**Newspaper.AI Development Roadmap**

This roadmap outlines a **detailed, step-by-step implementation plan** for developing **Newspaper.AI**, an AI-driven personalized news aggregator. The plan is organized into phased milestones, each focusing on a key aspect of the project. Following these phases in order will ensure the application is **fully deployable, scalable, and ready for real-world usage**. Each phase below includes the objectives, necessary tools/tech, tasks to complete, and the expected deliverables.

Supabase Project ID: mrfcrewlkwrqtwzlxpra

https://mrfcrewlkwrqtwzlxpra.supabase.co

Supabase Anon Public: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJzdXBhYmFzZSIsInJlZiI6Im1yZmNyZXdsa3dycXR3emx4cHJhIiwicm9sZSI6ImFub24iLCJpYXQiOjE3NDQxNDQ3MzIsImV4cCI6MjA1OTcyMDczMn0.6jzgK-K6nntipO0ZSnmXSAvb53xqp7-uQF\_S7KHDLJU

Service\_role eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJzdXBhYmFzZSIsInJlZiI6Im1yZmNyZXdsa3dycXR3emx4cHJhIiwicm9sZSI6InNlcnZpY2Vfcm9sZSIsImlhdCI6MTc0NDE0NDczMiwiZXhwIjoyMDU5NzIwNzMyfQ.pWvGZB2yzW9r2k9pBIX5SWeoV\_T-wwbDWLagoO6KyoYsecret

**News APIs USE ALL TO MAXIMUM API REQUESTS PER DAY:**

1. **TheNewsAPI**
   * URL: https://www.thenewsapi.com
   * API Key: c3AcERwz2ZsZc0ABKhwc00OnAOqhyCKAySAsdiSc
2. **NewsData.io**
   * URL: https://newsdata.io
   * API Key: pub\_8217957942c3b5247b479818ae6984adf5333
3. **ApiTube**
   * URL: https://apitube.io
   * API Key: api\_live\_kbAHRuznY0gfNohekvP6dDFgtYGzy0Afsnwg1hYA1jRvAOrDEblvhq
4. **Breaking News Trends Detection (GitHub Repo)**
   * URL: @https://github.com/oubaidHL/Breaking-News-Trends-Detection
   * No direct API key provided (this is a code repository, not an API service).
5. **NewsAPI**
   * URL: https://newsapi.org
   * API Key: 87ecc9641d894c0396b5495014879e9d

Please implement: Translation software for multilingual support/.

Supabase Project ID: mrfcrewlkwrqtwzlxpra

Github Repo

The repository used for your Web Service.

echo "# The-PWN" >> README.md

git init

git add README.md

git commit -m "first commit"

git branch -M main

git remote add origin https://github.com/ShaneAtkins1/The-PWN.git

git push -u origin main

git remote add origin https://github.com/ShaneAtkins1/The-PWN.git

git branch -M main

git push -u origin main

​​<https://github.com/ShaneAtkins1/The-PWN.git>

git@github.com:ShaneAtkins1/The-PWN.git

Render Deploy Hook:<https://api.render.com/deploy/srv-d00vp9idbo4c73dn9l10?key=yE-UIxSsPMM>

Build Command

Render runs this command to build your app before each deploy.

$ npm install && npm run build

Radar API Keys

| Live secret (server) | prj\_live\_sk\_a962f66029924cbd8ebe9914b1d1db2d20ce248b |
| --- | --- |
| Live publishable (client) | prj\_live\_pk\_1edbf17fac8ded8c1b82c18325bb62ec1aa956b5 |
| Test secret (server) | prj\_test\_sk\_c0f3b5843d7185039eb6bf51113602ee726a7e45 |
| Test publishable (client) | prj\_test\_pk\_96f89ba996fcca9bc84b26b46203bb8e1161e992 |

1. **Google Custom Search API**
   * Search Engine ID: 14062bd40491e400a
   * Google API Key: AIzaSyAuiMJJKV1d9m\_f7x08OhnTaCX4DAWEYTc

Openrouter API Key:

