Pixora AI – Smart Image Editing & Enhancement Platform

Project Documentation & Interview Guide

1. Project Overview

Pixora AI is a full-stack, AI-powered web application designed to provide users with a "Magic Studio" for intelligent image editing and enhancement. It operates on a Software-as-a-Service (SaaS) model, offering both a free-tier with usage limits and a "Pro" subscription plan for unlimited access.

The platform integrates multiple best-in-class technologies, including Next.js for the full-stack framework, NextAuth.js for authentication, Stripe for payments, Prisma for database management, ImageKit for high-performance image storage and real-time transformations, and the Google Gemini AI for advanced generative editing tasks.

The user experience is built to be seamless: a user uploads an image and can then apply a suite of AI-powered tools, from simple one-click background removal to complex, prompt-based generative edits.

2. Objective

The primary objective of Pixora AI is to democratize professional-grade image editing by leveraging the power of modern AI. It aims to solve several key user problems:

- **Complexity:** Traditional photo editors like Photoshop have a steep learning curve. Pixora AI provides one-click tools and natural language prompts to achieve complex results.
- **Speed:** AI-powered batch processing and real-time transformations save users significant time compared to manual editing.
- **Accessibility:** As a web-based platform, it's accessible from any device without requiring powerful hardware or software installation.
- **Monetization:** To build a viable business model around a high-demand service by implementing a secure and reliable subscription system.

3. Key Features

The application is centered around the "Magic Studio" and supported by a robust user and payment management system.

Core Editing Features:

- **Image Upload:** Secure, authenticated image upload system using ImageKit's client-side upload SDK.

- **AI Background Removal:** Two modes (standard and high-quality "Pro") for instantly removing image backgrounds.
- **AI Background Change:** Allows users to replace the background of an image using a text prompt (e.g., "a beach at sunset").
- **Generative AI Editing (Gemini):** A flagship feature where users can edit an image by providing a natural language prompt (e.g., "make the car red," "add a hat on the person"). This is powered by a custom backend API route (/api/gemini/edit) that communicates with the Google Gemini AI.
- **Generative Fill:** Expands the canvas of an image and intelligently fills the new areas based on existing content.
- **AI Upscale & Retouch:** Enhances image resolution (up to 4x) and automatically retouches photos to fix blemishes and improve quality.
- **Smart Cropping:** Intelligently crops images by detecting the main subject or focusing specifically on faces.

User & Platform Features:

- **Dual Authentication:** Secure user sign-up and sign-in using either **Google OAuth** or traditional **Email & Password** (Credentials), all managed by NextAuth.js.
- **SaaS Subscription Model:**
  - **Free Plan:** Grants users a limited number of AI edits (e.g., 3, as defined in the prisma/schema.prisma).
  - **Pro Plan:** A monthly subscription that grants unlimited usage (usageLimit: 999999).
- **Payment System:** Full integration with **Stripe** for one-time payments and recurring subscriptions.
  - **Stripe Checkout:** Redirects users to a Stripe-hosted checkout page to upgrade to Pro.
  - **Stripe Webhooks:** A dedicated API endpoint (/api/webhooks/stripe) listens for events from Stripe (e.g., checkout.session.completed, customer.subscription.updated) to automatically update the user's plan in the database.
- **Usage Tracking:** An API endpoint (/api/usage) tracks the number of edits a user has made and gates access based on their plan, prompting free users to upgrade when their limit is reached.
- **User Dashboard:**
  - **Profile Page (/profile):** Allows users to view their account details and current plan.
  - **Billing Page (/billing):** Allows users to manage their subscription (via Stripe Customer Portal).

