

Full-Stack Automated IT Job Search System – Project Plan (Updated)

Table of Contents

- 1. Project Overview & Goals
 - 2. System Architecture (Frontend-Backend Separation)
 - 3. Technology Stack
 - 4. Frontend (Next.js) Components
 - 5. Backend (FastAPI) API Design
 - 6. Data Processing Pipeline
 - 7. Automation & Scheduling
 - 8. Security & Authentication (Optional)
 - 9. Compliance & Ethical Considerations
 - 10. Implementation Roadmap
 - 11. Deployment & Hosting
 - 12. Scalability & Maintenance
 - 13. Expected Outcome & Success Metrics
-

1. Project Overview & Goals

This project aims to deliver a **completely free, self-hosted** system that streamlines the IT job search process by automatically **crawling, filtering, and aggregating** new job postings every day. The platform will surface relevant IT opportunities matching the user's profile and keywords, presenting them through a user-friendly **web dashboard**. By automating the repetitive parts of job hunting, the system saves time while maintaining zero operational costs through careful technology choices and architecture design.

- **Primary Purpose:** Create a personalized job discovery engine that scans popular job boards and company career pages each morning, finds IT positions matching predefined criteria, and presents a curated list of opportunities via a web interface (with optional one-click application support in the future).
- **User Benefits:** The system provides significant time savings by automating daily searches across multiple sites. It offers comprehensive coverage through multi-source aggregation and precise targeting via customizable keyword filters. Users get early access to new listings with daily updates, and can manage these opportunities (track applied jobs, add notes) from one dashboard. All data is stored locally for privacy, and the solution is completely cost-free with no subscriptions or paid APIs required.

2. System Architecture (Frontend-Backend Separation)

The solution follows a **full-stack architecture** with a clear separation between the frontend UI and backend logic. The frontend is built with **Next.js** (a React framework) and serves as the interactive dashboard, while the backend is a **FastAPI** (Python) service handling data crawling, processing, and storage. The two layers communicate via RESTful API calls over HTTP:

[User Browser] ⇌ ****Next.js**** Frontend (React UI) ⇌ ****FastAPI**** Backend (Python API) ⇌ Database

- **Frontend (Next.js):** Renders the job search dashboard and pages. It requests data and actions from the backend via HTTP APIs (e.g. fetch the latest jobs or trigger a new crawl). Next.js supports Server-Side Rendering (SSR) for fast loads and could use its built-in API routes for auth or proxying if needed.
- **Backend (FastAPI):** Exposes a set of REST endpoints that the frontend consumes. The backend contains the core logic: schedulers, crawlers, parsers, and the database interface. It processes incoming requests from the UI (for data or actions) and returns JSON responses. The backend also performs the daily automated crawling in the background.
- **Database:** A persistent store holds all job postings and related data. The FastAPI backend reads from and writes to this database. By default, a local **SQLite** database can be used for simplicity, while supporting **PostgreSQL** or **MongoDB** as options for larger scale or cloud deployment.
- **Communication:** The frontend and backend communicate over HTTP using a defined contract of API endpoints. For example, when the Next.js app needs the latest jobs, it calls the FastAPI `GET /jobs` endpoint; when a user clicks "Rescan", the app sends a `POST /rescan` request to trigger crawling. This decoupled design allows the frontend and backend to be deployed independently and scale separately ¹. The API design is documented in Section 5.

This layered architecture ensures that the web UI remains responsive and secure while heavy lifting (crawling and data processing) happens on the server side. It also makes the system more maintainable: changes in scraping logic or data handling on the backend do not require modifications to the frontend UI, and vice versa.

3. Technology Stack

The system uses a modern **full-stack JavaScript/Python** tech stack, leveraging free and open-source tools to meet all requirements at zero cost:

- **Frontend: Next.js** (React 18+) for the user interface, providing a rich interactive dashboard with Server-Side Rendering and routing. It runs on Node.js (v18+) and can be easily deployed on services like Vercel or run locally. The UI will be styled with a lightweight component library (or custom CSS/Tailwind) for a clean look, and use fetch/Axios for API calls.
- **Backend API: FastAPI** (Python 3.11+) for implementing the RESTful API and backend logic. FastAPI is a high-performance framework that provides automatic data validation and documentation. It supports asynchronous operations, which is ideal for I/O-bound tasks like web crawling. The backend will integrate with web scraping libraries and NLP libraries (see below) and run under Uvicorn/Gunicorn for production.
- **Web Crawling:** Use **Crawl4AI** or **Scrapy** for robust crawling, and **Playwright** (headless browser) or **Selenium** for dynamic pages that require JavaScript rendering. For simpler sites, **Requests + BeautifulSoup** will handle HTML fetching and parsing. These tools enable multi-source scraping with respect for rate limits and robots.txt. ² ³
- **Data Parsing & NLP:** Use **lxml/BeautifulSoup4** for parsing HTML into structured data. For relevancy scoring, utilize **spaCy** or small **HuggingFace Transformers** models (e.g. DistilBERT) to compute text similarity. This NLP layer filters and ranks jobs by matching them to the user's keywords and profile (all running locally for free).
- **Database: SQLite** (file-based SQL database) is the default storage for job data due to its zero configuration and light weight ⁴. It will store job records (with fields like title, company, description, date, etc.) and user annotations (e.g., applied status, notes). SQLite is sufficient for a

single-user setup and keeps everything self-contained. For more intensive use or multi-user scenarios, the code can be configured to use **PostgreSQL** (robust relational DB) or **MongoDB** (NoSQL), since the data access layer will be abstracted (e.g., using an ORM like SQLAlchemy for portability).