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

Google Oath Client ID: [996566628358-adhfr05nun4ngnqkbc84bglpg2msv1j4.apps.googleusercontent.com](http://996566628358-adhfr05nun4ngnqkbc84bglpg2msv1j4.apps.googleusercontent.com)

Google Cloud Console API Key: AIzaSyAuiMJJKV1d9m\_f7x08O

**Overview of Phases and Milestones**

**Phase** **Objective** **Key Tools/Tech** **Dependencies** **Deliverable**

**1. Setup** Initialize project environment and repository Node.js, React, Supabase, Git, CLI tools – (start) Base project structure & dev environment ready

**2. Arch Init** Set up frontend & backend architecture scaffolding React (Vite/CRA), Tailwind, Node/Express, Supabase client Phase 1 Running skeleton for frontend & backend (Hello World)

**3. Auth** Implement user authentication and onboarding Supabase Auth, React forms, Tailwind Phase 2 User sign-up/login flow and profile onboarding UI

**4. User Profile** Integrate Radar API & store user profile info (e.g. location) Radar SDK/API, Supabase DB Phase 3 User profile enriched with location/preferences in DB

**5. Aggregator** Develop news aggregation logic with external APIs TheNewsAPI, NewsData.io, APITube, etc., Cron jobs Phase 4 (profile data available) Automated fetch of latest news articles into database

**6. Summarization** Summarize news content using AI (LLM with RAG) OpenRouter (LLM API), possibly LangChain, pgVector (optional) Phase 5 (articles in DB) AI-generated summaries for news articles stored in DB

**7. Personalization** Generate “Why this affects you” context for user User profile data, OpenAI (for text) Phases 4,5,6 Personalized relevance note for each article per user

**8. Data Storage** Categorize content & ensure de-duplication in DB Supabase (Postgres), SQL, indexing Phase 5 (data ingest) Structured DB schema with categories & no duplicates

**9. Frontend UI** Build responsive UI for browsing news feed React, Tailwind CSS, Framer Motion Phases 3,5,6 (data & auth ready) Polished web app UI (mobile-friendly feed with animations)

**10. Testing** Ensure quality with unit and e2e tests Vitest, React Testing Library, Playwright Phases 1–9 (features done) Comprehensive test suite (unit & end-to-end) passing

**11. Deployment** Configure deployment and scheduling Render.com (CLI, Cron), Docker (optional), Supabase CLI Phases 1–10 Deployed application (frontend + backend jobs) in production

**12. Monitoring** Set up logging and user analytics Sentry (error logging), PostHog/Plausible Phase 11 (app running) Live monitoring of errors and user behavior analytics

**13. Multilingual** Add translation support for content OpenAI API or DeepL API, i18n library Phases 6–9 (content ready) News summaries available in multiple languages

**14. Enhancements** Implement optional features (save, notifications, dashboards) Service Workers (for push), Email API, Chart library Phases 9–12 Improved user engagement features (bookmarks, alerts, insights)

⸻

**Phase 1: Project Setup and Environment Configuration**

*Goal:* Set up the foundational development environment, repository, and project configuration for Newspaper.AI. This phase ensures all team members can start development with the necessary tools and that project standards are in place.

• **Initialize Version Control & Repo:** Create a new Git repository for the project. Set up the repository structure (e.g. a monorepo with separate folders for frontend and backend, or two repos if preferred). Include a README outlining the project and a .gitignore for Node/React, to avoid committing secrets or build artifacts.

• **Development Environment:** Install and configure the runtime and package managers:

• Choose Node.js LTS version and install it on all dev machines. Decide on a package manager (npm, Yarn, or pnpm) and initialize a package.json.

• Set up any needed global CLI tools (e.g. Render CLI, Supabase CLI) by installing them via npm or other sources.

• Ensure everyone has access to the same Node version (consider using an .nvmrc or .node-version file).

• **Frontend Bootstrapping:** Use a starter tool to create the React app:

• For example, run npm create vite@latest or create-react-app to scaffold a React project (TypeScript template recommended for type-safety).

