Project Blueprint: AI-Powered Government Job Aggregator

This document outlines the strategic vision, technical architecture, and development roadmap for building a next-generation, AI-powered platform for tracking government job vacancies.

1. Project Vision

To create a highly accurate, real-time, and user-friendly platform that automatically aggregates, verifies, and presents government job vacancies. The system will leverage modern AI (via the Gemini API) to intelligently search the web, extract data from disparate sources (including PDFs), and provide users with verified, citable information. The frontend will be a "smart" an adaptive, responsive application that provides an optimal experience on all devices by detecting viewport aspect ratios and capabilities.

2. Core Architecture Overview

The system is best understood as three distinct but interconnected components:

1. **The Frontend (User Application):** A highly responsive web application (likely a Next.js SPA) where users search, filter, and view job listings.
2. **The Backend (API & Control Center):** A central API (e.g., Python/FastAPI) that serves data to the frontend, manages user authentication, and orchestrates the Data Pipeline.
3. **The Data Pipeline (The "Smart Engine"):** An autonomous, server-side system of scrapers and AI processors that find, extract, verify, and store job information.

3. Phase-by-Phase Development Plan

Phase 1: Foundation & MVP (Manual-Entry)

Goal: Build the core platform and prove the user value before automating.

1. **Tech Stack Setup:**
    - **Frontend:** Next.js (React) with Tailwind CSS.
    - **Backend:** Python (FastAPI) or Node.js (Express).
    - **Database:** PostgreSQL.
2. **Database Schema:** Design the database to hold all required data points:
    - `Job(job_id, title, organization, description, ...)`
    - `JobDates(job_id, application_start, application_end, exam_date, ...)`
    - `Eligibility(job_id, age_min, age_max, education_level, ...)`
    - `Vacancies(job_id, category, count, ...)`

- `Cutoffs(job_id, year, category, score, ...)`
- `Source(source_id, job_id, url, citation_text, ...)`

3. **Core App:**
   - Build the frontend to display jobs from the database.
   - Implement robust search and filtering (by date, eligibility, category).
   - Create a secure admin panel for *manually* entering and updating job data.
4. **Initial Deployment:** Get the MVP online using Vercel (for Next.js) and a cloud provider (like AWS RDS or GCP Cloud SQL) for the database.

Phase 2: The Automated Data Pipeline

Goal: Build the "smart engine" that replaces manual data entry.

1. **Scraper Development:**
   - Use Python (Scrapy, BeautifulSoup, Playwright) to build scrapers for 10-15 *key* official sources (e.g., UPSC, SSC, major state PSCs).
   - Scrapers will download HTML content and relevant PDFs.
2. **AI Extraction (Gemini API):**
   - For each new piece of content (HTML or extracted PDF text), call the Gemini API (`gemini-2.5-flash-preview-09-2025`).
   - **Crucial Technique:** Use a `generationConfig` with a `responseSchema` to force the AI to return structured JSON that matches your database schema.
   - **Prompt:** "You are an expert data extractor. Analyze the following text and extract all details for the job posting. Use the provided JSON schema. If information is missing, use 'null'."
3. **Verification Engine:**
   - Build a service that implements the "cross-verification" logic.
   - **Logic:** When a job is found on `Source B` that *looks like* a job already in the database from `Source A` , flag it for review.
   - **Rule:** Do not "add the job" (i.e., set `is_published = true`) until key data points (e.g., `application_end` , `vacancy_count`) are present and, if possible, verified by a second source or a human admin.
4. **Source & Citation:**
   - For every piece of data extracted, the AI must also provide a `citation` (the text snippet) and `source` (the URL) which you store. This is vital for user trust.

Phase 3: Advanced AI & User Experience

Goal: Integrate AI for the user and perfect the "smart" frontend.

1. **AI-Powered Search (RAG):**
    - Implement Retrieval-Augmented Generation (RAG) for user search.
    - **How:** As jobs are added, use the Gemini API to create "embeddings" (vector representations) of the job descriptions and store them in a Vector Database (e.g., Pinecone, ChromaDB).
    - **Result:** A user can search "engineering jobs in Rajasthan with no application fee" instead of just keyword matching.
2. **Advanced Device Detection:**
    - This goes beyond standard responsive design (`min-width` breakpoints).
    - Implement CSS Media Queries for `aspect-ratio` to fine-tune layouts for specific screen shapes.
    - **Example:**

```
/* Standard 16:9 PC Monitor */
@media (aspect-ratio: 16/9) {
  .job-card-grid { grid-template-columns: repeat(3, 1fr); }
}

/* Common 20:9 Mobile Phone */
@media (aspect-ratio: 20/9) {
  .job-card-grid { grid-template-columns: repeat(1, 1fr); }
  .header { padding-bottom: env(safe-area-inset-bottom); }
}

/* Ultra-wide Monitors (handles "curved screen" use case) */
@media (min-aspect-ratio: 21/9) {
  .container { max-width: 1400px; /* Prevent content from becoming toc
}
```

3. **User Accounts:** Add features for users to save jobs, set alerts, and track applications.