- **Scheduling & Automation:** The backend can use Python scheduling (e.g. **APScheduler** or cron jobs) to automate the daily crawl. Optionally, **GitHub Actions** (CI) or a simple CRON on a server can trigger the crawl script each morning if the system is not running 24/7 ⁵. This flexibility ensures the job search runs every day without manual intervention.
- **Deployment & DevOps: Docker** is used to containerize the application for easy deployment. A Docker setup will include separate images/containers for the Next.js frontend (Node image) and the FastAPI backend (Python image), plus a DB container if using PostgreSQL. **Docker Compose** ties these together, so the entire stack can be started with one command ⁶ ⁷. This guarantees a consistent environment across development and production. For a cloud deployment, the frontend and backend containers can be hosted on any platform (Vercel, AWS, DigitalOcean, etc.), and an nginx reverse proxy or Traefik can route requests to the appropriate service.
- **Version Control & CI/CD:** The project will use **GitHub** for version control. Automated tests (using Pytest for backend and Jest/React Testing Library for frontend) can be set up. A CI pipeline (GitHub Actions) can run tests and build the Docker images on each commit. Deployment can also be automated with CD workflows or managed through services like Vercel (for frontend) and Fly.io/Heroku (for backend).

Overall, this stack ensures a **modern, maintainable, and free** solution. Next.js + FastAPI is a powerful combo for fast frontends and scalable backends, confirmed by industry best practices ⁸ ⁹. All components are chosen for cost-efficiency and their strong open-source communities.

4. Frontend (Next.js) Components

The Next.js frontend provides an intuitive web **dashboard** for the user to interact with the system. Key pages and components include:

- **Dashboard Page (Job Listings):** The main page (e.g. `/jobs` or homepage) that displays a list of all relevant job postings in a table or card layout. Each listing shows key details (title, company, location, post date, relevance score) and maybe an excerpt of the description. Users can search or filter this list (by keyword, location, etc.) using a filter bar at the top. There is also a "Refresh" or **Rescan** button on this page to manually trigger an immediate crawl for new jobs. The dashboard periodically indicates new postings (or auto-refreshes) so the user always sees up-to-date opportunities.
- **Job Detail Page:** A page (e.g. `/jobs/[id]`) that shows the full details of a specific job listing. When a user clicks on a job on the dashboard, the app navigates to this page, which displays the complete job description, requirements, and the original apply link. It also provides actions such as **"Mark as Applied"** (which updates the job's status in the database) or **"Save Note"** to add a personal note or rating to the job entry. These actions call backend APIs (e.g., a PUT/PATCH to update the job record). If the user decides to apply, an "Apply Now" button can open the original job posting URL in a new tab.
- **Settings Page:** A page (e.g. `/settings`) where the user can configure preferences. This includes managing the list of keywords/skills to filter for, toggling sources (which job boards to crawl), and notification settings (email/Telegram details if notifications are enabled). The settings page calls backend endpoints to fetch or update configuration (for example, getting the current keyword list, or posting a new list of keywords). This page ensures the system can be tailored to the user's needs without modifying code.

- **Login Page (Optional):** If authentication is enabled (see Section 8), the app will have a simple login page (`/login`) that prompts for a username/password or an access token. Upon successful auth, a session cookie or token is stored so the user can access the other pages. This page will use Next.js API routes or directly call FastAPI (e.g., `POST /login`) to verify credentials and establish a session.

Reusable UI Components: The Next.js app will be structured into reusable React components for maintainability. For example, a `JobList` component will render the list of job cards, and a `JobCard` component will represent a single job item. A `FilterBar` component will encapsulate search input and filters. These components can be composed on the Dashboard page. The use of a component library or design system (like Chakra UI or Material UI) can accelerate development of polished, responsive UI elements.

State Management & Data Fetching: The dashboard will fetch data from the backend API either via Next.js `getServerSideProps` (SSR) or through client-side fetch calls (using React hooks or libraries like SWR for auto-refresh). SSR can improve initial load and SEO by fetching jobs server-side on each request ¹⁰, whereas client fetches can provide live updates without full page reloads. In this app, SEO is not critical (it's a personal tool), so we might use client-side fetching for simplicity – the page will call the API to load jobs on mount and perhaps poll every few minutes for new ones. Next.js also supports API routes, but here we rely on the external FastAPI for the actual backend logic. The UI will handle loading states and errors gracefully (e.g., showing a message if the backend is unreachable).

Overall, the frontend focuses on clarity and ease of use: a single-page style experience where the user can scan new jobs, view details, and manage preferences all in one place. By leveraging Next.js, we ensure fast performance and the flexibility to deploy the UI anywhere (static or dynamic) independent of the backend.