• Verify the dev server runs (e.g. npm run dev for Vite, npm start for CRA) and “Hello World” appears in the browser.

• Add initial dependencies: React Router (for navigation), Tailwind CSS (for styling), Framer Motion (for animations). Configure Tailwind by creating a tailwind.config.js and include Tailwind directives in the CSS.

• **Backend Bootstrapping:** Set up a basic Node.js backend (if needed for API or cron tasks):

• If choosing Express.js (or a lighter framework) for any custom API endpoints, initialize it with npm init and install Express. Create a simple index.js or server.js that listens on a port and returns “OK” at a base route for health-check.

• If the backend will mainly consist of scheduled jobs and not a persistent server, you can still set up a basic project structure (perhaps as a standalone script or a Node project in a backend folder).

• Add necessary libraries for later integration (e.g. axios or node-fetch for API calls, openrouter or OpenAI SDK for AI calls, @supabase/supabase-js for database access).

• **Supabase Project Initialization:** Create a new project in Supabase (hosted Postgres database with Auth):

• Note the Supabase URL, public anon key, and service role key. These will be used in the app (public key for frontend, service key for secure backend operations).

• In the codebase, set up environment variable files (.env for local development) to store secrets such as Supabase keys, API keys (to be added later for news APIs, OpenRouter, etc.), and any config (like Render webhook if any). For example, define SUPABASE\_URL, SUPABASE\_ANON\_KEY, etc., and ensure these are loaded in the app (for Vite, use a .env.local and prefix Vite envs with VITE\_).

• **Code Quality and Tools:** Configure code linters/formatters for consistency:

• Install **ESLint** (with a React/TypeScript config) and **Prettier**. Optionally use pre-commit hooks (with Husky) to run linting/formatting on commits.

• Set up **GitHub Actions** or another CI pipeline (if using a platform like GitHub) to run tests (even if empty for now) and lint on each push, to catch issues early. This prepares for Phase 10 (testing) so that the pipeline is ready.

• **Verification:** Run the frontend and backend in development mode:

• Access the React app at http://localhost:3000 (or the port configured) to see the starter page.

• If a backend server is set up, hit its health endpoint (e.g. http://localhost:5000) to ensure it’s running.

• Commit the initial setup to the repo. This baseline will be built upon in subsequent phases.

• *Deliverable:* **Project Skeleton & Env Configured** – A Git repository is in place with a base React app (displaying a simple page) and a basic backend setup. All team members can run the project locally. Development tools (linters, formatters, environment files) are configured, and the Supabase project is ready to be used by the application.

**Phase 2: Frontend and Backend Architecture Initialization**

*Goal:* Establish a solid architecture for both the frontend and backend, setting the stage for feature development. This involves structuring the code, configuring frameworks, and preparing a cohesive stack (React for the client, Node/Supabase for the server side).

• **Frontend Architecture Setup:**

• Organize the React project structure into logical folders (e.g. components/, pages/, services/, etc.). If using React Router, define a basic route structure (e.g., routes for Login, Onboarding, NewsFeed, Saved articles, etc., even if pages are mostly stubbed out initially).

• Configure Tailwind CSS properly:

• Ensure the Tailwind directives (like @tailwind base; @tailwind components; @tailwind utilities;) are included in an index CSS and that Tailwind is processing the files (check that styling works by adding a Tailwind class to the sample page).

• Set up a basic theme or color scheme for the app’s branding using 353535 for the main color and 3c6e71, ffffff, d9d9d9, 284b63, for the secondary accent colors (optional at this stage, can use default Tailwind styles for now).

• Add Framer Motion and test a simple animation (for example, fade in the main heading) to verify the library is working and to lay groundwork for later adding transitions.

• If using a state management solution (though not strictly needed yet), decide on one (could be simple React Context for user/auth state, or a library like Zustand or Redux if complexity grows). Set up a basic context, e.g. an AuthContext that will hold current user info once logged in.

