ChatGPT

Product Requirements Document (PRD): AI-Powered Café Food Video Generator

Overview

This React Native Android app enables café clients to create cinematic promotional food videos using AI. Users log in (email/password or Google via Supabase Auth), upload product photos, choose from predefined *video styles*, and receive a high-quality video made of **three focused shots** stitched together. The back end uses Google's Gemini Video API (Veo 3.1) to generate each shot and Remotion to programmatically assemble them. Access is **subscription-based only** (no per-use or in-app purchases). Internal admin users have a separate dashboard to manage users, styles, and monitor processing.

The system uses Supabase for authentication, database, and file storage (images/videos), integrates Stripe or Razorpay for recurring billing, and orchestrates AI generation workflows on a Node.js server. The app **excludes** analytics, one-off purchases, and UI/branding polish; it focuses on core functionality and reliability.

Goals and Scope

- **Client Goal:** Empower café owners to easily create cinematic product videos from photos.
- **Admin Goal:** Provide internal staff with tools to manage clients, subscription plans, and video templates.
- **Core Features:** User login, photo upload, style selection, AI video generation (3 segments), subscription handling, admin management.
- **Exclusions:** No analytics dashboards, no in-app purchases, minimal UI styling (focus on functionality), no third-party integrations beyond those specified.

User Roles

- **Client (Café Owner/User):** Registers or logs in (email/password or Google). Manages subscription plan. Uploads product images (e.g. dishes or drinks). Selects a video style/template. Requests and receives AI-generated videos. Views history of generated videos.

- **Admin (Internal):** Logs into an admin portal (distinct UI or via role-based access). Manages user accounts (view/edit clients and their subscription status). Manages video style templates (create/edit predefined prompts). Views all video generation requests and statuses (monitor, retry failures). Manages subscription plans and payment configurations.

Functional Requirements

1. Authentication & User Management

- **User Signup/Login:** Users can sign up or log in via email/password or Google OAuth using Supabase Auth [1] [2]. All user sessions are managed with JSON Web Tokens (JWTs) [3].

- **User Profiles:** Store basic user info (email, name, role) in Supabase DB. Assign roles (`client` or `admin`) in user metadata or a roles table. Use Supabase Row-Level Security (RLS) to enforce access (e.g. only admin role can access admin APIs) [3].
- **Google Sign-In:** Enable "Sign in with Google" via Supabase social auth [2]. This simplifies login for clients and admins.

2. Subscription Management

- **Payment Providers:** Integrate **Stripe Billing** (global) and/or **Razorpay** (India) for subscriptions. Use Stripe's subscription API for recurring plans [4] and Razorpay's subscription API for region-specific clients [5]. Only active subscribers can generate videos.
- **Plans:** Define at least one subscription plan (e.g. monthly). The client must subscribe via Stripe or Razorpay before using the app. Support free trials or introductory periods if needed.
- **Subscription Flow:**
- User chooses a plan in-app.
- Frontend calls backend to create a customer and subscription (via Stripe or Razorpay API).
- Handle payment collection and webhook events: on successful subscription, mark user as "active" in the database; on cancellation or failure, revoke generation privileges.
- The app checks subscription status at login or before generation requests to enforce access.
- **Billing Webhooks:** Configure webhooks to listen for Stripe/Razorpay events (e.g. `invoice.paid`, `customer.subscription.deleted`). Update user records accordingly (enable/disable service) [6] [5].

3. Product Image Upload

- **Upload Interface:** Allow clients to upload up to three product images per video request. Images should be high-resolution photographs of food/drinks.
- **Validation:** Accept common image formats (JPEG/PNG). Enforce file size limits (e.g. <= 5MB) and dimensions (e.g. at least 720px width). Reject non-food images.
- **Storage:** Store uploaded images in Supabase Storage (Files bucket) [7] [8]. Use a secure, authenticated upload flow (Supabase provides APIs and CDN). Record the storage URLs in the database for later use. Supabase Storage is suitable for media (images/videos) and offers CDN delivery for performance [8].