4. Technologies Used

- **Framework: Next.js 15** (using App Router)
- **Frontend: React.js 19**, **Tailwind CSS**, **Framer Motion** (for animations)
- **Backend: Next.js API Routes**
- **Database: PostgreSQL**
- **ORM: Prisma ORM** (for type-safe database access)

- **Authentication: NextAuth.js v4** (with Google & Credentials Providers)
- **Payments: Stripe SDK** (for Checkout and Webhooks)
- **Image Storage & Processing: ImageKit.io** (for authenticated uploads and real-time transformations)
- **Generative AI: Google Gemini** (via @google/generative-ai SDK, specifically gemini-2.5-flash-image-preview)

5. Authentication Details

Authentication is handled by **NextAuth.js**, configured in src/lib/auth.ts. The system supports two methods:

1. **Google Provider (OAuth):**
    - A user signs in with Google.
    - The authOptions profile callback is triggered.
    - The system queries the **PostgreSQL** database via **Prisma** (prisma.users.findUnique).
    - If the user doesn't exist, a new user is created in the users table with the plan set to Free and a usageLimit of 3.
2. **Credentials Provider (Email/Password):**
    - A user signs up with an email and password.
    - The password would be hashed (e.g., using bcrypt) and stored securely in a password field in the users table (this field would be added to the Prisma schema).
    - When logging in, the authorize function of the CredentialsProvider is called.
    - This function finds the user by email and compares the provided password with the stored hash.

Session Management:

- Regardless of the provider, NextAuth.js uses a database session strategy.
- The jwt and session callbacks are used to augment the session token with custom data from our database, such as plan, usageCount, and usageLimit.
- **Protection:** The src/middleware.ts file protects routes like /profile and /billing, redirecting unauthenticated users.

6. Payment System

The payment system is built using **Stripe**.

- **Upgrading to Pro:**
    1. A user on the "Free" plan clicks an "Upgrade" button.
    2. This triggers a POST request to /api/create-checkout-session.
    3. This backend endpoint creates a new stripe.checkout.sessions.create instance, passing the user's email and userId (from the session token) into the session's metadata.
    4. The user is redirected to the Stripe-hosted checkout page.
- **Handling Successful Payments (Webhooks):**
    1. After a successful payment, Stripe sends an event to our /api/webhooks/stripe

endpoint.
2. This endpoint verifies the request signature using stripe.webhooks.constructEvent and the STRIPE_WEBHOOK_SECRET.
3. It listens for the checkout.session.completed event.
4. On this event, it retrieves the userId from the session's metadata.
5. It uses **Prisma** to update the corresponding user in the users table:
    - Sets plan to Pro.
    - Sets usageLimit to a high number (e.g., 999999) to represent "unlimited".
    - Saves the stripeCustomerId for future billing management.
6. It also handles other events like customer.subscription.updated or customer.subscription.deleted to manage plan downgrades or cancellations.

7. API Integration

ImageKit.io (Storage & Transformation)

- **Uploads:** ImageKit is a core part of the editing workflow.
    1. The client-side UploadZone component hits our backend at /api/upload-auth.
    2. This endpoint uses the ImageKit Node.js SDK to generate a secure, short-lived token (token, expire, signature).
    3. This token is passed back to the client, which then uploads the file *directly* to ImageKit. This avoids proxying the file through our server.
- **Transformations:** ImageKit's real-time transformation URL-based API is used for *most* of the AI features (Background Removal, Smart Crop, etc.).
    - The frontend (src/modules/editor/index.tsx) constructs a new ImageKit URL by appending transformation parameters (e.g., ?tr=e-bgremove).
    - This is extremely efficient, as ImageKit's CDN handles the AI processing and caching, offloading all computational work from our application.

Google Gemini (Generative AI)

- **Generative Edit:** For the "Gemini Edit" feature, a custom API route is used.
    1. The user provides a prompt (e.g., "make the car red").
    2. The client fetches the *original uploaded image* as a Base64 string.
    3. It sends this Base64 data and the text prompt to our backend at /api/gemini/edit.
    4. This endpoint uses the @google/generative-ai SDK to call the gemini-2.5-flash-image-preview model.
    5. It sends the original image and the user's prompt to the model.
    6. The model processes the request and returns a *new* image. The API route extracts the Base64 data of the new image and sends it back to the client.
    7. The client then displays this new Base64 image as the "processed" result.