5. Backend (FastAPI) API Design

The FastAPI backend exposes a set of **RESTful API endpoints** that allow the frontend to retrieve data and perform actions. All endpoints are prefixed with `/api` (for example, `/api/jobs`) for clarity. The API returns JSON responses and uses standard HTTP methods (GET, POST, etc.). Below is an overview of key endpoints and their purpose:

Endpoint	Method	Description
<code>/api/jobs</code>	GET	Retrieve the list of job postings. Supports query parameters for filtering (e.g., <code>?q=<keyword></code> or <code>?location=<city></code>) and pagination. Returns an array of job objects (each including id, title, company, location, date, score, etc.).
<code>/api/jobs/<id></code>	GET	Retrieve full details for a specific job by its ID. Returns a detailed job object including the full description, requirements, and any stored metadata (e.g., whether marked as applied, any notes).
<code>/api/jobs/<id></code>	PUT/ PATCH	Update an existing job entry. This is used to mark a job as applied, or to add/edit notes. The request body contains the fields to update (e.g., <code>"applied": true</code> or <code>"notes": "Great opportunity"</code>). Returns the updated job record. (Note: This would be an authenticated endpoint if auth is on, to prevent unauthorized modifications.)

Endpoint	Method	Description
<code>/api/rescan</code>	POST	Trigger an immediate crawl for new jobs. When the user clicks "Rescan" on the frontend, it calls this endpoint. The backend then initiates the crawling process (for all configured sources) in the background. The response could be a quick acknowledgment (<code>{"status": "started"}</code>) or a summary of results after completion. This endpoint might perform the crawl synchronously (blocking until done) or asynchronously (return immediately and run tasks in background threads), depending on implementation.

(The base URL is assumed to be the backend's address. For example, if running locally, the Next.js frontend might call `http://localhost:8000/api/jobs` to reach the FastAPI.)

The API is designed with simplicity in mind for a single-user scenario. Additional endpoints could be added as needed, such as `/api/settings` (GET/PUT to retrieve or update user preferences like keywords), or `/api/sources` to manage the list of job boards to crawl. These are optional and can be implemented in the future or in the settings phase.

API Implementation Details: FastAPI makes it straightforward to define these endpoints with Pydantic models for request/response schemas and automatic documentation (Swagger UI). Each endpoint will hook into the underlying logic of the system: - `GET /jobs` will query the database for the latest jobs (possibly applying filters in the DB query). It may support basic filtering logic (e.g., matching title or company to a query string). - `GET /jobs/<id>` will look up the job by primary key (or a hash id) in the database and return the record. - `PUT /jobs/<id>` will parse the JSON body and update the corresponding fields in the database (ensuring to only allow certain fields to be updated, like status or notes, not core fields). - `POST /rescan` will internally call the crawler module to perform a new scrape. To avoid long blocking requests, this might spawn a background task (using FastAPI's `BackgroundTasks` or an async task queue like Celery/RQ if needed). The endpoint could immediately return a 202 Accepted status while the crawl runs, and the frontend can poll `/jobs` for new results or show a loading indicator. Alternatively, for simplicity, the endpoint can perform the crawl synchronously and return the new jobs found.

All endpoints enforce proper error handling and return appropriate HTTP statuses (e.g., 400 for bad requests, 500 for server errors with error logs, etc.). If authentication is enabled, protected endpoints (like the PUT or rescan) will require a valid token or session (see Section 8).

6. Data Processing Pipeline

The backend's internal workflow for discovering and processing jobs follows a linear pipeline architecture. **Crawling, parsing, filtering, and storing** of job data happen as sequential steps, which can be triggered on a schedule or via the `/rescan` API call. The pipeline stages are as follows:

1. **Content Acquisition (Crawling):** The **Crawler** component initiates requests to each target source (e.g., specific job board URLs or company career pages). For each source, it fetches the raw HTML (or uses an API if available). This may involve making HTTP GET requests with appropriate headers (to mimic a real browser) and obeying each site's `robots.txt`. If a page is JavaScript-heavy, the crawler uses headless browser automation (Playwright) to load and render the page before extraction. The crawler is designed to be polite: it includes delays between requests and uses rotating user-agent strings to avoid overwhelming any site. It also

implements robust error handling (retry with backoff on network failures, skip or log any pages that consistently fail) to ensure one bad site doesn't halt the whole crawl.

