RFC UnReel

- **Start Date:** 2025-11-12
- **RFC PR:** (leave this empty)
- **Project Issue:** (leave this empty)

Summary

This RFC proposes the complete backend architecture for `UnReel`, an AI-powered service designed to analyze short-form videos. The system will be built as a Node.js API using the **Fastify** framework and written entirely in **TypeScript**.

The core functionality involves two main endpoints:

1. A synchronous `/analyze` endpoint that ingests a video URL, uses `yt-dlp` to fetch the video and its metadata (like the caption), and `ffmpeg` to extract audio and video frames.
2. All media and text are then sent to the **Gemini API** for multimodal analysis.
3. The results are saved to a **PostgreSQL** database (via **Prisma**) and a structured JSON response (containing a summary, translation, identified resources, etc.) is returned to the client.
4. A subsequent `/chat` endpoint allows a user to have a follow-up conversation based on the context of the initial analysis.

This architecture centralizes all heavy processing on the server, providing a clean, powerful, and stateless API for any mobile or web client.

Basic example

This proposal defines two new API endpoints. The client (React Native or Flutter app) will interact with the backend via these JSON contracts.

1. `POST /api/v1/analyze`

This endpoint initiates a new video analysis. It is a slow, synchronous request and the client **must** implement a loading indicator and handle potential timeouts.

Request Body:

```json
{
  "url": "https://www.instagram.com/reel/Cxyz.../"
}
```

Success Response (200 OK):

```json
{
  "analysisId": "clq5p1y0q0000abcde1234567",
  "originalUrl": "https://www.instagram.com/reel/Cxyz.../",
  "status": "completed",
  "metadata": {
    "title": "My trip to Tokyo!",
    "uploader": "travelvlogger123",
    "caption": "Just got back from the most amazing trip. #tokyo #japan #travel"
  },
  "content": {
    "summary": "The video is a travel montage of Tokyo, Japan. It shows clips of Shibuya Crossing, the Senso-ji Temple, and various food stalls.",
    "translation": "Das Video ist eine Reisemontage von Tokio, Japan. Es zeigt Clips von der Shibuya-Kreuzung, dem Senso-ji-Tempel und verschiedenen Essensständen.",
    "keyTopics": ["Tokyo", "Japan", "Travel", "Food", "Shibuya Crossing"],
    "mentionedResources": [
      {
        "type": "Location",
        "name": "Shibuya Crossing"
      },
      {
        "type": "Location",
        "name": "Senso-ji Temple"
      },
      {
        "type": "Song",
        "name": "Tokyo Drift - Teriyaki Boyz"
      }
    ]
  },
  "fullTranscript": "Hey everyone, just got back from my trip to Tokyo... [full audio transcript] ...",
  "createdAt": "2025-11-12T09:30:00.000Z"
}
```

2. `POST /api/v1/chat`

This endpoint allows a user to ask follow-up questions about a *previously analyzed* video.

Request Body:

```json
{
  "analysisId": "clq5p1y0q0000abcde1234567",
  "message": "What was the name of the temple shown?"
}
```

Success Response (200 OK):

```json
{
  "reply": "The temple shown in the video was the Senso-ji Temple in Asakusa."
}
```

Motivation

The primary motivation is to solve a common set of user frustrations with short-form video content:

1. **Context Gap:** Users frequently encounter videos (memes, trends, complex topics) without the necessary context to understand them.
2. **Language Barrier:** A vast amount of content is inaccessible to users who don't speak the language of the video.
3. **Information Retrieval:** Users struggle to find a song, product, book, or location mentioned or shown in a video, requiring tedious manual searching.

This backend architecture is motivated by the need to:

- **Provide a Centralized "Brain":** Create a single, authoritative API that does all the complex media processing and AI analysis. The mobile client remains "dumb" and only handles UI.
- **Ensure Scalability:** By designing a stateless API, we can scale the backend compute layer horizontally as user load increases.
- **Enable Persistence:** Storing analysis results and chat history in a PostgreSQL database allows users to revisit past analyses and enables the stateful "chat" feature.
- **Decouple Frontend from Backend:** A clearly defined API contract allows the mobile (React Native) and backend (Node.js) teams to work in parallel.