8. Deployment & Hosting

The application is deployed on **Render** (https://pixora-ai-8zbh.onrender.com).

- **Web Service:** A Render Web Service is used to host the Next.js application. It's

connected to the project's GitHub repository for automatic builds and deployments on every push.

- **Database:** A **Render PostgreSQL** instance is used as the managed database. Render provides a secure, internal DATABASE_URL connection string that is directly added to the Web Service's environment variables.
- **Environment Variables:** All secrets (Database URL, NextAuth Secret, Stripe keys, ImageKit keys, Gemini API key) are stored securely in Render's "Environment Group" feature, which injects them into the Web Service at runtime.
- **Build Command:** The build command in Render is set to npx prisma generate && next build.
    - npx prisma generate: This is crucial. It runs first to generate the Prisma Client tailored for the PostgreSQL database *before* the Next.js app is built.
    - next build: This builds the production-ready Next.js application.

9. Challenges Faced

- **Database Connection Management:** Deploying a Next.js app with Prisma to a serverless or container-based platform like Render requires careful connection management. A naive setup can quickly exhaust the PostgreSQL connection limit. This is a common challenge solved by implementing **Prisma's Data Proxy** or a custom connection pooling solution.
- **State Management:** Managing the state of the "Magic Studio" is complex. This includes the original image URL, the currently applied transformations (as a list of strings), the prompt text, the final processed image URL, and the loading/job status.
- **Real-time Polling for AI:** ImageKit's AI transformations are not instantaneous. The applyEffect function implements a polling mechanism, sending HEAD requests to the new transformation URL until it gets a 200 OK response, indicating the AI job is complete.
- **Security:**
    - **Stripe Webhooks:** Securing the webhook endpoint by verifying the Stripe signature.
    - **Image Uploads:** Securing uploads by having the backend provide short-lived, authenticated tokens for each client-side upload via /api/upload-auth.
- **Gemini API Handling:** The Gemini API can return data in different formats (a dedicated inlineData part or Base64 embedded in a text part). The api/gemini/edit/route.ts has to be robust enough to parse both preferred and fallback response structures.

10. Outcomes

The project successfully delivers a polished, functional, and monetizable SaaS platform.

- It demonstrates a mastery of the modern web stack (Next.js, Prisma, PostgreSQL, Tailwind).
- It shows a deep understanding of integrating and orchestrating multiple third-party APIs (NextAuth, Stripe, ImageKit, Gemini) into a single, cohesive application.
- The architecture is efficient, offloading heavy computation (image processing) to specialized services (ImageKit, Gemini) and keeping the application lightweight and

scalable on a platform like Render.

11. Future Enhancements

- **Batch Editing:** Allow users to upload multiple images and apply the same AI actions (e.g., "remove background") to all of them.
- **Password Reset:** Implement a "Forgot Password" flow for the Credentials-based authentication.
- **More AI Tools:** Integrate other AI models for features like object removal ("magic erase"), style transfer, or text-to-image generation.
- **Team Accounts:** Introduce a "Pro Team" plan where a single subscription can be shared by multiple users.
- **Prisma Data Proxy:** Officially implement the Prisma Data Proxy to manage the PostgreSQL connection pool efficiently on Render.

12. Common Interview Questions and Answers

Q: Can you walk me through the tech stack and why you chose these technologies?

A: "Certainly. For the core framework, I chose **Next.js** with the App Router because it provides a powerful full-stack experience, handling both the frontend UI and the backend API routes.

For the frontend, I used **React.js** with **Tailwind CSS** for rapid styling and **Framer Motion** for a polished, animated user experience.

On the backend, I used **Prisma** as the ORM for my **PostgreSQL** database. I chose Prisma for its type-safety and developer-friendly API, which simplifies database operations. I chose PostgreSQL for its robustness and scalability as a relational database.