2. **Information Extraction (Parsing):** Once raw content is fetched, the **Parser** component processes the HTML/JSON to extract structured data fields for each job posting. This involves using CSS selectors or XPath queries to find elements like job title, company name, location, posting date, job description, apply link, etc. The parser normalizes these fields (e.g., consistently formatting location or job titles, expanding abbreviations). It may also do some classification – for example, identifying if the job is full-time vs contract, or extracting salary range if mentioned. This stage yields a structured **Job object** for each posting. If the site offers an API (like a JSON feed), the system can directly map those fields. The parser also identifies any keywords from the user's list that appear in the job description or title (for use in the next stage).
3. **Filtering & Relevance Scoring:** The **Filter** component examines each structured job object to decide if it's relevant for the user. This is done through a combination of keyword matching and simple NLP: for instance, the system checks if the required keywords/skills (provided in settings) appear in the title or description. It can also use an NLP model to compute a similarity score between the job description and the user's profile or ideal job description. Each job gets a **relevance score** (e.g., a numeric score or a pass/fail) based on these criteria. Non-IT or irrelevant jobs are dropped at this stage. For example, if the user is looking for "Python Developer" roles, a job that mentions a lot of Python/Django will score high, whereas a generic "Project Manager" job with no tech keywords would be filtered out. This stage ensures the output list is tailored to the user's interests.
4. **Storage of Results:** Filtered relevant jobs are then saved to the **database**. The storage component assigns each job a unique ID (could be a hash of the URL or a combination of title+company) to detect duplicates. It performs deduplication: if a job already exists in the DB (from a previous run), it will skip or update it (e.g., if certain fields changed). New jobs are inserted with a timestamp. The database schema might include fields like `id, title, company, location, description, post_date, url, keywords_matched, relevance_score, applied_status, notes, crawl_date`. Using SQLite or another DB allows querying and updating this data easily.
5. **Delivery to User:** Once jobs are stored, they become available to the user through the UI and notifications. In the updated system, the primary delivery mechanism is the **Next.js dashboard**, which will fetch the latest jobs from the database via the API and display them. This means whenever the user opens the dashboard (or after a new crawl), the frontend calls `GET /jobs` and shows the new entries. Additionally, an **optional notification system** can send a summary of new jobs found via email or Telegram each day (for example, listing the top 5 new postings). In this case, the pipeline would compile the relevant new jobs into an email and send it via an SMTP service or to a Telegram bot. Notifications are an add-on for convenience; the web dashboard itself serves as the primary interface for reviewing new jobs each day.

Throughout this pipeline, extensive **logging** is maintained. Each run (whether automated daily or manual rescan) logs its actions and outcomes: which sources were crawled, how many jobs found, how many filtered out, any errors encountered, etc. Logs can be written to a file or to a log table in the database. This helps with debugging and monitoring the health of the system over time.

(Note: The above steps correspond to what happens when `/api/rescan` is called or the daily schedule triggers. If the user is just viewing the dashboard at any given time, the pipeline is not actively running; the dashboard is simply reading whatever data was last stored.)

7. Automation & Scheduling

To continuously discover new jobs without user intervention, the system includes an **automation mechanism** that triggers the crawling pipeline every day (or at a configured schedule). There are a few approaches to scheduling in a self-hosted environment, and the system is designed to accommodate flexibility:

- **Backend Scheduler (APScheduler):** The FastAPI backend can launch with an integrated scheduler (using a library like APScheduler or schedule) that programmatically triggers the crawl at a set time each day. For example, upon startup, the backend could schedule the `crawl_jobs()` task to run at 7:00 AM local time daily. This approach keeps everything within the application process. FastAPI's event handlers or background tasks can be used to kickoff such scheduled jobs. The crawler task would run in a background thread or async task so as not to block the API serving.
- **System Cron Job:** Alternatively, one can rely on the host machine's scheduler (cron on Linux/Mac or Task Scheduler on Windows) to call the crawl regularly. In this scenario, the crawling logic might be exposed as a management script or simply trigger the FastAPI `/rescan` endpoint via an HTTP request. For example, a cron job could curl `http://localhost:8000/api/rescan` every morning. This decouples scheduling from the app, which some may prefer for simplicity.
- **GitHub Actions (CI) or External Scheduler:** As an option for users who cannot host 24/7, the project can include a GitHub Actions workflow (as described in the original plan) that runs daily on GitHub's servers to execute the crawler and perhaps commit the results to a repository or Google Sheet. However, with the introduction of a persistent backend and database, GitHub Actions is less central. It could still be used for running nightly crawls and, for example, pushing the SQLite DB file or sending an email artifact. This is an optional deployment mode and requires the project to be public or use repository secrets for credentials ⁵.

In the default setup, we will use the **integrated scheduler within FastAPI** for a seamless self-hosted experience. We'll configure the job to run at a reasonable hour (e.g., early morning). The scheduling component will be careful to avoid overlapping runs – if a previous crawl is still running, the next scheduled run should skip or wait, to prevent stacking tasks.

The scheduling system will also allow on-demand runs (via the aforementioned `/rescan` API). This means if the user wants to refresh the listings outside the normal schedule, they can do so with a button click, and it won't interfere with the regular daily job.

All scheduling times are configurable (perhaps via a config file or environment variable). Timezone considerations will be documented (for instance, if using cron on a server, ensure it's set to the user's local TZ, etc.). Logging for scheduled tasks is important – e.g., if a scheduled crawl fails or throws an exception, it should be logged and possibly surfaced to the user (maybe via the UI or email alert) so they know the job search might have been interrupted.