4. Video Style Templates

- **Predefined Styles:** Offer a set of cinematic style presets (e.g. *"Cinematic Close-up," "Vibrant Product Shot," "Slow-Motion Artistic"*). Each style corresponds to specific prompt text guiding the AI model (e.g., a "Cinematic" style might use prompts emphasizing dramatic lighting and camera movement).
- **Prompt Guidance:** For each style, define text prompts or parameters. The backend will combine these with the user's images when calling the Gemini API.
- **Admin Management:** Admins can add or edit style templates via the admin UI. Each template includes a name and AI prompt text. These are stored in the database.

5. Video Generation Workflow

- **Overview:** When a user requests a video, the system generates **three separate short clips** (each focusing on different aspects of the product) and stitches them into one video.
- **Gemini (Veo 3.1) API:** Use Google's GenAI Gemini API (Veo 3.1 model) for text-to-video generation [9]. Veo 3.1 can produce high-fidelity 8-second videos at up to 4K resolution [9]. It supports "image-based direction," allowing up to 3 reference images to guide content [10].

- **Shot Generation:** For each of the three shots:
- Compose a prompt using the selected style template and any custom text (e.g. product name or descriptors).
- Include the relevant user-uploaded image(s) as reference frames. The Gemini API allows sending up to three reference images per request [10]. Use each uploaded photo accordingly.
- Call the Gemini generateVideos endpoint (e.g. `client.models.generateVideos`) with the prompt, model `"veo-3.1-generate-preview"`, and any options (e.g. resolution 1080p).
- Poll the operation until completion; download the returned video asset. (This is asynchronous – the backend should handle polling and retries.)
- **Remotion Stitching:** Once all three clips are generated:
- Use **Remotion** (a React-based video framework [11]) on the server to assemble the clips. Remotion lets you write video compositions in React as code [11].
- In a Remotion composition (Node.js environment), import the three generated video files and place them sequentially. Use the `<Series>` component to stitch scenes end-to-end [12]. Each `<Series.Sequence>` holds one clip.
- Apply any transitions or text overlay (e.g. fade between shots, title card) as needed via Remotion's APIs.
- Render the final video (e.g. MP4). Remotion can run headlessly on a server or cloud function.
- **Final Delivery:** Upload the final stitched video to Supabase Storage and record its URL and metadata (length, creation date) in the database. Notify the user in the app (and optionally via email/push) that the video is ready. The user can then stream or download the video from the app.

6. Video History & Management

- **User History:** Clients can view a list of their past video requests with status (processing, complete, failed) and timestamps. Each entry shows a preview thumbnail or title and a "Download" button for completed videos.
- **Regeneration:** If a video generation fails or the client is unhappy, they can either retry (if allowed by subscription) or contact support. (Optionally, allow one auto-retry per failure.)
- **Admin Oversight:** Admins can view all video jobs for all clients, filter by user, date, or status, and manually trigger a regeneration if needed.

7. Admin Dashboard Features

- **User Management:** List all client users with details: name, email, subscription status, signup date. Admins can deactivate/reactivate users, reset passwords, or impersonate for troubleshooting.
- **Subscription Oversight:** View summary of subscriptions (active, trialing, canceled) and revenue reports (via Stripe/Razorpay dashboards integration).
- **Style Template Management:** CRUD interface for video style templates (name, description, prompt text). Changes take effect immediately for new requests.
- **Video Jobs Monitoring:** Real-time view of video generation queue. Show recent jobs, status (pending, rendering, completed), and any errors. Provide logs or error messages from Gemini/Remotion for debugging.
- **System Settings:** Configure API keys (Gemini, Stripe, Razorpay) and environment variables. Possibly control limits (e.g. max concurrent jobs).

