need to include an Authorization header if required; possibly disable auth for dev to test).
    - Note the returned playlist id. Then create a track (maybe directly in DB or we can add a temporary mutation for creating track if needed to test playlist linking). Perhaps easier: use Prisma studio or psql to insert a track row with your userId.
    - Test query libraryTracks: see it returns that track.
    - Test addTrackToPlaylist using the IDs: ensure no errors.
    - Query playlist(id) and see if tracks are included.
- Run `pnpm test` – adapt any tests that might break if assuming certain Prisma client behaviors (shouldn't, as we mostly added new stuff).
- Check that existing app (Studio UI) still runs and nothing crashes due to these additions (it shouldn't, since we haven't integrated UI).
- **Definition of Done:**
- [x] Database schema extended with Track, Playlist, PlaylistItem models and migrated.
- [x] Prisma client updated; new fields accessible via `prisma.track`, etc., with no type errors.
- [x] GraphQL schema extended with Track/Playlist types and relevant queries/mutations stubbed or implemented.
- [x] Basic resolvers for tracks/playlists function (tested via GraphQL queries or direct calls).
- [x] Auth rules enforced in resolvers (no cross-user data leaks).
- [x] All tests passing, including new ones for track/playlist basics.
- [x] Documentation reflecting new schema is updated (or planned to be updated in final milestone docs).
- [x] No regression in existing features (login, session mgmt, upload) – verify by running the app and maybe doing a quick session upload (should still work as we didn't touch it).
- **Out of Scope:**
- This PR does **not** implement the file import or metadata extraction logic [205]. It does not handle uploading files to create tracks. Track records might be created manually or via a simple placeholder mutation (which we haven't exposed to GraphQL, as we'll use the import route in next PR).
- It does not implement the search weighting or advanced filter logic yet (we may only do a simple contains search or none at all until PR 1D) [208].
- No front-end components or pages – this is purely backend. The UI will come in PR 1C.
- Not handling playlist sharing or public visibility (left `isPublic` out for now).
- Not implementing track deletion or any cascading behavior on user deletion (not needed yet).
- Not integrating with the existing Recording/Work models – we treat Track as separate for user library. (We might unify later but out of scope now.)
- No AI or analysis fields (bpm, key) population – though we added bpm/key fields, this PR won't populate them (left null unless input specifically sets them via updateTrackMetadata).
- No subscription events for track or playlist changes – those could be future enhancements.

PR 1B: "Library Import Endpoint & Audio Metadata Extraction"

- **PR Title:** Library Import Endpoint & Audio Metadata Extraction
- **Goal:** Implement a new backend endpoint that allows users to import audio files into their music library, automatically extracting metadata (title, artist, album, etc.) from the files' ID3 tags and storing it in the new Track model. Reuse the existing upload/storage mechanism for file handling so that the audio content is stored in our configured storage (local/MinIO), and create corresponding Track records in the database with metadata and links to stored files. In short, this PR delivers the **"Import to Library"** functionality for one or multiple files, with metadata parsing [237].
- **Files to Inspect First:**
- `apps/studio-web/app/api/upload/route.ts` – Use this as reference for handling file upload streams and storage integration [64] [67]. We'll likely create a similar route for library import (e.g., `/api/library/import`).
- `packages/storage/src/index.ts` – Verify how to use `putObject` and `getDownloadUrl` for saving files and generating URLs [38] [24]. We will call `putObject` in our import endpoint just like the upload route does.
- **ID3/Metadata library docs** – Choose a Node library for reading audio metadata. Likely `music-metadata` (from NPM) which supports MP3, FLAC, etc. Check usage examples (e.g., `import { parseBuffer } from 'music-metadata';` or `parseStream`). Alternatively, if `music-metadata` is already a dependency or similar. If none, we'll add it.
- `packages/db/prisma/schema.prisma` – Look at Track fields we need to fill (title, artist, album, year, genre, durationMs, artworkUrl) [238] and Upload vs Track relation. Also note if we added `uploadId` in Track in PR1A; if so, we'll set that.
- `services/graph-api/src/index.ts` – See if any portion of GraphQL is relevant for import. We might not use GraphQL for import because file upload suits REST better. But if we consider adding a GraphQL mutation accepting Upload scalar (which is complex), likely we stick to REST.
- **Implementation Tasks:**
- **Create Import API Route:** Under `apps/studio-web/app/api/library/import/route.ts` (new file):
    - Use `export async function POST(req)` similar structure to upload route [58].
    - Authenticate user (perhaps use the same `requireUserId` logic from upload route) [56].
    - Parse `req.formData()` to get files. Accept multiple files (the front-end will send multiple).
    - For each file in form data:
    - Validate type and size (likely reuse rules: must be audio or maybe also accept .wav, .flac, .mp3 etc.). Use similar logic as `ALLOWED_MIME_PREFIXES` from upload route [239].
    - Save the file to storage: generate a storage key (perhaps reuse `generateStorageKey(userId, file.name)` [64]).
        - We might want a different prefix for library tracks vs session uploads. The current generateStorageKey uses `uploads/<userId>/<timestamp>-<filename>`. That's fine to use for library too (no harm mixing in same bucket). Or we could prefix `library/` vs `sessions/`. Not crucial; using same is okay but maybe do `uploads/{userId}/library/{filename}` to differentiate. Up to us – for clarity, maybe incorporate "library" in key. We can modify `generateStorageKey` to accept an optional subfolder or create a separate function. Simpler: use generateStorageKey as is for now (it time-stamps so uniqueness is fine).
    - Call `putObject({ key, contentType: file.type, body: buffer })` to store file [64].

- Get back a `storageUrl`.
- Read metadata from the file:
  - Use `music-metadata` (install if not present: e.g., `npm install music-metadata`). It can parse from a Buffer or stream. We have the file as a Buffer (we needed to buffer to send to storage anyway).
  - `const meta = await mm.parseBuffer(buffer, file.type)` (with appropriate import alias).
  - Extract fields:
  - `meta.common.title`, `meta.common.artist`, `meta.common.album`, `meta.common.year`, `meta.common.genre` (genre might be array, take first).
  - Duration: If `meta.format.duration` is provided (in seconds), convert to milliseconds.
  - Artwork: `meta.common.picture` array might contain an image (usually first element has `data` and `format`). If present, we need to store artwork:
    - We can call `putObject` again with a key like `artwork/<trackId or unique>.jpg` or so. Or reuse the same key but different extension? Better to separate: `covers/{userId}/{timestamp}-{originalFileName}.jpg` or so.
    - Or store as a data URL in DB which is not ideal. Better store image in storage and save URL.
    - For now, implement storing artwork if found:
    - define a new storage key e.g. `uploads/<userId>/artwork/<timestamp>-cover.${ext}` (ext from format or assume jpg).
    - call putObject for that image buffer.
    - get a URL (or if none because local, we'll have a way via /api/upload/file).
    - We'll then set Track.artworkUrl to that returned URL (or local path).
    - If no artwork found, Track.artworkUrl remains null.
  - If some fields are missing:
  - Title: fallback to file name (without extension) [190] .
  - Artist: if missing, maybe leave blank or "Unknown".
  - Year/genre: optional, fine if missing.
  - Ensure to catch errors in parsing: if parse fails (e.g., unrecognized format), proceed without metadata beyond filename.
- Create a Track record in the DB:
  - Use `prisma.track.create` with data: { userId, title, artist, album, year, genre, durationMs, bpm: null (unless we intend to compute BPM now, which we don't in this PR), key: null (musical key not from ID3 typically), artworkUrl (set if we stored), source: 'IMPORT', // If we have uploadId field in Track schema, we need to create an Upload entry or somehow link. But perhaps we decided not to tie directly and just rely on storageUrl. // Alternatively, we could also create an Upload row for completeness, but that might be duplication. Possibly skip creating Upload model entry for library (not needed unless we want to manage cleanup). // Actually, consider: our storage cleaning logic (upload-cleaner) might assume anything in Upload table older than X is not used. If we don't create Upload entries for library tracks, those files might be considered orphan. So, to integrate with existing logic, maybe we *should* create an Upload entry too and mark it not tied to session. // E.g., create prisma.upload for each imported file with sessionId null. That way the upload-cleaner will not delete them if we modify that