Detailed design

This design is broken down into the file structure, database schema, and the implementation details of each core module.

1. System Architecture

A high-level overview of the request flow:

1. **Client App** sends a `POST /api/v1/analyze` request with a URL.
2. **Fastify Server** receives the request.
3. **Analysis Service** orchestrates the workflow:
   a. Calls **Media Service** to download the video/caption (`yt-dlp`) and extract audio/frames (`ffmpeg`). All files are saved to a temporary directory.
   b. Calls **AI Service** with the file paths and text caption.
   c. **AI Service** (Gemini) returns a structured JSON string.
   d. **Analysis Service** parses the AI result.
   e. Calls **Prisma Service** to save the final `Analysis` and `ChatMessage` (the summary) to the **PostgreSQL DB**.
   f. Calls **Media Service** to clean up the temporary files.
4. **Fastify Server** returns the newly created `Analysis` object as JSON to the **Client App**.

2. File Structure (Monorepo)

A modular, scalable, and type-safe file structure is essential.

```
unreel-api/
├── prisma/
│   ├── schema.prisma             # Prisma schema (our "database truth")
│   └── migrations/               # Generated SQL migrations
├── src/
│   ├── modules/                  # Business logic modules (features)
│   │   ├── analysis/
│   │   │   ├── analysis.controller.ts # Handles HTTP req/res
│   │   │   ├── analysis.service.ts    # Orchestrates the analysis
│   │   │   ├── analysis.routes.ts     # Defines the /analyze endpoint
│   │   │   └── analysis.schema.ts     # Zod schemas for validation
│   │   └── chat/
│   │       ├── chat.controller.ts
│   │       ├── chat.service.ts
│   │       ├── chat.routes.ts
│   │       └── chat.schema.ts
│   ├── services/                   # Shared, re-usable services
│   │   ├── prisma.service.ts       # Singleton for Prisma client
│   │   ├── media.service.ts        # Wrapper for yt-dlp & ffmpeg
│   │   └── ai.service.ts           # Wrapper for Gemini API
│   ├── plugins/                    # Fastify plugins
│   │   ├── env.plugin.ts           # Loads and validates .env
```

```
│   │   └── swagger.plugin.ts        # Auto-generates API docs
│   ├── app.ts                       # Fastify server setup (plugins, routes)
│   └── server.ts                    # Entry point (starts the server)
├── .env                             # Environment variables (API keys)
├── .gitignore
├── package.json
└── tsconfig.json
```

3. Database Schema (`prisma/schema.prisma`)

The schema defines our data models. We'll use `Json` types for flexibility with AI output.

```
// This is your Prisma schema file,
// learn more about it in the docs: https://pris.ly/d/prisma-schema

generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

model User {
  id        String     @id @default(cuid())
  email     String     @unique
  createdAt DateTime   @default(now())
  analyses  Analysis[] // A user can have many analyses
}

model Analysis {
  id        String     @id @default(cuid())
  originalUrl String
  status    String     @default("processing") // e.g., "processing", "completed", "failed"

  // Metadata from yt-dlp
  title     String?
  uploader  String?
  caption   String?
```

```
  // AI-generated content
  summary     String?
  translation String?
  fullTranscript String?
  keyTopics   Json?      // Will store: string[]
  mentionedResources Json? // Will store: {type: string, name: string}[]

  createdAt DateTime   @default(now())
  updatedAt DateTime   @updatedAt

  // Relation to a user (optional for MVP)
  user   User?   @relation(fields: [userId], references: [id])
  userId String?

  // Relation to chat messages
  chatMessages ChatMessage[]
}

model ChatMessage {
  id        String   @id @default(cuid())
  role      String   // "user" or "model"
  content   String
  createdAt DateTime @default(now())

  // Relation to the analysis
  analysis   Analysis @relation(fields: [analysisId], references: [id], onDelete: Cascade)
  analysisId String

  @@index([analysisId, createdAt])
}
```

4. Core Module Implementation (Code Snippets)

Here is the high-level code for the key modules.