System Architecture

- **Frontend (Android App):** Built with React Native. Provides UI for clients to authenticate, manage subscription, upload images, select style, request videos, and view history. Communicates with backend via HTTPS (REST or GraphQL).
- **Backend API:** A Node.js/TypeScript server (could use Express, Next.js API routes, or serverless functions). Responsibilities:
- Handle client API calls (authentication tokens validated via Supabase, image uploads, generate requests, history).
- Interact with Supabase services (Auth, Postgres DB, Storage).
- Orchestrate video generation jobs (queueing, calling Gemini API, running Remotion, storing results).
- Manage Stripe/Razorpay integration (creating customers/subscriptions, handling webhooks).
- **Supabase Services:**
- **Auth:** Manages user login, JWT issuance, social auth (Google) [1] [2].
- **Database (Postgres):** Stores user profiles, subscription status (linked via webhook events), video job records (images URLs, status, output URL), and style templates.
- **Storage:** Holds user-uploaded images and final video files. Uses Files buckets with public or authenticated access as needed [7] [8].
- **AI & Video Services:**
- **Gemini API (Google GenAI):** Accessible via Google Cloud client libraries or REST. The backend uses the Gemini Video `generateVideos` endpoint to create clips [9] [10].
- **Remotion Engine:** Hosted on the same server or cloud (e.g. AWS Lambda/Google Cloud Run with Node). Takes generated video segments, composes them using React components (`<Series>`), and renders a final MP4 [12] [11].
- **Payment Systems:**
- **Stripe:** Backend uses Stripe SDK to create customers and subscriptions. Webhooks update Supabase DB.
- **Razorpay:** Similarly, backend calls Razorpay's Plans and Subscriptions APIs [5] and listens to their webhooks.
- **Admin Interface:** A web app (could be separate React project or integrated into the same codebase) for internal users. Connects to the same backend with admin-level authentication.

The following diagram (conceptual) illustrates the flow:

```
[React Native App]
      | (HTTPS/API calls)
[Backend Server (Node.js)]
      |-- Supabase Auth/DB/Storage
      |-- Google Gemini Video API
      |-- Remotion (video composer)
      |-- Stripe/Razorpay APIs
      `-- (Admin Web App)
```

Data Flows

1. **Login Flow:** User enters credentials or taps "Sign in with Google." The app sends credentials to Supabase Auth. On success, Supabase returns a JWT. The app stores this token for authenticated requests.

2. **Image Upload:** User selects photo(s); the app sends each file to Supabase Storage (via its SDK/API). Supabase returns a secure URL or path. The app sends these URLs to the backend to create a new video job record (status "Uploaded").
3. **Video Request:** User selects a style and taps "Generate." The app calls backend `/generate-video` with style ID and references to the uploaded images. The backend verifies subscription status, then creates a job entry and enqueues processing.
4. **AI Video Generation:** The backend service dequeues the job: for each of three shots, it calls Gemini's video API with the style prompt and images [9] [10]. After all clips are retrieved, it runs Remotion to stitch them [12] [11].
5. **Result Storage:** The final video file is uploaded to Supabase Storage. The job record is updated with the video URL and status "Complete."
6. **Notification:** The backend may notify the user (via push notification or email) that the video is ready. The app also polls or refreshes job history to show the completed video.
7. **Subscription Events:** Stripe/Razorpay send webhook events to a backend endpoint. The backend updates the user's subscription status in the database, enabling or disabling the "Generate" function in the app.

API Integrations

- **Supabase Auth & SDK:** For user login and database operations [1] [7]. The app uses the Supabase JS/React Native client; backend may use Supabase's Admin API or direct DB queries.
- **Stripe API:** To create subscriptions (`POST /customers`, `/subscriptions`) and handle billing. Use Stripe Webhook endpoints to receive events like `customer.subscription.deleted`. See Stripe docs on subscriptions [4].
- **Razorpay API:** To create Plans and Subscriptions for India-based clients (using their Subscriptions APIs [5]). Configure Razorpay webhooks similarly.
- **Google GenAI (Gemini) API:** Use Google's client library (`@google/genai`) or REST to call `generateVideos`. Sample code (Python/JS) is provided in Google docs [13] [14]. The backend should handle asynchronous operations by polling.
- **Remotion:** Include Remotion in the backend code. Use the `<Series>` component to concatenate clips [12]. Remotion's renderer outputs an MP4 which is saved.
- **Supabase Storage API:** To upload and serve image/video files. Follow Supabase Storage docs for file buckets [8].
- **Notifications (optional):** Could use Firebase Cloud Messaging (FCM) or email service to alert users when videos are done.