logic to consider library tracks. // The upload-cleaner currently deletes uploads older than 30 days *without a linked session* [82]. If we do nothing, it might delete library tracks (since those uploads have no session). // So solution: either adapt upload-cleaner to skip if corresponding Track exists, or easiest: stop upload-cleaner from deleting anything with sessionId null (maybe not good if user uploaded something that didn't become a track). // For now, we can create Upload records for library imports to satisfy current cleaner logic (set sessionId null, userId, storageKey, storageUrl, etc.). Mark them with maybe a different mime or flag? Or consider adding a field to Upload like isLibrary = true, but that complicates. // Simpler: modify upload-cleaner later to exclude uploads that have a Track referencing their storageKey. That is advanced; skip for now, but note potential issue. }

    - Actually, let's do this: **Also create an Upload record** for consistency and potential re-use of download logic. Many front-end components currently expect an Upload when dealing with files. Our library might benefit from re-using the same download URLs.
    - Create upload = prisma.upload.create({ data: { userId, sessionId: null, fileName, fileSize, mimeType, storageKey, storageUrl } }).
    - Then set track.storageKey or track.uploadId if we have such field referencing upload.
    - If Track has `uploadId`, we link it to that upload's id.
    - If not, we at least have the upload record in DB for completeness (though not directly linked, we could find by storageKey).
    - The advantage: our existing `/api/upload/[id]/file` route can serve local files if needed. But it requires upload ID. If we link Track to Upload, we can get that ID for download route or we could embed the URL anyway. We'll go with linking if possible.
  - Collect the created track (and maybe upload) in an array to return.
  - After processing all files, prepare a response:
  - Possibly an array of tracks, plus their `downloadUrl` similar to upload route.
  - Actually, better: mimic the response shape of upload list: e.g., `{ tracks: [ { track, downloadUrl }... ] }`.
  - For downloadUrl: we can call `getDownloadUrl(key)` for each file. If it yields a URL (like S3 signed or CDN), use it. Otherwise, fall back to our own API route as we do in upload (i.e., `/api/upload/[upload.id]/file`) [70].
  - Now, if we didn't create Upload entry, we wouldn't have an ID for file route. But since we are creating one, we have upload.id. So we can do the same: attempt getDownloadUrl; if null (for local storage), use origin + `/api/upload/${upload.id}/file`.
  - Return the list of `{ track, downloadUrl }`.
  - Make sure to handle errors gracefully: if storage or DB fails for one file, proceed to next or abort all? Possibly abort on first error and return error message. Or attempt all and report successes and failures. Simpler: if one fails, return error status 500 with message; user can retry that file.
  - Ensure the whole process is within `withSpan("studio-web-api", "library.import", ...)` for telemetry consistency (similar to upload.post span) [58].

- **Integrate with Track Model:**
  - If in PR 1A we did not include an `uploadId` in Track, then we rely on `storageUrl` in upload entry. We could store storageKey in Track as well for easy reference (maybe not needed if we can join via upload).
  - Confirm Track-Upload linking: if Track has no field for it, perhaps add one now (we can still modify since earlier PR is not merged in real scenario, but let's assume it is).

    - If adding `uploadId String?` to Track now, that's a minor schema change – but we ideally included it earlier. If not, we can still create Upload records and maybe leave an TODO to link them by storageKey if needed later.
    - Given the cleaner argument, ideally Track.uploadId was added. Let's assume we have Track.uploadId (if not, consider doing a quick migration to add it, but that complicates PR dependencies).
    - We'll proceed as if track and upload are separate but use storageKey to correlate in logic.
- **Update Upload Cleaner (optional):** If time, adjust the `upload-cleaner` service logic to avoid deleting library files:
    - Open `services/upload-cleaner/src/index.ts` (or wherever the logic is) – it's likely scanning for Uploads with sessionId = null older than X [82].
    - Modify condition: maybe skip those where an associated Track exists. But that requires joining track, maybe heavy. Alternatively, mark library uploads differently.
    - Perhaps simpler: do nothing now; assume user will keep library tracks anyway and if we risk deletion, we note it as an issue to fix in milestone 3 or in docs.
    - Out-of-scope if complex, but at least add a comment or issue for it.
- **Front-end Integration Prep:** Though the actual UI (upload button on library page) will come in PR 1C, ensure that this endpoint is discoverable:
    - Add it to Phase 1 spec or relevant docs if listing endpoints.
    - Decide route path: probably `/api/library/import` as above. We'll call it via fetch from the Library page.
    - No front-end code changes in this PR, but we can test it manually via something like curl or Postman to ensure it works (simulate form-data with audio file).
- **Tests to Add:**
    - Add a **unit test** for the metadata extraction function if possible: e.g., create a small MP3 buffer in tests (maybe encode a known ID3 tag in it) and run our parse logic to see if it correctly gets metadata fallback, etc. Could use a fixture audio file: perhaps include a tiny MP3 in `apps/studio-web/e2e/fixtures` that has known tags (we have `sample.wav` but WAV might not have ID3 tags easily). Maybe find a tiny MP3 with tags or generate one.
    - Alternatively, test the Import route handler with a mock FormData containing a small file: use Node's `File` and `FormData` polyfill to simulate a Request as we do for API routes testing.
    - Validate that after import, the database has new Track(s) with correct fields and Upload(s) created.
    - Possibly test that the response includes a downloadUrl that is accessible (simulate getting it).
    - E2E: Extend Playwright test (if possible) to use the UI when built in PR 1C, but for now skip E2E automation of import (lack UI).
- **Docs to Update:**
    - `docs/features/library-api.md` – document the new REST endpoint `/api/library/import` with sample request (form-data with files) and response [240].
    - If there's an API table in Phase 1 spec or separate doc, add a row for this endpoint (method POST, etc.).
    - Update `README.md` or environment docs if any new env needed (we might not need any new env for this, unless adding ffmpeg – but we decided not to require ffmpeg for just tags).
    - Possibly note in `docs/STATUS_GAPS.md` that library import is now done.
    - If any known limitation (e.g., "Large files may slow down import" or "Embedded artwork is extracted for JPEG/PNG only, other formats ignored"), mention it in docs or comments.
- **Telemetry:**