`src/app.ts` (Fastify Server Setup)

This file bootstraps the Fastify server, registers plugins, and attaches our feature modules.

```typescript
import Fastify, { FastifyInstance } from 'fastify';
import envPlugin from './plugins/env.plugin';
import swaggerPlugin from './plugins/swagger.plugin';
import analysisRoutes from './modules/analysis/analysis.routes';
import chatRoutes from './modules/chat/chat.routes';

export function buildServer(): FastifyInstance {
  const server = Fastify({
    logger: true,
  });

  // Register plugins
  server.register(envPlugin);
  server.register(swaggerPlugin);

  // Register feature routes
  server.register(analysisRoutes, { prefix: '/api/v1/analyze' });
  server.register(chatRoutes, { prefix: '/api/v1/chat' });

  return server;
}
```

src/modules/analysis/analysis.routes.ts

This file defines the API schema and links the route to the controller.

```typescript
import { FastifyInstance } from 'fastify';
import { AnalysisController } from './analysis.controller';
import { $ref } from './analysis.schema'; // Assumes Zod schemas are defined

async function analysisRoutes(server: FastifyInstance) {
  const controller = new AnalysisController();

  server.post(
    '/',
    {
      schema: {
        body: $ref('analyzeRequestSchema'),
        response: {
          200: $ref('analyzeResponseSchema'),
```

```
        },
      },
    },
    controller.createAnalysisHandler,
  );
}

export default analysisRoutes;
```

`src/modules/analysis/analysis.service.ts` (The Orchestrator)

This is the most important service, coordinating all the other services.

```typescript
import { PrismaService } from '../../services/prisma.service';
import { MediaService } from '../../services/media.service';
import { AiService, AiAnalysisResult } from '../../services/ai.service';
import { Analysis } from '@prisma/client';
import { promises as fs } from 'fs';

export class AnalysisService {
  private prisma = new PrismaService();
  private media = new MediaService();
  private ai = new AiService();

  public async createAnalysis(url: string): Promise<Analysis> {
    let mediaData;
    try {
      // 1. Process media
      console.log('Processing media...');
      mediaData = await this.media.processVideo(url);

      // 2. Get AI analysis
      console.log('Getting AI analysis...');
      const aiResult = await this.ai.getAnalysis({
        audioPath: mediaData.audioPath,
        imagePaths: mediaData.framePaths,
        caption: mediaData.metadata.caption,
      });

      // 3. Save to database
      console.log('Saving to database...');
```

```
      const analysis = await this.prisma.analysis.create({
        data: {
          originalUrl: url,
          status: 'completed',
          title: mediaData.metadata.title,
          uploader: mediaData.metadata.uploader,
          caption: mediaData.metadata.caption,
          summary: aiResult.summary,
          translation: aiResult.translation,
          fullTranscript: aiResult.fullTranscript,
          keyTopics: aiResult.keyTopics,
          mentionedResources: aiResult.mentionedResources,
        },
      });

      // 4. Save initial chat messages
      await this.prisma.chatMessage.createMany({
        data: [
          {
            analysisId: analysis.id,
            role: 'user',
            content: `Analyze this video: ${url}`,
          },
          {
            analysisId: analysis.id,
            role: 'model',
            content: `Here is the summary: ${aiResult.summary}`,
          },
        ],
      });

      return analysis;
    } catch (error) {
      console.error('Analysis failed:', error);
      // TODO: Save a "failed" status to DB
      throw new Error('Analysis process failed.');
    } finally {
      // 5. Cleanup temp files
      if (mediaData?.tempDir) {
        console.log('Cleaning up temp files...');
        await fs.rm(mediaData.tempDir, { recursive: true, force: true });
      }
    }
  }
}
```