Assumptions and Constraints

- **Video Length & Quality:** Each shot is up to 8 seconds (Gemini Veo limit) and up to 1080p (Gemini supports 720p/1080p/4K [9]). The final video is ~24 seconds. Remotion rendering time will scale with length and effects.
- **Input Images:** Users upload high-quality photos; Gemini needs good references. Limit of 3 images per job (matching Gemini's "image-based direction" limit [10]).
- **Content:** The model is assumed to handle food imagery well. Prompt engineering ensures style and focus (e.g., "close-up of a latte being poured...").
- **Performance:** Video generation is compute-intensive. The backend should queue jobs and possibly limit concurrent jobs per user or overall. Expect minutes per video.
- **Subscriptions:** Must enforce subscription-only access. We assume at least one pricing plan, and revenue collection via Stripe/Razorpay. No free/per-video payments.
- **Scalability:** Supabase free tier has limits; for many clients, a paid Supabase plan or alternative may be needed. Similarly, Gemini API usage may be limited by Google's quotas.

- **Security:** All user data and media are private. Use HTTPS for all endpoints. Supabase Auth and RLS ensure users see only their own data. Admin routes are protected by role checks.
- **Offline:** The app requires internet for generation; no offline mode.
- **No Analytics/Tracking:** Per requirements, skip any analytics, user tracking, or marketing tools.
- **Compliance:** Ensure usage of AI content adheres to policy (food product images only).

Non-Functional Considerations

- **Reliability:** Provide status feedback (e.g., "Video processing...") and retry mechanisms. Handle failed AI calls gracefully (inform user, allow retry).
- **Maintainability:** Structure code modularly (authentication, payments, AI workflows, Remotion). Document APIs clearly for developers.
- **Extensibility:** New styles can be added by admins without redeploying code. The video pipeline should handle adding more segments if needed (though currently fixed to 3).
- **Usability (minimal):** The UI should be straightforward: login screen, upload photos, select style, and a clear "Generate" button. Admin UI lists tables of users and jobs.

Summary of Key Components

- **Authentication:** Supabase Auth (email/password, Google OAuth) [1] [2]. Roles enforced by RLS.
- **Storage:** Supabase Storage for images/videos [7] [8].
- **Database:** Supabase Postgres stores user profiles, subscriptions, style templates, video jobs.
- **AI Generation:** Google Gemini (Veo 3.1) API to create each video shot [9] [10]. Style-driven prompts.
- **Video Stitching:** Remotion on server to programmatically compose final video [12] [11].
- **Payments:** Stripe and/or Razorpay for subscription billing [4] [5].
- **Admin Portal:** Web dashboard for managing users, styles, and monitoring.

This PRD outlines all features, workflows, and integrations needed to implement the app. Developers should use the cited references for details on Gemini video generation [9] [10], Remotion composition [12] [11], Supabase auth/storage [1] [7], and subscription APIs [4] [5] to ensure correctness.

---

[1] [3] Auth | Supabase Docs
https://supabase.com/docs/guides/auth

[2] Login with Google | Supabase Docs
https://supabase.com/docs/guides/auth/social-login/auth-google

[4] [6] docs.stripe.com
https://docs.stripe.com/billing/subscriptions/overview

[5] Razorpay Docs
https://razorpay.com/docs/api/payments/subscriptions/?preferred-country=US

[7] [8] Storage | Supabase Docs
https://supabase.com/docs/guides/storage

[9] [10] [13] [14] Generate videos with Veo 3.1 in Gemini API | Google AI for Developers
https://ai.google.dev/gemini-api/docs/video

[11] Introducing Remotion | Remotion | Make videos programmatically
https://www.remotion.dev/blog/introducing-remotion

12 | Remotion | Make videos programmatically
https://www.remotion.dev/docs/series