8. Security & Authentication (Optional)

Since this system is primarily for personal use (running locally or in a private server), it can initially be deployed without complex authentication. However, if exposed on the internet or used by multiple

users, **authentication and basic security** measures should be added to protect access. Here are some considerations and options:

- **Basic Login Protection:** The simplest approach is to put the entire frontend behind a login. For example, implement a Next.js **middleware** that checks for a cookie or session, and if not present, redirects to a `/login` page. The backend can have a `/api/login` endpoint where the user submits a preset username/password (defined in config). On success, the frontend stores a session token (e.g., JWT or HttpOnly cookie). This prevents random internet users from accessing your job dashboard.
- **API Key or Token:** For API calls, FastAPI can require an API key header or token. For instance, the user could set an environment variable `API_TOKEN`, and the FastAPI endpoints would check for this token in requests (either as a header or query param). Only calls with the correct token succeed. This is easy to implement with FastAPI dependency injections. The Next.js frontend would be configured to include this token in its requests (possibly stored securely). This effectively restricts API usage to your UI.
- **JWT Authentication:** FastAPI has built-in support for OAuth2 and JWT. We could implement a more standard auth where the user logs in and receives a JWT that must accompany future API requests (in an Authorization header). The Next.js app can store this token (probably in memory or secure storage) and attach it to fetch calls. This approach is more scalable and aligns with modern practices ¹¹, but for a single-user MVP, it might be overkill. Still, it's good to note that FastAPI makes it easy to integrate JWT auth, and libraries like NextAuth.js can handle the frontend side if needed.
- **HTTPS and CORS:** If deploying the frontend and backend separately, ensure that the backend has proper **CORS** settings to allow the frontend domain to fetch data. Also, when deploying externally, use **HTTPS** for both frontend and backend to encrypt traffic, especially if credentials or tokens are involved. If using Docker Compose locally, this might not matter, but any cloud deployment should obtain TLS certificates (e.g., via Let's Encrypt, possibly automated by a tool like Traefik).
- **Rate Limiting and Throttling:** As an added security measure, the FastAPI could implement simple rate limiting (to prevent abuse of the `/rescan` endpoint, for example). This could be as simple as not allowing more than N rescan calls per minute. Given the personal scope, this is rarely an issue, but it's part of robust API design.

For the initial version, we can assume the app runs locally or in a private environment, so we might skip authentication entirely for convenience. The user's machine or network security acts as the gatekeeper. If the app is later hosted on a cloud VM or behind a public URL, we strongly encourage enabling one of the above auth methods (even a single hardcoded password) to prevent unauthorized access.

9. Compliance & Ethical Considerations

Web scraping must be approached carefully to maintain ethics and legal compliance. The system will incorporate the following best practices:

- **Respect Robots.txt and Terms of Service:** Before scraping a site, the crawler will check the site's `robots.txt` to ensure it is allowed to scrape the particular pages. Any site that disallows scraping their job listings will be skipped to comply with their rules. Additionally, the system avoids actions that would violate a website's Terms of Service.
- **Polite Crawling:** The crawler is designed to be **non-intrusive**. It uses a reasonable delay between requests to the same domain (to avoid hammering any single site). We will set a custom User-Agent string identifying the tool and an contact email, so site administrators know this is a

benign job-search bot. No scraping of any site will be done at a rate faster than what a human user would do.

- **Avoiding Protected Content:** The system will only scrape publicly accessible pages. It will not attempt to bypass paywalls, CAPTCHAs, or login-required pages. If a job board requires a login, that source will be considered out-of-scope (unless it provides an open API). This avoids legal and technical issues.
- **Data Privacy:** The data collected is only job listing information – which is public job postings, not personal user data. All data is stored locally under the user's control (in the SQLite/DB file or Google Sheet if that option is used). No personal data is being collected from users, and the user's own information (like keywords or profile) is kept private within the app. If any personal details are needed for auto-apply (like resumes), those will be stored securely and not transmitted anywhere except the target application site under user control.
- **Transparency and Open Source:** The entire codebase is open-source, meaning the user (and others) can inspect how data is collected and used. This transparency builds trust that the system is doing nothing nefarious. Users can modify the code to further comply with any specific ethical guidelines they prefer.
- **Legal Considerations:** We acknowledge that scraping even public data can have legal constraints (e.g., some jurisdictions have laws against scraping without permission). By respecting the site's `robots.txt` and not redistributing the scraped data beyond personal use, we minimize risk. The system acts as an **assistant to the user**, effectively doing what the user could do manually in a browser, which is generally considered acceptable for personal use.

By adhering to these practices, the project strives to be a “good citizen” in the web ecosystem while providing the desired automation benefits to the user.