`src/services/media.service.ts`

This service wraps the `yt-dlp` and `ffmpeg` CLI commands.

```typescript
import { exec } from 'child_process';
import { promisify } from 'util';
import { promises as fs } from 'fs';
import path from 'path';
import os from 'os';

const execAsync = promisify(exec);

export class MediaService {
  public async processVideo(url: string) {
    const tempDir = await fs.mkdtemp(path.join(os.tmpdir(), 'unreel-'));
    const videoPath = path.join(tempDir, 'video.mp4');
    const audioPath = path.join(tempDir, 'audio.mp3');

    // 1. Download video and caption
    const metaPath = path.join(tempDir, 'video.info.json');
    await execAsync(
      `yt-dlp -f "bestvideo[ext=mp4]+bestaudio[ext=m4a]/best[ext=mp4]/best" -o "${videoPath}" --write-info-json "${url}"`
    );

    const metadataJson = await fs.readFile(metaPath, 'utf-8');
    const metadata = JSON.parse(metadataJson);

    // 2. Extract audio
    await execAsync(
      `ffmpeg -i "${videoPath}" -vn -acodec libmp3lame -q:a 2 "${audioPath}"`
    );

    // 3. Extract frames (1 frame every 5 seconds)
    const framesDir = path.join(tempDir, 'frames');
    await fs.mkdir(framesDir);
    await execAsync(
      `ffmpeg -i "${videoPath}" -vf "fps=1/5" "${framesDir}/frame-%03d.png"`
    );

    const frameFiles = await fs.readdir(framesDir);
    const framePaths = frameFiles.map(f => path.join(framesDir, f));

    return {
```

```
      tempDir,
      videoPath,
      audioPath,
      framePaths,
      metadata: {
        title: metadata.title,
        uploader: metadata.uploader,
        caption: metadata.description,
      },
    };
  }
}
```

src/services/ai.service.ts

This service wraps the Gemini API, handling the multimodal prompt.