• **Backend Architecture Setup:**

• If a dedicated backend server is planned (for handling protected API calls or webhooks), flesh out the structure:

• Create an api folder or similar for route handlers. Perhaps plan for routes like /api/fetch-news (if triggering fetch via HTTP) or /api/summary (if summarization as a service), etc.

• Set up middleware in Express (if used) for JSON parsing, CORS (to allow the React frontend to call it if on different domain/port), and any logging (maybe morgan for request logs during dev).

• Test one sample endpoint that returns dummy data (e.g. GET /api/test returns { status: "ok" }).

• If the backend is mostly going to be a cron job process on Render, ensure that part of the code can run independently:

• Structure the code so that the news fetching logic (to be written in Phase 5) can be invoked either via a schedule or manually. This might mean writing the aggregator in a modular way (a function or class that can be called from a script).

• You might still use the Express server to expose an endpoint to trigger a fetch (for manual triggers or debugging) or to serve as a health check.

• **Supabase Integration:** Set up Supabase client usage in both frontend and backend:

• In the React app, install the Supabase JS SDK (@supabase/supabase-js). Initialize a client with the anon public key and URL (from Phase 1 env) at the entry point of the app. This will be used for authentication and data fetching in later phases.

• On the backend side, decide if you will use the Supabase client with the service key for admin operations (like inserting news articles server-side). If yes, install the SDK on backend as well and initialize with service role key (which has full DB permissions). This will allow the backend scripts to bypass row-level security when needed (e.g., inserting data).

• Test the DB connection: try a simple query to Supabase from the backend, for example creating or reading a test table. If not ready to query actual data, at least ensure no connection errors (Supabase uses REST under the hood, so check that the URL and keys are correct).

• **Shared Types and Configuration:** If using TypeScript (recommended), ensure both frontend and backend are configured for TS:

• Set up tsconfig.json for each (or a single one if monorepo). Include type definitions (e.g. for Node, Express, etc., and for React).

• Define common types/interfaces in a shared location (if monorepo, you could have a types/ folder). For instance, an interface for a NewsArticle (with fields like title, url, summary, etc.) that both front and back will use. This ensures consistency and ease of data transfer.

• **Basic CI/CD Integration:** Although deployment happens later, start configuring pipelines:

• Use GitHub Actions (or other CI) to automatically build the frontend and run the backend tests (if any) on each push. This ensures from now on, broken builds are caught early.

• Configure the CI to also run lint checks and perhaps type checks (e.g. tsc --noEmit).

• This lays the groundwork so that in Phase 10 (testing) we just integrate tests into this pipeline.

• **Documentation:** Update the README (or docs/ folder) with architecture decisions:

• Mention the tech stack (React + Tailwind + Framer on frontend, Node/Express + Supabase on backend).

• Document how to run both the frontend and backend locally (e.g. two commands).

• Document how environment variables are managed (which files to copy or how to set them).

• *Deliverable:* **Initial Architecture in Place** – The application has a foundational structure: the React frontend is set up with styling and basic navigation, and the backend framework (or at least the scaffold for scheduled tasks) is ready. Supabase connectivity is confirmed. At this milestone, the project can serve a simple page and has the skeleton to build upon in subsequent phases.

**Phase 3: Supabase Authentication and Onboarding Flow**

*Goal:* Enable user accounts in Newspaper.AI so that each user can have a personalized experience. This includes implementing sign-up/login via Supabase Auth and an onboarding process to gather user preferences (which will later drive personalized news delivery).

• **Enable Supabase Auth:** In the Supabase dashboard, configure authentication settings:

• Ensure email/password auth is enabled (by default, Supabase allows it). Optionally, enable third-party OAuth providers (Google, Facebook, etc.) if desired for easier sign-in – this can be done by providing OAuth client IDs in settings.

• Set up email confirmation or password recovery templates as needed (Supabase can handle sending confirmation emails by default).

• Define a **user profile table** in the database to store extra info (Supabase often creates a public.profiles table by default in their starter schema). If not present, create a table profiles with columns: id (UUID, matches auth.users ID), full\_name, avatar\_url (if needed), plus fields for preferences like interests and location (to be filled in onboarding). Ensure the id is PK and linked to auth user ID.