For services, I integrated:

- **NextAuth.js** for authentication. I implemented both **Google OAuth** for quick sign-in and a traditional **CredentialsProvider** for email/password, offering users flexibility.
- **Stripe** for payments, because its robust API and webhook system are industry-standard for building reliable SaaS subscriptions.
- **ImageKit** for image handling. This was a key architectural decision. It handles secure uploads and, most importantly, provides a URL-based API for *most* AI transformations, offloading heavy processing to their CDN.
- **Google Gemini** for the advanced generative edits that ImageKit doesn't cover, allowing for powerful, prompt-based image manipulation."

Q: You mentioned you deployed to Render. Why not Vercel, which is common for Next.js?

A: "While Vercel is fantastic for Next.js, I chose **Render** for this project because it excels at hosting *all* the components of a full-stack application in one place. With Render, I can easily manage my **Next.js Web Service** and my **PostgreSQL Database** from a single dashboard.

Render provides an internal connection string for the database, which simplifies setup and security. This unified infrastructure is perfect for a SaaS application that relies heavily on its database, and Render's pricing model for these services was a great fit for the project."

Q: You have two types of authentication. How does the session management work?

A: "NextAuth.js handles the session abstraction beautifully. Regardless of whether a user logs in via Google or with their email and password, NextAuth.js creates a standardized session. I use the database session strategy, so the session is persisted in my PostgreSQL database. In the authOptions, I use the session and jwt callbacks to augment this session. After a user is authenticated, these callbacks fetch their custom data from my users table—like their plan, usageCount, and usageLimit—and add it to the session token. This way, my frontend and API routes can instantly know the user's subscription status and permissions, no matter how they logged in."

Q: How does a user's subscription status (Free vs. Pro) affect their experience?

A: "The subscription status is central to the application flow.

1. When a user logs in, their session token is populated with their plan and usageCount from the database.
2. On the frontend, before an image upload is initiated, the app calls a custom API endpoint at /api/usage.
3. This endpoint checks the user's session data. If usageCount is greater than or equal to usageLimit, it returns a 403 Forbidden error.
4. The client catches this 403 error and, instead of uploading, triggers a modal that prompts the user to upgrade to the 'Pro' plan.
5. If the user is 'Pro', their usageLimit is set to a virtually unlimited number, so this check always passes."

Q: Walk me through the process of a user applying a "Gemini Edit".

A: "This is a key feature that involves a round-trip to my backend and the Google AI API.

1. First, the user uploads an image via **ImageKit**, and the original URL is stored in the app's state.
2. The user types a prompt like "give him sunglasses" and clicks "Apply".
3. The client-side applyGeminiEffect function is triggered. It first fetches the *original* image from the ImageKit URL and converts it to a Base64 string.
4. It then sends a POST request to my /api/gemini/edit endpoint (running on Render), with a JSON body containing the Base64 image data and the text prompt.
5. My backend API route receives this, initializes the Google Gemini AI SDK, and sends both the image and the prompt to the gemini-2.5-flash-image-preview model.
6. The Gemini API processes this and returns a *new* image. My backend parses the model's response, extracts the Base64 data for the *new* image, and sends it back to the client.
7. The client then updates its state to display this new Base64 image as the result, and the

'Job Status' panel updates to 'Completed'."

Q: How do you handle database connections in a serverless environment like Render?

A: "That's a critical challenge. Serverless functions, like the ones Render uses for Web Services, can scale up and open many concurrent database connections, potentially exhausting the PostgreSQL limit.
The prisma/schema.prisma file shows the connection to the database, but for a production environment on Render, the best practice (and a key future enhancement for this project) is to use Prisma's Data Proxy.
This would involve running npx prisma generate --data-proxy and updating the DATABASE_URL to point to the Prisma proxy. The proxy manages a persistent connection pool, so my serverless functions don't open and close new connections on every request. This is essential for both performance and stability."