TYPESCRIPT

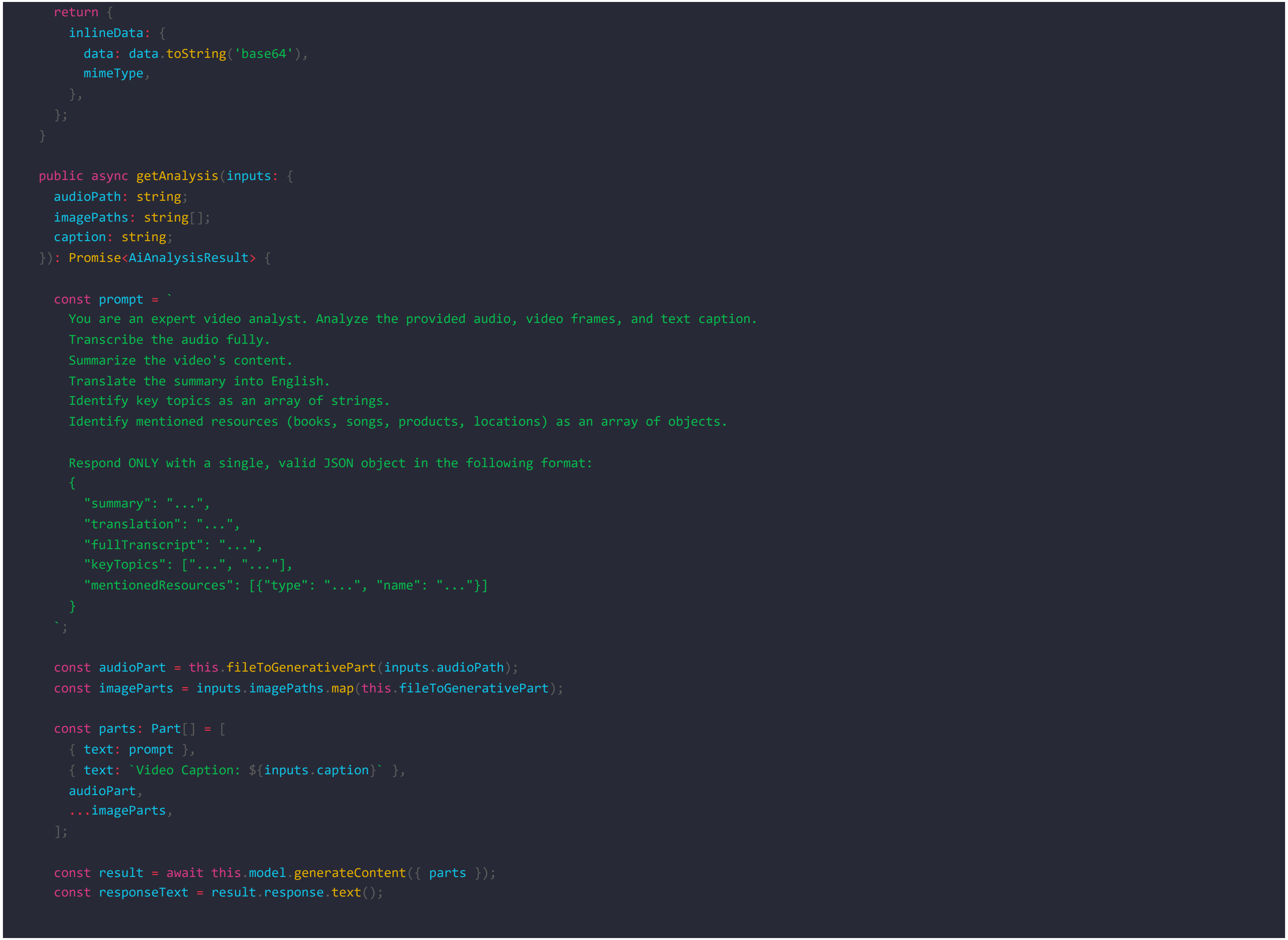
```typescript
import { GoogleGenerativeAI, Part } from '@google/generative-ai';
import { promises as fs } from 'fs';
import mime from 'mime-types';

export interface AiAnalysisResult {
  summary: string;
  translation: string;
  fullTranscript: string;
  keyTopics: string[];
  mentionedResources: { type: string; name: string }[];
}

export class AiService {
  private genAI: GoogleGenerativeAI;
  private model;

  constructor() {
    this.genAI = new GoogleGenerativeAI(process.env.GEMINI_API_KEY!);
    this.model = this.genAI.getGenerativeModel({ model: 'gemini-1.5-flash' });
  }

  private fileToGenerativePart(path: string): Part {
    const data = fs.readFileSync(path);
    const mimeType = mime.lookup(path) || 'application/octet-stream';
```



```
    return {
      inlineData: {
        data: data.toString('base64'),
        mimeType,
      },
    };
  }

  public async getAnalysis(inputs: {
    audioPath: string;
    imagePaths: string[];
    caption: string;
  }): Promise<AiAnalysisResult> {

    const prompt = `
      You are an expert video analyst. Analyze the provided audio, video frames, and text caption.
      Transcribe the audio fully.
      Summarize the video's content.
      Translate the summary into English.
      Identify key topics as an array of strings.
      Identify mentioned resources (books, songs, products, locations) as an array of objects.

      Respond ONLY with a single, valid JSON object in the following format:
      {
        "summary": "...",
        "translation": "...",
        "fullTranscript": "...",
        "keyTopics": ["...", "..."],
        "mentionedResources": [{"type": "...", "name": "..."}]
      }
    `;

    const audioPart = this.fileToGenerativePart(inputs.audioPath);
    const imageParts = inputs.imagePaths.map(this.fileToGenerativePart);

    const parts: Part[] = [
      { text: prompt },
      { text: `Video Caption: ${inputs.caption}` },
      audioPart,
      ...imageParts,
    ];

    const result = await this.model.generateContent({ parts });
    const responseText = result.response.text();
```

```
    try {
      // Clean the response text in case Gemini adds markdown backticks
      const jsonString = responseText.replace(/```json/g, '').replace(/```/g, '').trim();
      return JSON.parse(jsonString) as AiAnalysisResult;
    } catch (error) {
      console.error('Failed to parse Gemini JSON response:', responseText);
      throw new Error('AI response was not valid JSON.');
    }
  }
}
```

Drawbacks

- **Performance:** The `/analyze` endpoint will be **very slow** (30-90 seconds) due to video downloading, `ffmpeg` processing, and AI API latency. This creates a poor synchronous UX. The MVP will accept this, but a V2 must move to an asynchronous (queue-based) architecture.
- **Cost:** Multimodal AI calls are expensive. Processing audio *and* multiple video frames for every request will incur significant costs from Google Cloud.
- **Brittleness:** `yt-dlp` is a web scraper. If Instagram, YouTube, or TikTok change their front-end HTML/API, `yt-dlp` will break, and our service will be down until it's patched. This is a high maintenance risk.
- **Resource Intensive:** `ffmpeg` is CPU-intensive. Running this on a single server will not scale. High-volume usage will require moving this processing to dedicated worker instances or serverless functions (e.g., AWS Lambda with an EFS layer for `ffmpeg`).
- **Temporary Storage:** The design uses `/tmp` for media files. This is not suitable for a production environment, as files are ephemeral. A V2 must use a proper blob store like **AWS S3** or **GCS** for staging these files.