10. Implementation Roadmap

The development will be structured in phases, roughly over **6 weeks**, to incrementally build and integrate the full-stack system. Each week's focus and tasks are outlined below:

- **Week 1: Backend Foundation & Crawling Basics** – Set up the Python project structure and FastAPI application. Implement 2-3 initial crawlers for selected job sources (e.g., one popular job board and one company careers page) using requests/BeautifulSoup or Crawl4AI. Test fetching and parsing these pages to ensure we can extract job fields correctly. Define the data model and create the SQLite database schema (tables for jobs, etc.). Also, stub out the FastAPI endpoints (`GET /jobs` and `POST /rescan` with dummy data) to lay the groundwork for API integration. By end of week, we have a basic backend that can scrape a couple of sources and store results locally, and serve a placeholder API.
- **Week 2: Backend Logic & API Completion** – Expand the crawling to more sources and refine parsing logic. Implement the **filtering and ranking** module: create a function to score jobs based on keywords (and possibly a simple spaCy similarity). Integrate this into the crawl pipeline so irrelevant jobs are dropped. Complete the FastAPI endpoints to actually read from the database (e.g., `GET /jobs` returns real data from SQLite). Implement the `POST /rescan` endpoint to run the crawl pipeline on-demand (this might use a background task thread). Unit test the pipeline components with sample pages. By end of week, the backend can autonomously collect and filter jobs and expose the data via API.
- **Week 3: Frontend Setup & Job Listings UI** – Initialize the Next.js project (create a new Next.js app). Set up the project structure with pages for Dashboard, Job Detail, and Settings (they can start as simple stubs). Implement the Dashboard page to display jobs: this involves calling the

backend API (perhaps using `getServerSideProps` or an SWR hook) to fetch the list of jobs and rendering them in a table or list. Create React components for job items and ensure the page is styled for readability. At this stage, we may use mocked API responses or the real backend if the backend is running. Ensure CORS is configured on FastAPI to allow the Next.js dev server to fetch data. By end of week, the Dashboard UI should be showing actual job data from the backend API.

- **Week 4: Frontend-Backend Integration & Features** – Enhance the frontend with interactivity and integrate all pages with the backend. Implement the Job Detail page: when a user clicks a job on the dashboard, fetch the job details from `GET /jobs/{id}` and display the full description, etc. Add the ability to mark a job as applied or add notes – this means creating a form or button on the detail page that calls the `PUT /jobs/{id}` endpoint to update the DB, and updating the UI accordingly (maybe show a checkmark or label on applied jobs). Implement the Settings page to allow updating filter keywords: for now, this could simply edit a JSON or text config via an API endpoint (or directly update a config file). Also, incorporate the "Rescan" button on the dashboard: hook it up to call `POST /rescan` and provide user feedback (e.g., disable the button and show "Scanning..." until new results appear). By the end of this week, the front-to-back integration should be fully functional – users can view, refresh, and update job entries through the UI. Basic styling and UX improvements can be done here so the app feels cohesive.
- **Week 5: Testing, Refinement & Optional Enhancements** – This week is for polishing the system and ensuring reliability. Write integration tests for the API (using Pytest and FastAPI's test client) and for the frontend (basic React tests or manual testing). Refine the crawling: add more job sources or improve parsing where errors were observed. Optimize performance (e.g., if certain pages are slow, consider caching or increasing asynchronous usage). If not already, integrate the NLP ranking more deeply – for example, use a small transformer model to re-rank the jobs and display a "relevance score" on the UI. This week can also introduce **authentication** if needed: for instance, add a simple login page and protect the backend routes (implement the token check). Ensure that the system behaves well under edge cases (no jobs found, API downtime, etc.). By end of week, the system should be robust and ready for deployment.
- **Week 6: Deployment & Dockerization** – Containerize the application and finalize deployment details. Create a production build of the Next.js frontend and set up the Node server to serve it. Write a Dockerfile for the frontend and backend, and a Docker Compose file that brings up: the FastAPI app, a PostgreSQL DB (if switching from SQLite for production), and the Next.js app (or Next.js can be exported to static if possible). Test the whole system in Docker (run `docker-compose up`) and ensure the frontend can talk to backend correctly via network). Document the setup steps in a README (how to configure environment variables, etc.). Additionally, deploy the system to a cloud environment for a trial run – e.g., spin up a small VM and run Docker Compose, or deploy the frontend to Vercel and backend to Render. Verify that scheduling still works in the deployed environment (may need to adjust cron timing or server timezone). By end of week 6, we have a deliverable: the full system running in a production-like setting, with documentation for others to replicate it.

Post-MVP Extensions: If time permits or for future phases, consider implementing the advanced features from the original plan:

- **Advanced NLP and ML (Phase 2):** integrate more sophisticated AI models (like GPT-4 via API or advanced transformers) to parse job descriptions and even auto-generate cover letters. This could also include a recommendation engine that learns from jobs the user applies to.
- **Automated Application (Phase 2):** extend the system to auto-fill applications on certain sites. This would involve storing user resume details and using a headless browser to navigate application forms.

Given the ethical considerations, this should remain opt-in and carefully monitored.

- **Multi-User Support:** Convert the system to support multiple users with separate accounts, each with their own saved jobs and preferences. This would involve a user database and authentication, and is a larger scope (would likely require role-based access in the API, etc.).

These are optional and can be planned in subsequent iterations. The initial 6-week roadmap focuses on delivering a solid, usable full-stack product.