- Add OpenTelemetry tracing around the import process: e.g., in the route handler, wrap with `withSpan("studio-web-api", "library.import", ...)` as suggested 58 .
- Set attributes like total files imported, total bytes, etc. For each file, record something like `span.addEvent("imported file", { fileName, trackId })`.
- If any error occurs for a file, use `span.recordException`.
- Log warnings if metadata parse fails or if artwork not found (maybe `console.warn`).
- **Verify Locally:**
- Build and run migrations if needed (if we changed track schema for uploadId).
- Start the dev server and make a manual HTTP request:
    - Use `curl -X POST http://localhost:3000/api/library/import -F "file=@path/to/song.mp3"` with cookies or auth (maybe log in via browser first, or temporarily disable auth check for testing).
    - Check the server logs for any errors.
    - See the response JSON: should list track info. Verify title/artist in response matches the file's tags or filename fallback.
    - Check database: `prisma.track.findMany()` in a Node REPL or via Prisma Studio to see the new track.
    - Check the `uploads` table for a new entry with storageKey matching track's file.
    - If running MinIO or local storage, verify file actually saved (e.g., in `.uploads` directory).
    - Try a file with known ID3 tags to see if extraction works. Also try a WAV (no tags) to see fallback to filename.
    - Use the returned downloadUrl: if local, it will likely be something like `http://localhost:3000/api/upload/XYZ/file`. Hit that URL in browser or curl, ensure it downloads/plays the content.
- Run tests (`pnpm test`) to ensure all new and existing tests pass.
- Evaluate performance with a moderately sized file (like a few MB) to ensure no huge delays or memory issues. The dev server console might show memory usage – if parseBuffer is okay (should be).
- **Definition of Done:**
- [x] `POST /api/library/import` route implemented and accessible, requiring auth.
- [x] Multiple files import supported in one request (the FormData loop works).
- [x] Audio files are stored in the configured storage backend (verify on disk or bucket).
- [x] Track entries created in DB with correct metadata (title, etc. or fallback, and source=IMPORT).
- [x] Upload entries created for those files (so that `downloadUrl` and cleaning logic integrate).
- [x] Embedded artwork is saved and Track.artworkUrl populated when applicable (test with an MP3 that has album art).
- [x] Metadata extraction covers at least title, artist, album, year, genre, duration; fields appear reasonable.
- [x] The response returns the newly created track info along with a working download URL for each.
- [x] AuthZ: a user can import only into their own library (the route inherently uses session user, no userId parameter from client).
- [x] Added tests around metadata parsing and route logic are passing.
- [x] No regression on existing file upload (session upload unaffected).
- [ ] Documented usage of the import endpoint and any noteworthy constraints.
- **Out of Scope:**
- This PR does **not** implement the front-end UI that calls this endpoint (that's PR 1C).

- It does not do any BPM or key detection (that's for AI milestone). So Track.bpm and Track.key remain null unless present in metadata (rarely these are in ID3 tags, and we aren't extracting them anyway).
- It doesn't integrate imported tracks into GraphQL queries yet except they will show up in `libraryTracks` because we fetch all tracks for user. But maybe add a note: GraphQL `libraryTracks` now will include newly imported tracks since they are in Track table.
- It doesn't handle duplicate detection (if same file imported twice, we will create two tracks; no dedupe).
- No progress feedback – this is a one-shot call; if a user imports 20 files, it may take a few seconds and the client has to wait (we can refine later with maybe chunking or progress events).
- Not dealing with non-audio files (we restrict to audio/* and maybe ignore video though video could be considered for extraction audio – skip).
- Not doing advanced error handling per file (if one file fails, we might currently .catch and continue, but primarily focusing on success path).
- The `upload-cleaner` nuance: We have not fully ensured library uploads won't be deleted by cleaner. We'll assume either to adjust cleaner soon or in worst case instruct disabling cleaner if using library heavily. We'll mention as a note if needed.

PR 1C: "Library UI Pages & Playlist Management"

- **Goal:** Create the user interface for the Music Library feature in the Studio web app. This includes a **Library List page** where users can see all their tracks, search and filter them, and an **Import button** to add new tracks (hooking into the import endpoint from PR 1B). Also a **Track Detail page** to view track info, play the track, and edit metadata, as well as basic **playlist management UI** (create playlist, add/remove tracks, view playlist). The UI should integrate with the GraphQL and REST APIs implemented in prior PRs.
- **Files to Inspect First:**
- `apps/studio-web/app/page.tsx` – Possibly how the landing or sessions page is structured. Might mimic patterns from sessions listing [144].
- `apps/studio-web/app/sessions/page.tsx` (if exists) or similar – see how list of sessions is implemented (component structure, forms for create).
- `apps/studio-web/components/session/upload-panel.tsx` – For reference on how uploading in session works (to adapt an import flow) [59] [241].
- `apps/studio-web/hooks/usePresence.ts` – Unrelated to library, but check if any search or list hooks exist (maybe not).
- `packages/ui` or similar – is there a design system (shadcn UI components)? Possibly in `apps/studio-web/components/ui/` for things like Button, Input (we saw usage of `<Button>` in upload panel which likely comes from shadcn) [242] [243]. We'll use those.
- **Implementation Tasks:**
- **Add Navigation to Library:**
    - Insert a link in the main navigation (if there's a sidebar or header menu) for "Library". For example, if there's a component `NavBar` or layout in `apps/studio-web/app/(...)layout.tsx` where sessions and maybe profile are listed, add "Library" tab that navigates to `/library`.
    - Ensure it's only shown to logged-in users (the whole app probably is authed anyway).

- **Library List Page (`/library`):**
    - Create `apps/studio-web/app/library/page.tsx`. This should be a **server or client component** depending on data fetching approach. We likely will fetch library tracks on the client side via GraphQL or a hook, because search will be interactive.
    - Possibly use Next.js 13 *Server Components* for initial load and then hydrate for interactivity. But simpler: make it a Client Component (with `"use client"` at top) so we can manage state for search, filters.
    - Use `useEffect` or React Query or SWR to fetch tracks from GraphQL API (GraphQL endpoint might be at /api/graphql or separate server at port 4000; if GraphQL is separate, we might need to proxy or use a cross-origin fetch).
    - Perhaps easier: we could create a Next.js API route `/api/library/tracks` that calls `prisma.track.findMany` directly as a simpler approach to avoid GraphQL complexity on client. However, since we invested in GraphQL, let's try use it.
    - Potential approach: use `graphql-request` or Apollo Client. But bringing Apollo might be heavy if not already in project. If minimal, we can do a fetch to our GraphQL API (like `fetch('http://localhost:4000/graphql', {method:'POST', body: { query: ... }}` with proper headers).
    - Alternatively, since Next and GraphAPI are separate processes, perhaps the Studio app isn't yet configured to talk to Graph API. It's possible they planned to do so but not done. We might for now cheat by using the Prisma client directly in the Next app for library queries (not great separation, but in monorepo it's feasible to import `@omnisonic/db` in the Next app and query server-side). Given time, using Prisma on server components is straightforward and avoids needing to stand up GraphQL consumption.
    - Actually, doing server-side: we can make `/library` a server component and use `prisma.track.findMany` in `getTracks()` inside it (since Next 13 supports async server components). That could work nicely for initial list and even filters via server actions or search via a form.
    - But search needs to update quickly without full page reload. Perhaps we can use a client component for the search bar and do client fetches (either to a Next API route or GraphQL).
    - Balanced approach:
        - Initially load library tracks on server component (fast initial render, no loading state for initial content).
        - Then have a client-side search input that triggers client side filtering: we can either filter the already loaded list on client (for small libraries fine, for big not) or call an API.
        - Probably call the existing GraphQL Query for search to leverage DB search. Use fetch or Apollo for that.
    - Layout: Use a table or list. Possibly like:
    - Search bar at top (Input and maybe filter dropdown for genre).
    - "Import tracks" button at top to trigger file selection.
    - List of tracks: maybe a simple table with columns: Title, Artist, Album, Duration, and a "..." menu for actions (edit, add to playlist, delete).
    - Each row clickable to go to detail page or have a play button.
    - If using shadcn components, see if there's a Table component. If not, basic `<table>` styling with Tailwind.
    - Pagination: If user has many tracks, ideally paginate or infinite scroll. For now, we can load all tracks (assuming manageable count < couple thousand). If performance issues, implement a "Load more" or pagination control.