Phase 4: Security, Scale & Maintenance

Goal: Harden the platform and ensure long-term health.

1. **Security Hardening (The "Expert Stuff"):**
    - **Follow OWASP Top 10:**
        - **SQL Injection:** Use an ORM (like Prisma, SQLAlchemy) to prevent this.
        - **Broken Access Control:** Ensure users can only edit their *own* saved jobs.
        - **Cross-Site Scripting (XSS):** Sanitize all user-generated content (e.g., in search queries) and content from scrapers.

- **API Security:** Implement rate limiting (to prevent abuse) and DDOS protection (via Cloudflare or your cloud provider).
- **HTTPS:** Enforce SSL/TLS everywhere.

2. **CI/CD & Testing (Lowering Technical Debt):**
   - **CI/CD:** Set up GitHub Actions to automatically test and deploy new code.
   - **Testing:** Write unit tests (e.g., Pytest, Jest) for critical functions (especially data extraction and verification logic).
   - **Linting:** Use tools like Black (Python) and ESLint (JS) to maintain a consistent code style.
3. **Monitoring:**
   - Implement real-time monitoring (e.g., Sentry, Datadog) to catch errors in the Data Pipeline immediately.
   - Create a dashboard to monitor the health of your scrapers (e.g., success rate, new jobs found).

4. Deep Dive: Key Technologies & Concepts

A. Gemini API Integration (The "Smart Code")

You will use the Gemini API (specifically `gemini-2.5-flash-preview-09-2025`) in two ways:

1. **For Data Extraction (Backend):**
   - **API:** `generateContent`
   - **Model:** `gemini-2.5-flash-preview-09-2025`
   - **Payload:**

```
{
  "contents": [{ "parts": [{ "text": "Extracted text from PDF/HTML..."
  "generationConfig": {
    "responseMimeType": "application/json",
    "responseSchema": {
      "type": "OBJECT",
      "properties": {
        "job_title": { "type": "STRING" },
        "application_end_date": { "type": "STRING", "description": "Fc
        "vacancies": {
          "type": "ARRAY",
          "items": { "type": "OBJECT", "properties": { "category": { "
        }
        /* ... all other fields ... */
      }
    }
  }
```

```
}
```

2. **For Web Search & Verification (Backend):**
    - To find new official sources or to cross-verify a job, you can use the API with Google Search grounding.
    - **Payload:**

```
{
  "contents": [{ "parts": [{ "text": "Find the official notification f
  "tools": [{ "google_search": {} }]
}
```

    - The response will include `groundingAttributions` with `uri` and `title` which you can use as primary sources to scrape.

B. "Curved Screen" & Advanced Device Detection

A "curved screen" is a hardware feature. From a developer's perspective, this is not something you detect. The *real* challenge is **ultra-wide aspect ratios**.

Your CSS must be robust enough to handle *both* width and aspect ratio.

- Use `min-width` breakpoints for general layout (mobile, tablet, desktop).
- Use `aspect-ratio` queries (as shown above) to fine-tune layouts for "tall" (e.g., 20:9 phones) or "wide" (e.g., 21:9 monitors) screens.
- Use `max-width` on your main content container to prevent text from becoming unreadably wide on ultra-wide monitors.
- Use CSS `clamp()` and `calc()` functions for fluid typography and spacing.

C. Real-Time Updates

"Real-time" can mean two things:

1. **Data Freshness:** Your backend Data Pipeline should run on a schedule (e.g., every 4 hours via a Cron job) to find new jobs.
2. **Pushing to Client:** When new data is added, you don't want users to have to refresh.
    - **Simple:** The Next.js app re-fetches data on every navigation.
    - **Advanced:** Use WebSockets or Server-Sent Events (SSE) to push a small "new jobs available" notification to active users, prompting them to refresh or automatically loading the new data.

5. Recommended Technology Stack

- **Frontend: Next.js** (or SvelteKit)
- **Styling: Tailwind CSS**
- **Backend: Python (FastAPI)** (Recommended due to its strength in AI, scraping, and data processing)
- **Database (Jobs): PostgreSQL**
- **Database (Vectors): ChromaDB** (Open-source) or **Pinecone** (Managed)
- **AI Model: Google Gemini API** (`gemini-2.5-flash-preview-09-2025`)
- **Scraping: Python (Scrapy, Playwright)**
- **Deployment: Vercel** (Frontend), **AWS/GCP/Azure** (Backend, Database, Pipeline)