The Detailed Technical Plan: Minimal Shopify UI + Railway Backend

This document provides a detailed, code-rich plan for building your Shopify app using the minimal frontend approach. It covers the exact project structure, frontend components, and the API communication layer that connects to your existing Railway backend.

The Architecture at a Glance

As discussed, we are building a very thin frontend layer that lives inside Shopify and a powerful backend that does all the work. This is the most efficient path to market.

- **Frontend (React + Polaris):** A single page with a form. Its only jobs are to collect the user's prompt, send it to your backend, and show the progress/result.
- **Backend (Your existing Railway App):** Contains all the AI logic. It will be adapted to accept authenticated requests from the Shopify frontend and to manage the long-running generation task as a "job".

Phase 1: Frontend Project Structure

After running `shopify app init` as described previously, you will have a project structure. We will simplify it. You can delete most of the template files. Your React frontend (`web/frontend/`) will only need a few key files:

Plain Text

```
/web/frontend
├── App.jsx                 # Main app wrapper with Polaris & App Bridge provid
├── components/
│   └── ProductGenerator.jsx  # The core UI component with the form and logic
└── pages/
    └── index.jsx           # The main page that renders ProductGenerator
```

This is it. You don't need complex routing or state management libraries for this minimal setup.

Phase 2: The Frontend Code (React + Polaris)

This is the code for the three key files in your frontend.

`App.jsx` - The Main Wrapper

This file sets up the necessary providers from Shopify. It's mostly boilerplate.

JSX

```jsx
import { AppProvider as PolarisProvider } from "@shopify/polaris";
import { Provider as AppBridgeProvider } from "@shopify/app-bridge-react";
import { AppProvider, Frame } from "@shopify/polaris";
import enTranslations from "@shopify/polaris/locales/en.json";
import Pages from "./pages";

export default function App() {
  const host = new URL(window.location.href).searchParams.get("host");

  const appBridgeConfig = {
    apiKey: process.env.SHOPIFY_API_KEY, // Injected by Shopify CLI
    host: host,
    forceRedirect: true,
  };

  return (
    <PolarisProvider i18n={enTranslations}>
      <AppBridgeProvider config={appBridgeConfig}>
        <Frame>
          <Pages />
        </Frame>
      </AppBridgeProvider>
    </PolarisProvider>
  );
}
```

`pages/index.jsx` - The Main Page

This simply renders your main UI component inside a Shopify `Page` component.

JSX

```jsx
import { Page } from "@shopify/polaris";
import { ProductGenerator } from "../components/ProductGenerator";

export default function HomePage() {
  return (
    <Page title="AI Product Generator">
      <ProductGenerator />
    </Page>
  );
}
```

`components/ProductGenerator.jsx` - The Core UI and Logic

This is the most important frontend file. It contains the form, the button, the API call logic, and the status polling.

```jsx
import { useState, useCallback, useEffect } from "react";
import {
  Card,
  Form,
  FormLayout,
  TextField,
  Button,
  ProgressBar,
  Banner,
  Link
} from "@shopify/polaris";
import { useAppBridge } from "@shopify/app-bridge-react";

export function ProductGenerator() {
  const app = useAppBridge();
  const [prompt, setPrompt] = useState("");
  const [jobId, setJobId] = useState(null);
  const [jobStatus, setJobStatus] = useState(null);
  const [finalProduct, setFinalProduct] = useState(null);
  const [error, setError] = useState(null);

  const handlePromptChange = useCallback((value) => setPrompt(value), []);

  // Polling effect to check job status
  useEffect(() => {
    if (!jobId || jobStatus === "completed" || jobStatus === "failed") return;

    const interval = setInterval(async () => {
      try {
        const response = await app.fetch(`/api/job-status/${jobId}`);
        const data = await response.json();

        setJobStatus(data.status);

        if (data.status === "completed") {
          setFinalProduct(data.result);
          setJobId(null); // Stop polling
        } else if (data.status === "failed") {
          setError("The generation process failed. Please try again.");
          setJobId(null); // Stop polling
        }
      } catch (e) {
        setError("Failed to get job status.");
        setJobId(null); // Stop polling
      }
    }, 3000); // Poll every 3 seconds

    return () => clearInterval(interval);
  }, [jobId, jobStatus, app]);

  const handleSubmit = async () => {
    setError(null);
    setFinalProduct(null);
    setJobStatus("running");

    try {
      // This is the call to your Railway backend
      const response = await app.fetch("/api/generate-product", {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({ prompt }),
      });

      if (!response.ok) {
        throw new Error("Failed to start generation job.");
      }

      const { jobId } = await response.json();
      setJobId(jobId);

    } catch (e) {
      setError(e.message);
      setJobStatus(null);
    }
  };

  return (
    <Card>
      <Card.Section>
        <Form onSubmit={handleSubmit}>
          <FormLayout>
            <TextField
              value={prompt}
              onChange={handlePromptChange}
              label="Product Idea"
              placeholder="e.g., a vintage t-shirt with a surfing astronaut"
              multiline={4}
              autoComplete="off"
            />
            <Button submit loading={jobStatus === "running"}>
              Generate Product
            </Button>
          </FormLayout>
        </Form>
      </Card.Section>

      {jobStatus === "running" && (
        <Card.Section>
          <ProgressBar progress={75} size="small" />
          <p>Generating your product... This may take up to a minute.</p>
        </Card.Section>
      )}

      {error && (
        <Card.Section>
          <Banner title="Error" tone="critical">{error}</Banner>
        </Card.Section>
      )}

      {finalProduct && (
        <Card.Section>
          <Banner title="Success!" tone="success" onDismiss={() => setFinalProd
            <p>
              Your draft product has been created.{" "}
              <Link url={finalProduct.adminUrl} target="_blank">View and publis
            </p>
          </Banner>
        </Card.Section>
      )}
    </Card>
  );
}
```

Phase 3: The Backend Code (Your Railway App)

Your existing Railway backend needs to be adapted to handle this new asynchronous workflow. Instead of processing everything in a single, long-running request, you will now create a "job" and let the frontend poll for its status.