- For each track, possibly display if it's currently playing (we might highlight, handled by detail or a global player state).
- State management:
- Use `useState` for search query and filters.
- Use `useEffect` to refetch track list whenever query/filter changes (debounce input by ~300ms).
- The fetch function: can call our GraphQL API (maybe accessible at `/api/graphql` if we set up a proxy route in Next for it, or directly to the Graph API endpoint with proper CORS).
- If GraphQL is complicated, consider adding a fallback Next API route `GET /api/library/tracks` that uses Prisma to query tracks (with query param for search). That might be simplest given time. Then use `fetch('/api/library/tracks?q=...')`.
- We'll do that: implement quick `app/api/library/list/route.ts` as GET that returns `{ tracks: [...] }` by querying `prisma.track.findMany` with contains search on title/artist. This leverages server directly and avoids cross-service call. It's a bit redundant to GraphQL but pragmatic.
- Hook up the Import button:
- Possibly reuse the `<input type="file" multiple>` and hidden approach used in UploadPanel [244].
- Create a component `ImportTracksButton` with an invisible file input (accept multiple audio/*).
- On change, call our import API: basically replicate `handleUpload` from UploadPanel but to `/api/library/import` and handling multiple files.
- Show a loading state or progress (for many files, could list them as uploading).
- After import API returns, update the track list state with the new tracks (or refetch from backend).
- Also possibly display any errors from import (if we catch them).
- For styling: keep consistent with sessions page: likely minimal, tailwind classes (e.g., headings, spacing).
- Ensure the page is responsive (tailwind usually helps, but if tables overflow, consider making it scrollable).

• **Track Detail Page (`/library/[id]`):**

- Create `apps/studio-web/app/library/[id]/page.tsx`. It's a dynamic route page.
- Use a server-side data fetch for the track by id. Possibly use Prisma directly: `const track = await prisma.track.findUnique({ where: { id }, include: { upload: true, playlists: { include: { playlist:true } } } })`. Or call GraphQL query.
- If we have an Upload linked, we can get the storageUrl and generate a playback URL. But better to rely on the `downloadUrl` we can compute via API: for local, we need to go through /api/upload file route for streaming, for S3, the storageUrl might be public or require signing again (we can reuse getDownloadUrl).
- Perhaps use our library list data or small GraphQL query. Simpler: create Next API route `GET /api/library/track?id=` to fetch track + upload and return a `downloadUrl` by calling `getDownloadUrl`. We actually have similar logic in GET `/api/upload/[id]` route [53], we can replicate for track id.
- But to avoid too many new endpoints, we can also have the detail page call the existing list of libraryTracks and filter by id, but that's inefficient.

- I'll plan to do server fetch: `track = await prisma.track.findUnique(...)` and then for streaming:
- If track.uploadId exists, get `upload = prisma.upload.findUnique({id: track.uploadId})` and then build a downloadUrl as done in /api/upload GET [245] (using storage util getDownloadUrl with try).
- If track.uploadId is null (maybe if we decided not to link), instead use track.storageKey if we had it or call getDownloadUrl on track.storageKey if we had store it. But we didn't store storageKey in track, only in upload. So linking is beneficial.
- Let's assume track.uploadId exists from PR 1B's implementation.
- Render the track details:
- Display title as <h1>, and maybe a subtitle with artist - album - year, genre, etc.
- Show the cover art if artworkUrl exists: use <Image> tag linking to artworkUrl. If artworkUrl is a local `local://...` scheme or something, we might need to convert it (likely `putObject` returns either CDN or local://key).
  - If local://, maybe we rely on /api/upload file route by ID of artwork. Did we create an Upload for artwork? Possibly not, we directly got a CDN or local file path. For simplicity, we might skip showing cover if it's complicated, or ensure we also created an Upload entry for artwork with an ID, so we can have a route.
  - Could store artwork in DB but not worth it. Perhaps we consider not implementing artwork display in UI due to complexity, or if `storageUrl` for artwork has http scheme (like if using local, getDownloadUrl would have returned null and we fallback to /api/upload/id/file – but we didn't create upload for artwork because no user-facing reason, however maybe we should for consistency).
  - Could cheat: load artwork by making a small Next API that reads from storage (like how /api/upload file does) given we have key.
  - If time short, we might skip artwork showing or leave a placeholder image if artworkUrl is not a fully accessible URL.
- Provide an audio player:
  - Could use plain `<audio controls src={downloadUrl} />` for now. That gives play/pause and scrub functionality out of the box.
  - If we want a custom UI later (waveform, etc.), skip now.
  - The `downloadUrl` likely will be to /api/upload/ID/file if local, or a signed S3 link, which <audio> can use.
  - Possibly store `downloadUrl` in state via a useEffect calling an API if not loaded server side. But we can also pass it from server as a prop (the server can pre-compute a signed URL or as said).
  - But signed URL expires in an hour; however, likely fine (user usually won't keep page open that long; if they do, they can refresh or we can refresh URL with an on-demand call).
  - We'll do: server side attempt getDownloadUrl; if returns null (because local and no CDN), we'll generate a URL to our own API route. This we can do server side if we know origin (we might not easily know origin in server component).
  - Alternative: do it client-side via fetch to /api/upload/[id] which returns { downloadUrl }, similar to how upload list does.
  - Actually, easier: the `libraryTracks` query already returned a downloadUrl in PR 1B response, if we use that, the client might have it from initial load. But in server component, not easily without static generation.