Alternatives

1. **Asynchronous Architecture (The V2 Plan):** Instead of a slow synchronous request, the `/analyze` endpoint could immediately return a `202 Accepted` response with a `jobId`. The client would then poll a `/status/{jobId}` endpoint or receive a webhook.
   - *Pros:* Far better UX.
   - *Cons:* Massively more complex. Requires a message queue (e.g., RabbitMQ, BullMQ), worker services, and state management. This is rejected for the MVP due to complexity.
2. **Separate AI Models:** Instead of one multimodal (Gemini) call, we could use multiple, cheaper models: `Whisper` for transcription, `Tesseract` (or a vision API) for OCR, and `GPT-3.5-Turbo` for summarization.
   - *Pros:* Potentially cheaper.
   - *Cons:* More complex orchestration. Multiple API calls, multiple points of failure. The simplicity of a single multimodal call is preferred for the MVP.
3. **Self-Hosting Models:** We could self-host open-source models (like Whisper or LLaVA).
   - *Pros:* Drastic cost savings at scale.
   - *Cons:* Prohibitive operational complexity. Requires managing GPU-powered servers, scaling inference endpoints, etc. This is not feasible for an MVP.

Adoption strategy

This is a new backend service, so it does not break any existing applications. The adoption strategy is focused on the client-side (the React Native app).

1. **API Contract:** This RFC and the auto-generated **Swagger/OpenAPI** documentation (from the `swagger.plugin.ts`) will serve as the single source of truth for the mobile development team.
2. **Staging Environment:** A staging version of this API will be deployed for the mobile app to be tested against.
3. **Client-Side Implementation:** The mobile app will be built to spec against the two defined endpoints (`/analyze`, `/chat`). The most critical part of adoption is the client-side handling of the **long response time** for the `/analyze` endpoint, which must be built with a robust loading/polling state.

How we teach this

This backend introduces a simple but powerful pattern for the mobile developers.

- **Names & Terminology:**
  - **"Analysis"**: The core data object created by the `/analyze` call.
  - **"Analyze-then-Chat"**: The core user flow. You *must* create an "Analysis" to get an `analysisId` before you can use the `/chat` endpoint.
  - **"Slow Endpoint"**: We will explicitly teach that `/analyze` is a "slow endpoint" and must be treated as an asynchronous job from the user's perspective (i.e., show a full-screen loading/processing UI).
- **Documentation:** The primary teaching tool will be the **interactive Swagger (OpenAPI) documentation** generated by the server. Mobile developers can make test calls directly from their browser.
- **Teaching New Developers:** A new mobile developer will be taught:
  1. Your app is a "dumb client." All logic lives on the API.
  2. Your first call is `POST /analyze` with a URL. Show a loading spinner.
  3. When you get the JSON response, display it and **save the** `analysisId`.
  4. For all follow-up messages, send the `analysisId` and the user's message to `POST /chat`.

Unresolved questions

- **Granular Error Handling:** What is the JSON response for a *failed* analysis? (e.g., URL is private, `yt-dlp` fails, AI times out). A proper error schema (`{ "code": "...", "message": "..." }`) needs to be designed.
- **Authentication:** The current design is completely open and unauthenticated. A V2 will need to implement user authentication (e.g., JWT) and link the `Analysis` model to the `User` model via the `userId` field.
- **Production File Storage:** We are using `/tmp`. This is a blocker for production. We must replace the local `fs` calls with a proper blob storage service (like **AWS S3**). The `media.service.ts` would be refactored to upload files to S3 and pass S3 URIs to the AI service.
- **Rate Limiting:** To prevent abuse and control costs, we must add rate limiting, likely on a per-IP basis for the MVP and per-user once authentication is added.