11. Deployment & Hosting

By default, the entire system is **self-hosted** – meaning you can run it on your local machine or a private server – using Docker for convenience. Deployment can also be done to cloud services if needed. Key deployment scenarios:

- **Local Deployment (Docker Compose):** We provide a `docker-compose.yml` that defines all services: e.g., a service for the FastAPI backend, one for the Next.js frontend, and one for the database (if using a server-based DB like Postgres). Using Docker Compose allows us to start everything with a single command and ensures all components can communicate in a network. For example, the backend might be accessible at `http://backend:8000` within the Docker network, and the frontend at `http://frontend:3000`. We map these to host ports so you can open the browser to the Next.js app on `localhost:3000`. Environment variables are used to configure the database connection, API keys, etc., without hardcoding them. Docker will also make it easy to restart the app on reboots and isolate dependencies. **Example:** a Compose file will build the images and run containers for each component, as shown in guides ¹² ¹³. With one command (`docker compose up -d`), your entire stack comes online.
- **Cloud VM Deployment:** You can deploy the Dockerized app to any VM or cloud instance (AWS EC2, DigitalOcean Droplet, etc.). This is as simple as copying the docker-compose setup to the server and running it. Ensure to update DNS or IPs so you can access the frontend via a browser. For a production environment, it's recommended to put a reverse proxy like **Nginx** or **Traefik** in front of the services. The proxy can handle serving the Next.js static files (if exported) and routing API calls to FastAPI, as well as terminating SSL (HTTPS). We might include a sample Traefik config for reference, which can automatically get Let's Encrypt certificates for your domain.
- **Vercel + Separate Backend:** Another deployment strategy is to host the Next.js frontend on **Vercel** (a platform optimized for Next.js), and deploy the FastAPI backend to a service like **Render**, **Heroku**, or **Google Cloud Run**. Vercel will build and serve the frontend, and we'd configure it with the backend's URL for API calls. The FastAPI could be containerized and run on one of those platforms, possibly with a managed database (like Heroku Postgres) attached. This approach might incur some costs if usage is beyond free tiers, but it offloads infrastructure management. Vercel provides automatic scaling for the frontend, and something like Cloud Run can scale the backend. According to a modern reference architecture, the frontend could run on a platform like Vercel while the backend runs as a Docker container on a service like Fly.io or AWS, behind a gateway ⁹.
- **Configuration Management:** In all deployment scenarios, sensitive information (like email credentials for notifications, or API tokens) should be managed via environment variables or secrets. Docker Compose allows defining these in an `.env` file which is not committed to source control. We will document all the required configurations (e.g., `DB_URL`, `TELEGRAM_TOKEN`, etc.) in the README for deployment.
- **Resource Considerations:** The app is lightweight – it can run on a small VM (even a \$5/month droplet or a Raspberry Pi). If using headless browsing (Playwright), ensure the environment can

support it (Docker image may need to include the browser drivers). We'll provide a base image that has these dependencies. Logging and monitoring can be done via container logs; for a more robust setup, one could integrate something like Prometheus/Grafana to monitor the system, but that's optional.

- **Running without Docker:** Developers can also run the system directly: e.g., start the FastAPI server with Uvicorn on port 8000, run `npm run build && npm start` for Next.js on port 3000, and ensure a database is running. This is fine for development or debugging, though Docker will be the recommended path for deployment consistency.

In summary, deployment is flexible: you can keep it entirely self-contained on your hardware (no recurring costs, just your electricity and hardware) or deploy to the cloud for accessibility. The provided Docker configuration emphasizes ease-of-setup for self-hosting, aligning with the goal of a free and private solution. ⁶

12. Scalability & Maintenance

Even though this is a personal project, it's built with scalability and easy maintenance in mind. Key practices and features ensure the system can grow and adapt without significant cost or effort:

- **Modular Scaling:** Thanks to the frontend-backend separation, each part can be scaled independently. If the job database grows or crawl tasks increase, you can scale up the backend (e.g., run multiple FastAPI workers or move to a more powerful machine) without changing the frontend. Similarly, the Next.js app can be optimized (using static generation for certain pages, caching results) to handle more load if needed. In a multi-user adaptation, you could run multiple backend instances behind a load balancer, and Next.js could be served via a CDN for global reach. These changes can be made without altering core logic, due to the loose coupling in the design ⁹.
- **Zero-Cost Scaling Strategies:** The system implements intelligent **caching** to avoid unnecessary work. For example, crawlers can cache the last-seen posting for each source and only fetch new posts beyond that, or skip crawling a site if it was just crawled an hour ago (unless it's the scheduled time). This reduces bandwidth and CPU use. The architecture supports adding new job sources easily by configuration – you can drop in a new spider class or URL in a config file, and the pipeline will include it in the next run, making the system extensible. Data processing is incremental; only new or updated listings are processed in depth, which means even as the database grows, daily processing time stays manageable.
- **Monitoring & Logging:** The system includes comprehensive **logging** at each step of the pipeline, which aids in monitoring. A log of each crawl run (how many jobs found, any errors) is maintained. For maintenance, the system could provide a simple dashboard view (or even just log files) to review these logs. Source-specific success rates are tracked – if one source consistently fails (maybe the site changed its HTML or blocked the crawler), the system can flag it for review. We could even implement an auto-disabling feature: if 5 attempts to crawl a site fail, skip that source for a few days and notify the user to check the credentials or method.
- **Error Handling & Resilience:** The code is written to be resilient to individual failures. A failure in one part of the pipeline will not crash the entire system. For example, if one job listing's parsing fails due to unexpected format, that error is caught and logged, and the pipeline continues with the next listing. If one source website is down or returns error, the crawler skips it and continues with others, ensuring the overall job run completes. We also implement **retry with backoff** for transient network issues. In case the entire crawl fails (e.g., no internet connectivity), the system will try again at next schedule and can send an alert if failures persist. The database is periodically pruned of old entries (configurable retention, e.g., keep last 6 months of jobs) to prevent indefinite growth and to remove jobs that are no longer active.