        - Possibly do detail page as client component and fetch track + downloadUrl via our own internal API (like /api/upload/{uploadId}) when component mounts.
        - Simpler approach: embed downloadUrl in a data attribute in HTML via server if possible, else fetch on client.
    - Provide an Edit button: to edit metadata. Use shadcn Dialog or a simple form.
        - On click Edit, either navigate to an edit page or open a modal with a form (fields for title, artist, album, year, genre).
        - Implement onChange for these fields and a Save button.
        - Use the `updateTrackMetadata` GraphQL mutation (or create a Next API route for it calling prisma.track.update).
        - For quickness, maybe a Next API route `POST /api/library/update` expecting trackId and fields.
        - Or directly call GraphQL from client with a fetch.
        - Could do `await prisma.track.update` in a server action (Next 13 allows defining an async function in component that executes on server on form submit).
        - Actually, Next has a new "Server Actions" feature with forms that can call a server function. This is cutting-edge but could be used. If not comfortable, use client-side fetch.
        - Possibly simplest: in the modal component, do `fetch('/api/library/update', {...})` with JSON.
        - We'll implement a small route: `PUT /api/library/track` that updates fields (ensuring auth).
        - Then on success, update UI (maybe close modal and update state or refetch track).
    - Provide an "Add to Playlist" control:
        - Could be a button that opens a dropdown or modal listing user's playlists (we need playlist data, which we can fetch server-side or have via library list context).
        - Possibly in detail, we know which playlists this track is already in (if we did track include playlists in query).
        - But to add, we can call GraphQL `addTrackToPlaylist` or our own route as done.
        - Simplest: in detail page, have an "Add to Playlist" that opens a small popup:
        - If user has no playlists yet, maybe show "No playlists. Create one first."
        - If playlists exist, list them with checkboxes or plus buttons.
        - On selecting one, call `POST /api/library/playlist/add` or GraphQL mutation with trackId & playlistId.
        - Because of time, maybe implement just one: allow adding to one playlist at a time via a select dropdown + button.
        - If time is short, one could skip detailed UI: just show playlist names this track belongs to, and an input to add to a playlist by name (which if not exists, could create new). But that might be too fancy.
        - Perhaps better to implement playlist add in playlist context, see next bullet.
- **Playlist Pages:**
    - Perhaps not required in depth for milestone acceptance, but it's in scope to show at least basic playlist usage.
    - Create `/library/playlists` list page or incorporate in library main page as a sidebar.
    - Could show a list of playlists and allow selecting one to view tracks in that playlist.
    - Simpler: a section on library page with playlists.

    - Possibly implement as separate route: `apps/studio-web/app/library/playlist/[id]/page.tsx` to show tracks of a specific playlist.
    - At minimum implement "Create Playlist":
    - On library page, a small form (e.g., an input + button or a modal) to create a new playlist given a name.
    - On submit, call our GraphQL or Next API to create playlist.
    - Update playlist list UI.
    - "Add to Playlist" interactions:
    - Option 1: In library list, allow multi-select of tracks and then add to playlist (out of scope for now).
    - Option 2: On each track row or detail, have an add to playlist control as described.
    - Show playlist content:
    - Possibly reuse the same list component but filtered to that playlist's tracks. E.g., a playlist detail page showing tracks (like library list but only those tracks, maybe with their positions).
    - Provide remove from playlist action (maybe a trash icon on that page for each track).
    - Perhaps allow reordering via drag-and-drop or up/down buttons (if time allows minimally up/down).
    - Implement remove by calling `removeTrackFromPlaylist`.
    - Reordering could call `reorderPlaylistItems` with new order or multiple calls to update positions; might skip reorder in UI if too complex.
    - UI hints from sessions:
    - If sessions list had create session inline with list, we can do similarly for playlists (maybe a text input and + icon).
    - Keep playlist UI minimal but functional.
- **Search Implementation:**
    - On the library page, implement searching:
    - Already covered partially: input triggers refetch.
    - Use either GraphQL query with text search or our Next API. If we did Next API in step 2, implement `app/api/library/list` GET:
        - Use query param `q` to filter by `OR title ILIKE q OR artist ILIKE q OR album ILIKE q` perhaps.
        - Also consider filters like genre (if user picks from a dropdown of genres present).
        - Also consider BPM range or key if we wanted, but we have no data for those yet.
        - Sorting: maybe sort by title by default. Could allow sort by artist.
        - But keep it simple: sort by title ascending always.
    - Ensure the search is case-insensitive and works with partial matches.
    - Debounce: use a `setTimeout` in onChange or use a library.
    - Testing search: ensure it yields expected results quickly (maybe test in UI after hooking up).
- **Player integration:**
    - The `<audio>` approach on detail works, but maybe the user wants to play directly from list.
    - If possible, add a small play button icon on each track row in library list:
    - On click, either navigate to detail or directly play the audio without leaving page.
    - Possibly implement a lightweight player in library page that when you hit play, it loads the audio and plays.
    - Could use the HTML audio element hidden or one per row and just call play on that track's element.
    - Simpler: navigate to detail on play might be okay for now (less user-friendly though).

- Perhaps better: we implement a simple audio context in the list:
    - Maintain a state `currentTrackIdPlaying` and an <audio> element in the library page component outside the list (global for that page).
    - When user clicks play on a row: ~ If it's not playing anything yet or playing another, set currentTrackId to that, set audio src to that track's `downloadUrl` (we need that in list – we might have them from import or we can generate them similarly as upload list did). ~ Then call audio.play(). ~ If they click pause, call audio.pause() and update state.
    - You might need to prefetch `downloadUrl` for each track in list to implement this. That could be heavy if many tracks (but we could get from /api/upload for each maybe).
    - Simpler: clicking play could fetch /api/upload/{uploadId}/file but that returns binary. Instead, audio src can be set to '/api/upload/{uploadId}/file' directly, which will trigger a GET when audio loads. That's fine.
    - So we can do that if we have uploadId. If not, use storageUrl (S3 or CDN) directly as src if available.
    - For local, have to use /api/upload route since we can't access .uploads folder directly.
    - We can embed uploadId in a data attribute or as part of track object in state (if using our /api/library/list route, we can include uploadId, as we can join track with upload).
    - Yes, modify our list API to include `uploadId` in each track (since we know it server-side).
    - Then the list item has something like: `<button onClick={() => playTrack(track.uploadId)}>Play</button>`.
    - `playTrack(id)` sets audioRef.current.src = `/api/upload/${id}/file` and audioRef.current.play().
    - The audio element is maybe rendered with `controls` hidden and we manage via JS, or we could even display controls.
    - For simplicity, maybe show controls at bottom for whichever track is playing currently (like a mini player).
    - Could just reuse the audio controls on detail but in list context. Possibly show the HTML5 controls in a fixed bottom bar when a track is playing. That might require more state mgmt but doable.
    - Given time, possibly skip a fancy persistent player; just allow playing inline with an audio tag's controls in each row or open detail to play fully.
    - For MVP, we can require user to click the track and go to detail to play, which is acceptable albeit one extra step.
- We'll choose: detail page has player, list page doesn't directly play (to reduce complexity). If user clicks a track row, navigate to detail.
- However, if user just wants to quickly preview tracks, that requires going in and out; not ideal but we can refine in future.

• **Testing (Manual / Unit):**
- Unit test React components minimally (if using React Testing Library, test that library page renders track titles given a list prop or so).
- Possibly simulate a search: feed it a list of tracks in state and ensure filter function works.
- For integration:
- Spin up dev, navigate to /library, ensure it lists tracks (assuming some tracks in DB from earlier import).
- Test Import: click import, select a file, see it appear in list.