The Job Management Logic

We will create a simple in-memory object to store the status of our generation jobs. For a production app, you would replace this with a Redis instance or a PostgreSQL table.

Here is the full backend code, showing the two new endpoints and the background processing logic.

JavaScript

```javascript
// In your main backend file (e.g., index.js)
const express = require("express");
const { shopifyApi, LATEST_API_VERSION } = require("@shopify/shopify-api");
const { v4: uuidv4 } = require('uuid'); // To generate unique job IDs
require("dotenv").config();

// --- Shopify Auth Setup (from previous guide) ---
const shopify = shopifyApi({ ... });
const app = express();
app.use(express.json());

// --- In-Memory Job Store (replace with DB for production) ---
const jobs = {};

// --- The Long-Running AI Workflow ---
// This is your existing logic, now wrapped in an async function
async function runPodWorkflow(prompt) {
  // 1. Generate artwork (call LLM, image models, etc.)
  // const artwork = await generateArtwork(prompt);

  // 2. Create mockups
  // const mockupUrl = await createMockup(artwork);

  // 3. Write copy
  // const copy = await writeProductCopy(prompt);

  // This is a placeholder for the real result
  return {
    title: `AI Product: ${prompt.substring(0, 20)}`,
    descriptionHtml: "<p>Generated by AI.</p>",
    imageUrl: "https://via.placeholder.com/300", // Replace with your real mock
  };
}

// --- The API Endpoints ---

// Endpoint 1: Start the Generation Job
app.post("/api/generate-product", async (req, res ) => {
  // Authenticate the request
  const session = await shopify.session.getCurrent({ rawRequest: req, rawRespon
  if (!session) {
    return res.status(401).send("Unauthorized");
  }

  const { prompt } = req.body;
  const jobId = uuidv4();

  // Create the job in our store
  jobs[jobId] = {
    status: "running",
    prompt: prompt,
    result: null
  };

  // Immediately return the Job ID to the frontend
  res.status(202).json({ jobId });

  // --- Run the actual workflow in the background ---
  // We don't `await` this. This lets the function run without blocking the res
  (async () => {
    try {
      // 1. Run your existing AI workflow
      const workflowResult = await runPodWorkflow(prompt);

      // 2. Create the product in Shopify
      const client = new shopify.clients.Graphql({ session });
      const { data } = await client.query({
        data: {
          query: `mutation productCreate($input: ProductInput!) {
            productCreate(input: $input) {
              product { id handle }
            }
          }`,
          variables: {
            input: {
              title: workflowResult.title,
              descriptionHtml: workflowResult.descriptionHtml,
              status: "DRAFT",
              images: [{ src: workflowResult.imageUrl }],
            },
          },
        },
      });

      const productId = data.productCreate.product.id;
      const productHandle = data.productCreate.product.handle;
      const adminUrl = `https://admin.shopify.com/store/${session.shop.split('.
      // 3. Update the job as completed
      jobs[jobId].status = "completed";
      jobs[jobId].result = { adminUrl };

    } catch (error) {
      console.error("Job failed:", error);
      jobs[jobId].status = "failed";
    }
  })();
});

// Endpoint 2: Get Job Status
app.get("/api/job-status/:jobId", async (req, res) => {
  // Authenticate the request (important!)
  const session = await shopify.session.getCurrent({ rawRequest: req, rawRespon
  if (!session) {
    return res.status(401).send("Unauthorized");
  }

  const { jobId } = req.params;
  const job = jobs[jobId];

  if (!job) {
    return res.status(404).json({ error: "Job not found" });
  }

  res.json({
    status: job.status,
    result: job.result
  });
});

// Your Shopify auth routes (/api/auth/begin, /api/auth/callback) go here...

app.listen(3000, () => console.log("Backend listening on port 3000"));
```

How the Frontend and Backend Communicate

This architecture creates a clean, robust loop:

1. **User clicks "Generate"** in the React UI.
2. The frontend `app.fetch` sends a `POST` request to `/api/generate-product`.
3. The Railway backend immediately creates a job, returns a `202 Accepted` status with the `jobId`, and starts the AI workflow in the background.

4. The React frontend receives the `jobId` and starts its `setInterval` polling, sending `GET` requests to `/api/job-status/:jobId` every 3 seconds.
5. The backend is busy running the AI workflow. During this time, the status endpoint returns `{ status: 'running' }`.
6. Once the AI workflow is done and the draft product is created in Shopify, the backend updates the job status to `completed` and stores the final product admin URL.
7. On its next poll, the frontend receives `{ status: 'completed', result: { adminUrl: '...' } }`.
8. The `useEffect` hook in the React component sees the "completed" status, stops the polling, and updates the UI to show the success banner with the link.

This asynchronous, job-based approach is the standard and correct way to handle long-running tasks in web applications. It prevents server timeouts, provides a great user experience, and is much more scalable than trying to process everything in a single request.