- **Maintenance and Updates:** Because the project is open-source and modular, maintenance mostly involves updating scraping rules when websites change. To facilitate this, we separate the configuration for each source (like CSS selectors or API endpoints for each job board) into easily editable modules or config files. If a site redesigns its HTML, updating the selector in one place fixes it. We also document how to add a new source or update an existing one. Regular library updates (for security and improvements in FastAPI, etc.) should be done – using tools like Dependabot or pip-tools to manage dependencies can help keep things up to date. Writing unit tests for critical parts (parser, filter) will catch if a change introduces a bug, making maintenance safer.

Overall, the system's design favors **low maintenance overhead** – once it's set up, it should run daily with minimal issues. Scalability is inherently limited by the fact it's a personal tool (one user, moderate data volume), but the architecture doesn't preclude scaling to more users or more data. And importantly, all this is achieved without incurring costs: by using local resources efficiently, leveraging free tiers (for notifications or cloud runners), and not over-using any external service, the system remains cost-free to operate.

13. Expected Outcome & Success Metrics

Upon completion, this project will deliver a **full-stack job search automation platform** that significantly enhances a user's ability to find relevant IT jobs effortlessly. The success of the system can be measured in both functional deliverables and quantitative metrics:

- **Core Deliverables:** Users will get a **web-based dashboard** that lists IT job opportunities tailored to their preferences, updated automatically each day. The deliverables include a well-documented codebase (Python backend and Next.js frontend) with instructions to run or deploy it, and a pre-configured set of job sources to crawl. The system will have an integrated schedule for daily crawls and an interface for manual refresh. It will also include the necessary scripts/config for Docker deployment. Essentially, anyone should be able to pull the repository, set a few configs (like their keywords, email for notifications), and start the system to have their personal job-hunting assistant running. All data (job listings, logs, config) will be stored locally by default, ensuring privacy and control.
- **Efficiency Gains:** The user should experience at least a **90% reduction** in the time spent on manual job searching. Instead of checking multiple sites every day, the system aggregates those and presents results automatically. If the user previously spent an hour a day on job boards, this could drop to just 5-6 minutes of scanning the dashboard or reading an email summary (time saved can be redirected to preparing applications or learning new skills).
- **Coverage and Relevance:** The system aims to find **50+ new job postings per day** that match the user's profile (exact numbers will vary based on market activity and chosen sources). It will cover a broad range of sources (at least 5-10 different job boards or company sites), ensuring the user isn't missing opportunities. The relevancy filtering should achieve about **80% precision** – meaning 8 out of 10 jobs presented should be on-target (relevant to the user's keywords), minimizing noise. This can be measured by user feedback or by evaluating how many of the listed jobs the user is actually interested in. Over time, the user can tweak keywords to improve this precision.
- **System Robustness:** The job crawler should run daily with a high success rate (e.g., >95% of scheduled runs execute without errors). Any transient failures should self-recover by the next run. The system's logging and error notifications will contribute to this reliability by alerting the

user if something goes wrong (e.g., if all sources failed due to no internet, the user can notice and fix the issue).

- **User Engagement:** As a qualitative metric, the user's interaction with the dashboard (e.g., marking jobs applied, adding notes) indicates that the system becomes an integrated part of their job application workflow, not just a read-only feed. The easier it is to manage applications through the dashboard, the more value it provides. Success would mean the user can track which jobs they applied to directly in the system, creating a one-stop location for their job search status.

In conclusion, this project will result in a **powerful yet cost-free job search assistant** that runs on a full-stack architecture. It combines the strength of automated backend processing (crawling and AI-driven filtering) with the convenience of a modern frontend UI. Success is achieved when the user no longer needs to manually scour job sites – instead, they rely on the system to surface the right opportunities, thereby making the job hunt more efficient and even enjoyable. With its modular design and detailed roadmap, the system is set up not only to meet immediate needs but to evolve with future requirements in the ever-changing landscape of job search and recruitment.

1 8 9 10 11 Creating a Scalable Full-Stack Web App with Next.js and FastAPI | by Vijay Potta (pottavijay) | Medium

<https://medium.com/@pottavijay/creating-a-scalable-full-stack-web-app-with-next-js-and-fastapi-eb4db44f4f4e>

2 3 4 5 Project Plan 3.txt

file:///file_00000000a3346209b97f2e2412e9f80e

6 7 12 13 How to Develop a Full Stack Next.js, FastAPI, PostgreSQL App Using Docker - Travis Luong

<https://www.travisluong.com/how-to-develop-a-full-stack-next-js-fastapi-postgresql-app-using-docker/>