  - Edit metadata: open track detail, change fields, save, see updated on detail (and if go back to list, updated there too maybe by refetch or by optimistic update).
  - Playlist: create a playlist, then on track detail or list add to playlist, then visit that playlist page, confirm track is there.
  - Remove track from playlist, confirm removal.
  - Try search bar: type part of a title or artist, see list filtered down.
  - Try edge: search for something not present -> either empty list message "No tracks found".
  - Try import multiple files at once (control-select 2 files) and see both added.
  - If possible, have an MP3 with tags to see them show up, and one without to see fallback.
  - Cross-test: session uploads vs library: ensure session page still works and the presence of library code didn't break global styling or routing (like /sessions still accessible).
  - Check responsiveness: shrink window to mobile width, list items should ideally stack nicely (maybe not a priority but quick tailwind adjustments if needed).
- **Docs to Update:**
- If there's a user-facing guide, update to show how to use Library UI.
- `docs/features/library-ui.md` – write up usage instructions, maybe even include a screenshot placeholder (the prompt suggests adding screenshots placeholders) [203].
- Document how to import tracks, how to create playlists, etc.
- `README.md` might now highlight "Music Library & Search: implemented" or something.
- Possibly update `docs/STATUS_GAPS.md` marking "Music Library UI" as done.
- **Telemetry:**
- Add some events: e.g., when user imports tracks from UI, that calls our import route which already has telemetry. Could also instrument UI events slightly (not much telemetry on client unless we log to console).
- When search query is submitted, maybe call a telemetry function (if we had one) to count search terms (but not in MVP).
- Possibly instrument track play event: e.g., if using an audio element, listen to `onPlay` event and issue a fetch to some new endpoint `/api/telemetry/trackPlayed` with trackId. If that endpoint records in DB or logs, could be something. But out-of-scope unless trivial.
- Telemetry might mostly be on backend, which we handled partly. We can ensure our withSpan covers the import and perhaps we wrap GraphQL track mutations too (some already done in resolvers).
- **Verify Locally Steps Recap:**
- (Already described above in testing manual). Ensure combining with previous PR data and flows everything is cohesive.
- **Definition of Done:**
- [x] A user can navigate to a **Library** page from the app and see a list of their tracks (if none, maybe show "No tracks, start by importing").
- [x] The list is nicely formatted with key info (title, artist, etc.), and is updated when new tracks are added or edited.
- [x] The user can use a search bar to filter tracks by text (and optionally by genre etc.), and it filters in real-time or on enter.
- [x] The user can import audio files using an **Import** button. Selecting files triggers upload; after completion, the new tracks appear in the list with their metadata filled in (or filename if no tags). Large files might take a moment but UI shows an indication (like "Uploading...").
- [x] The user can click on a track to go to its **Detail** page. On detail:
  - They see all metadata and cover art (if available).

    - They can play the track using an audio player (and hear the audio).
    - They can edit the track's metadata via an Edit dialog and see the changes saved (and reflected in list on return).
    - They can add the track to a playlist via a control, and get feedback that it was added (maybe just silently updated playlist listing).
- [x] The user can create a new **Playlist**, see it in a playlist list (maybe on library page or separate page).
- [x] The user can view a playlist and see all tracks in it, in order. They can remove tracks from it.
- [x] Basic playlist ordering is preserved (e.g., insertion order or as updated via some method).
- [x] The UI uses existing design system components and looks integrated (buttons, inputs same style as rest of app).
- [x] All new UI elements are responsive or at least not broken on different screen sizes.
- [x] All tests are passing and no new console errors in browser or server logs.
- [x] Feature flag: If we planned `ENABLE_LIBRARY` flag, ensure toggling it hides the library UI and perhaps disables import APIs. (If not implemented, fine, but better to have a kill-switch). We might skip this for now if not specified, but original prompt suggested feature flags by default for heavy features [246]. Up to consider: could wrap library routes/components such that if env says disabled, maybe 404. Possibly skip due to time.
- **Out of Scope:**
- No waveform or audio visualization on track detail (just simple audio control) [195].
- No continuous multi-track playback or queue (just one track at a time).
- No share functionality (tracks are private).
- Not implementing an advanced tag editing UI (like multi-select tags or mood tagging).
- The design might not be polished for all edge cases (like very long titles might overflow, minimal styling only).
- Not dealing with thousands of tracks performance elegantly beyond simple pagination possibility.
- No confirming modals on delete actions (we assume user won't accidentally or we could add a simple confirm).
- The playlist reorder drag-and-drop not implemented (if needed, user can remove and re-add in desired order as workaround).
- Importing non-audio files yields an error, but we assume user will pick audio (error message might be generic).
- Not implementing saved searches feature (though prompt 1D suggests it, we likely skip because it's optional) [198].

PR 1D: "Enhanced Search & Library UX Polish"

- **Goal:** Improve the music library search functionality and overall UX based on initial implementation. This includes adding weighted search ranking (so results are ordered by relevance, e.g., title matches rank higher than others) [247], possibly adding a rudimentary "Did you mean?" suggestion for minor typos, and implementing any deferred features from Milestone 1 like saved searches if time permits. Also address any UI/UX issues or minor features left (like track deletion, feature flagging, etc.) to declare Milestone 1 complete.
- **Files to Inspect First:**
- `apps/studio-web/app/api/library/list/route.ts` (from PR 1C) – The current search/filter logic to enhance [209].

- `packages/db/prisma/schema.prisma` – If considering using Postgres full-text or trigram, maybe enable extension via migration. Check if `pg_trgm` extension was considered (not yet).
- `services/graph-api/src/index.ts` – If we want to do search on GraphQL side instead, see how to implement full-text in Prisma query (Prisma might not directly do rank, but we can do `contains`).
- `apps/studio-web/app/library/page.tsx` – The search implementation on client to adjust ordering display or incorporate suggestions.
- (Optional) Look up how to implement simple "did you mean": maybe using the `didYouMean` package or just a precomputed list of known track titles to do Levenshtein distance.
- **Implementation Tasks:**
- **Weighted Search Ranking:**
    - If using Postgres: one approach is to use `ORDER BY CASE ...` in query: e.g., if query "beatles", rank exact title match highest, then artist match, then partial.
    - If using Prisma, we may fetch all matches via OR and then sort in JS with a custom comparator.
    - Simpler: sort by whether title contains query (bool) descending, then artist contains, then album contains, then by title alphabetically.
    - e.g., in code: `tracks.sort((a,b) => score(b) - score(a) || a.title.localeCompare(b.title))`.
    - Where `score(track)` could give 3 points if title matches query substring, 2 if artist matches, 1 if album matches.
    - Or check prefix vs anywhere (prefix match might be better).
    - Without a full text index, this is a heuristic but okay for moderate lists.
    - Implement in either:
    - On server in `/api/library/list`: fetch all that match (like we did), then sort as described before sending.
    - Or fetch on client as we did and sort on client. But better on server to reduce data if needed.
    - We'll do it server-side in Node since easier than writing a PG query with CASE. We'll likely already have an array from Prisma, easy to sort.
    - Also consider exact vs partial:
    - If user enters full title exactly, that track should come first. We can detect equality (case-insensitive) for title/artist.
    - Maybe weighting:
        - if track.title ILIKE query exactly (after trimming) -> score 100
        - else if track.title contains query -> score 10
        - if artist contains query -> score 5
        - if album contains query -> score 3
        - if genre contains query -> score 1
    - This is arbitrary but ensure title hits rank top.
    - Implement accordingly.
- **Typo Tolerance ("Did you mean"):**
    - This can be complex, but we can do a simple suggestion:
    - If the search returns 0 results, then attempt to find the closest matching track title or artist using Levenshtein.
    - Could use `pg_trgm` similarity search if we had that extension enabled. Possibly faster: we can do it in JS by computing distance to all track titles in user's library. But if library huge, not great. If small, fine.

- Or we maintain an index of all unique words in track titles to compare.
- Simpler: we can use `natural` or `didyoumean` NPM but let's not add heavy dep. Instead:
  - If no result and query length >=3, go through user's track list (which maybe we have or can query all track titles and compare).
  - Use a function to compute Levenshtein distance (there are small implementations or code one).
  - Find the track with minimum distance in title or artist.
  - If distance <= 2 or so (meaning just a small typo), suggest that.
- This requires having all track titles which if library is large could be heavy. But maybe acceptable for up to thousands.
- Could limit to check only first letters or something for performance, but not necessary if moderate scale.
- Implement a helper function in the API route: if results array empty:
  - retrieve maybe up to 50 tracks even if not matching (maybe already did initial query that yielded none).
  - Actually, to get suggestions we need to consider all tracks, not just filtered by query because none matched.
  - We might have to fetch all user tracks or at least track names if none matched.
  - That could be heavy if user has thousands, but if none matched maybe thousands are fine.
  - Or better, maintain a global list on client or context. But easier: do one more DB query `prisma.track.findMany({ where: { userId } select: { title, artist } })` if we detect results empty (or cache it somewhere).
  - Then do distance calc on those to find best match.
- Provide suggestion: how to show to user? Could modify the API response to include a `suggestion` field if any.
- Or front-end can call a separate endpoint to get suggestion.
- Simpler: in our list route, if no tracks found and we found a suggestion name, we can either:
  - Return something like `{ tracks: [], suggestion: "Beatles" }`.
  - The UI sees no tracks and sees suggestion, then can display "Did you mean 'Beatles'?" as a clickable link to search that term.
- Implement accordingly. Possibly not super precise but at least addresses obvious typos by one or two letters.

- **Saved Searches (Optional):**
  - This was mentioned as optional but recommended [247] [199]. If time:
  - Add a `SavedSearch` model: id, userId, name, queryJSON (or just query string since we have only basic filters).
  - Provide UI: maybe on library page, an icon or button near search bar to "Save this search". Clicking prompts for a name, then calls an API to save.
  - Also show a dropdown of saved searches if any, to quickly apply them.
  - This is nice-to-have but can be skipped if short on time. Perhaps implement a simplified version:
    - Not altering DB (to avoid another migration).
    - Instead, maybe store in localStorage as a quick hack. But that wouldn't sync across devices.
    - Or implement minimal: using something like playlists approach but for queries – due to time, likely skip fully or just note in docs.

    - Because complexity vs value isn't huge, might skip actual implementation beyond structure.
    - I'll lean to skip code and mention it's deferred.
- **General UI Polish:**
    - Check for any usability issues:
    - Maybe add an indicator or toast after import completes (some feedback "3 tracks imported").
    - Possibly add a confirm dialog when deleting a track or playlist to prevent accidents.
    - Ensure editing a track updates the list without needing a full refresh (maybe our state management already covers if we updated state locally or we refetch).
    - If not, consider adding a state update or a call to refetch list after an edit or playlist change.
    - Implement track deletion if it was not done: user might want to remove a track from library:
        - Could add a "Delete track" button on track detail (or in row "..." menu).
        - If clicked, confirm, then call `prisma.track.delete` (which should also cascade its playlist items and ideally delete its upload & storage).
        - Actually, need to delete from storage too (like we do in /api/upload/[id] DELETE).
        - We could leverage that by calling that route for its upload or replicating logic:
        - E.g., call our upload DELETE for the underlying upload, which will remove file and DB entry.
        - But then track still in DB unless cascade (we didn't set cascade on track-upload relation likely).
        - Better: do track deletion and in code also call `deleteObject(storageKey)` from storage module and remove upload entry.
        - We'll implement a `DELETE /api/library/track?id=` route to handle removal:
            - Auth check, find track, get its upload (if any), use `deleteObject` to remove file from storage, then `prisma.upload.delete` and `prisma.track.delete`.
        - Then UI can call that on confirmation.
        - This ensures user can free up space and remove unwanted tracks.
    - Similarly, playlist deletion is probably implemented (we have GraphQL but easier to do via `prisma.playlist.delete`).
    - We'll ensure playlist deletion in UI if needed (maybe a trash icon on playlist page).
    - Check mobile UI: possibly stack columns or make table scroll horizontally. Maybe add some Tailwind classes or a media query to hide less important columns on small screens (e.g., hide album or year).
    - Accessibility: ensure buttons have labels, etc. Basic keyboard nav likely fine due to <button> usage.
    - Performance: if list is big, maybe add a simple pagination (like show first 100 and a "Load more..." button).
    - Could do that by adding `limit` and `offset` to our list API and a button. If time, maybe not needed, but consider if risk of too heavy. Possibly skip unless large libs expected (most personal libs might be <1000 tracks, manageable).
    - Feature flag gating:
    - If not done, add an environment check in navigation or layout to exclude Library UI when disabled (like `if (!process.env.ENABLE_LIBRARY) don't show link`).
    - On the server side, maybe protect library routes by returning 404 if disabled (Next can do that in route handler or config).
    - Also maybe skip GraphQL resolvers registration for library if disabled (not necessary but could).

    - Given the prompt emphasis on feature flags, good to include one. In `.env.example`, add `ENABLE_LIBRARY=true` by default for dev.
    - Implement in a central config (maybe `process.env.NEXT_PUBLIC_ENABLE_LIBRARY` to also use on client).
    - E.g., in Navbar component: `if (!enabled) return null` for library link.
    - And in `app/library` pages: if disabled, maybe redirect or show 404 (we can check env in server component).
    - We should test turning it off hides features.
- **Testing:**
    - Repeat manual tests with search ranking: test queries where a partial match in title vs in artist to see ordering. E.g., have track "Love Me Do" by Beatles, and track "Do Re Mi" by someone; search "do" should ideally rank "Love Me Do" higher as "Do" is entire word vs partial in "Do Re Mi"? Actually both have "do". If we gave title heavier, "Love Me Do" might come first because "do" at end vs "Do Re Mi" "Do" at start – depends how we weight maybe we don't differentiate position.
    - Test a typo suggestion: if searching "Beatls" returns suggestion "Beatles".
    - If suggestion displayed, click it or manually correct search to see that results appear.
    - Ensure no weirdness if suggestion itself doesn't yield results (in case suggestion algorithm picks something that still yields none due to combined query).
    - If implemented track delete: try deleting a track, check DB that track and upload removed, file removed (check .uploads directory or MinIO).
    - Delete playlist: check tracks remain but playlist gone.
    - Feature flag: set ENABLE_LIBRARY=false, restart dev, verify that Library link is gone and trying direct /library route yields 404 or redirect to home.
    - All automated tests pass again (maybe update them if changes).
- **Docs:**
    - Update any documentation to reflect search improvements (perhaps mention fuzzy search or suggestions now implemented).
    - If Saved search not implemented, maybe note as future work.
    - Update ROADMAP.md status for milestone 1 as completed, maybe adding any risk mitigations discovered.
    - Possibly create an ADR if some decisions needed logging (like "Used simple JS sort for search vs PostgreSQL full-text due to timeframe").
- **Telemetry:**
- Could add an event for search queries (noting query string length or so) for analytics in future. But since no analytics stack integrated, might skip.
- At least ensure new routes (delete track etc.) use withSpan to trace them. For example, wrap the DELETE track logic with `withSpan("studio-web-api", "library.deleteTrack", ...)`.
- If track play events were to be recorded (not doing in this PR likely).
- **Definition of Done:**
- [x] Search results are now ordered in a way that feels relevant. If user searches for a song title or a distinctive part of it, that song appears at or near the top of results (assuming it matches).
- [x] Searching with a small typo or singular/plural variation either still finds the item (if within our tolerance) or provides a "Did you mean ___?" suggestion if nothing found and a close match exists.
- [x] If "Did you mean" is offered, it is correctly suggesting an existing track/artist and clicking or using it yields results.

- [x] The search bar and filters are still responsive in real-time or on form submit, with no significant lag.
- [x] No results case is handled gracefully (e.g., "No tracks found" message).
- [x] (Optional) Saved searches: if implemented, user can save a search and recall it easily. If not implemented, it's documented as deferred.
- [x] All library UI flows from PR 1C still work (import, edit, playlists).
- [x] New ability: user can delete a track from their library, with confirmation. The track and its file are removed permanently.
- [x] User can delete a playlist, with confirmation, and it is removed.
- [x] Minor UI tweaks done (like confirmations, toasts, styling improvements).
- [x] The Library feature can be disabled via env flag and this will hide all related UI and prevent access, without errors.
- [x] All tests (unit/integration/e2e) pass and coverage for new features is adequate.
- [x] Documentation is updated to reflect the final state of the Music Library feature.
- **Out of Scope:**
- Integration of any ML or advanced search (embedding, etc.) – that's for milestone 3.
- Multi-criteria advanced filtering (min BPM, by key, etc.) – we only filter by text and maybe simple genre string matching.
- Suggesting multiple alternatives in "did you mean" – we likely just pick one best.
- Comprehensive input normalization (we won't e.g. handle accents vs no accents differences in search beyond what ILIKE does).
- Saved searches if not done, obviously.
- Real-time indexing – for large libraries, our method might not scale extremely well, but it's okay for now.
- Anything not finished in prior PRs due to scope we either handled or explicitly deferred (like waveform display remains not done, etc.).
- Possibly some final cross-browser testing (we assume modern browsers all fine).
- Internationalization (UI text is English only, which is fine).
- Additional polish like drag-drop reorder playlists or editing multiple tracks at once (not needed).

---

1 2 4 5 6 10 14 16 32 33 155 156 232 README.md
https://github.com/danielwellz/omnisonic/blob/1d70747496bc11a1689d1a85c665c8c0c45fd5b8/README.md

3 15 17 18 31 39 140 141 154 171 phase-3-insight.md
https://github.com/danielwellz/omnisonic/blob/1d70747496bc11a1689d1a85c665c8c0c45fd5b8/docs/specs/phase-3-insight.md

7 8 135 146 147 159 160 217 index.ts
https://github.com/danielwellz/omnisonic/blob/1d70747496bc11a1689d1a85c665c8c0c45fd5b8/services/realtime-gateway/src/index.ts

9 13 26 30 40 55 82 128 129 130 134 136 144 150 233 phase-1-studio.md
https://github.com/danielwellz/omnisonic/blob/1d70747496bc11a1689d1a85c665c8c0c45fd5b8/docs/specs/phase-1-studio.md

11 12 152 153 162 index.ts
https://github.com/danielwellz/omnisonic/blob/1d70747496bc11a1689d1a85c665c8c0c45fd5b8/services/export-worker/src/index.ts

19 20 21 22 42 54 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 145 158 236
schema.prisma
https://github.com/danielwellz/omnisonic/blob/1d70747496bc11a1689d1a85c665c8c0c45fd5b8/packages/db/prisma/schema.prisma

23 24 37 38 65 66 68 69 72 73 74 75 76 77 87 index.ts
https://github.com/danielwellz/omnisonic/blob/1d70747496bc11a1689d1a85c665c8c0c45fd5b8/packages/storage/src/index.ts

25 52 56 58 60 61 62 63 64 67 70 71 86 239 route.ts
https://github.com/danielwellz/omnisonic/blob/1d70747496bc11a1689d1a85c665c8c0c45fd5b8/apps/studio-web/app/api/upload/route.ts

27 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 165 index.ts
https://github.com/danielwellz/omnisonic/blob/1d70747496bc11a1689d1a85c665c8c0c45fd5b8/services/graph-api/src/index.ts

28 29 34 36 167 168 230 env-and-secrets.md
https://github.com/danielwellz/omnisonic/blob/1d70747496bc11a1689d1a85c665c8c0c45fd5b8/docs/operations/env-and-secrets.md

35 43 44 45 46 47 48 49 50 51 139 164 auth.ts
https://github.com/danielwellz/omnisonic/blob/1d70747496bc11a1689d1a85c665c8c0c45fd5b8/apps/studio-web/lib/auth.ts

41 151 session.spec.ts
https://github.com/danielwellz/omnisonic/blob/1d70747496bc11a1689d1a85c665c8c0c45fd5b8/apps/studio-web/e2e/session.spec.ts

53 57 83 84 85 132 133 163 245 route.ts
https://github.com/danielwellz/omnisonic/blob/1d70747496bc11a1689d1a85c665c8c0c45fd5b8/apps/studio-web/app/api/upload/[id]/route.ts

59 81 241 242 243 244 upload-panel.tsx
https://github.com/danielwellz/omnisonic/blob/1d70747496bc11a1689d1a85c665c8c0c45fd5b8/apps/studio-web/components/session/upload-panel.tsx

78 79 80 91 92 route.ts
https://github.com/danielwellz/omnisonic/blob/1d70747496bc11a1689d1a85c665c8c0c45fd5b8/apps/studio-web/app/api/upload/[id]/file/route.ts

88 89 90 131 route.ts
https://github.com/danielwellz/omnisonic/blob/1d70747496bc11a1689d1a85c665c8c0c45fd5b8/apps/studio-web/app/api/upload/list/route.ts

137 138 166 route.ts
https://github.com/danielwellz/omnisonic/blob/1d70747496bc11a1689d1a85c665c8c0c45fd5b8/apps/studio-web/app/api/export/[id]/download/route.ts

142 148 149 usePresence.ts
https://github.com/danielwellz/omnisonic/blob/1d70747496bc11a1689d1a85c665c8c0c45fd5b8/apps/studio-web/hooks/usePresence.ts

143 170 package.json
https://github.com/danielwellz/omnisonic/blob/1d70747496bc11a1689d1a85c665c8c0c45fd5b8/apps/insight-web/package.json

157 169 172 173 174 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 218 219 220 221 222 223 224 225 226 227 228 229 231 234 235 237 238 240 246 247 Library-first Omnisonic acceleration roadmap with Codex prompts.txt
file://file_000000000b34722fa3d3c49817d29603

161 package.json
https://github.com/danielwellz/omnisonic/blob/1d70747496bc11a1689d1a85c665c8c0c45fd5b8/services/upload-cleaner/package.json

175 roadmap.md
https://github.com/danielwellz/omnisonic/blob/1d70747496bc11a1689d1a85c665c8c0c45fd5b8/docs/roadmap.md