• Implement Row Level Security (RLS) on the profiles table so that users can only select/update their own profile. Supabase provides a policy template for user ID = auth.uid; enable that policy.

• **Frontend Auth Integration:** Using the Supabase JS SDK on the frontend:

• Create a sign-up page and a login page (or combine into one component with toggle). Use Supabase’s signUp and signIn methods. For email/password, call supabase.auth.signUp({ email, password }) and handle the result (if email confirmation is on, inform the user to check email).

• Similarly implement signIn for returning users. On success, Supabase provides a session and user object.

• Manage the auth state in the React app:

• Use a Context or simple state at the App level to store the current user session. Supabase SDK can auto-track the session, e.g. supabase.auth.onAuthStateChange to react to login/logout events.

• Protect routes: Set up React Router so that routes like /feed (the main news feed) are only accessible if logged in. If not, redirect to /login.

• Conversely, if user is logged in, redirect from /login page to the app (to prevent redundant login screen).

• **Onboarding UX:** After a new user signs up, guide them through an onboarding sequence to collect profile information:

• **Step 1: Interests Selection** – Present a list of news categories or topics and let the user choose what they care about. This could be multi-select checkboxes or stylish toggles for categories like *Technology, Sports, Politics, Health, Business, Science*, etc. (You can start with a fixed set of broad categories).

• Use Tailwind to style this selection page (e.g. category pills that highlight when selected).

• Save the selected interests in a temporary state; they will be written to the DB at the end of onboarding.

• **Step 2: Location permission** – Ask the user if they want to enable location-based news. Explain that allowing location can help deliver local news or relevant context (“news near you”).

• Provide a button to “Use my location” which will trigger the Radar integration (handled in Phase 4, but design the UI here).

• Also allow an option to skip or manually input a location (city or zip code) if they prefer not to share precise coordinates.

• If user agrees, you will later use Radar API to get location details; if they skip, it’s fine (profile can have location as null or a generic location).

• **Finalize Onboarding:** Have a summary or finish step thanking the user. When they finish:

• Write the collected profile data to Supabase: call supabase.from('profiles').insert({ id: user.id, interests: [...], location: ... }) using the currently authenticated user’s ID. (Because of RLS, ensure the insert or upsert uses the user’s JWT – Supabase JS handles this automatically if the user is logged in.)

• Alternatively, Supabase provides a user metadata or profile auto-fill mechanism, but explicit insert is straightforward here.

• After saving, navigate the user to the main news feed page.

• **Back-End Support for Onboarding:** (If needed)

• If any preprocessing is needed (for example, if the user picks very specific topics not just broad categories, you might want to normalize or map them to actual API query terms), plan for that. For now, storing the raw interests (category names or keywords) is fine – later phases will use these.

• Ensure that the profiles table is properly related: perhaps set the id = auth.uid and use that as foreign key. This way, whenever you query for user profile, you match on id = current user’s UID.

• **Testing Auth Flow:**

• Manually test the entire flow in development: create a new account, go through onboarding, then check in Supabase dashboard that a new auth user exists and their profile row is created with the chosen interests and possibly location placeholder.

• Test login with an existing account, and test logout (Supabase auth.signOut() method) to ensure it clears session and the app returns to login page.

• Test edge cases: try signing up with an existing email to see error handling, etc., and display any errors on the form (e.g. “Email already registered”).

• *Deliverable:* **User Auth & Onboarding Complete** – Users can register and log in securely. Upon first sign-in, they are taken through an onboarding process to set their news preferences (like interests and optionally location). The app now maintains a user profile in the database containing these preferences, which will be critical for personalized news retrieval in later phases. The authentication flow is fully functional and the app can distinguish between authenticated users and guests.

**Phase 4: Integration of Radar API and User Profiling Storage**

*Goal:* Enhance the user profile with contextual data using the Radar API (for location-based information) and ensure profile preferences are properly stored. This phase focuses on incorporating geolocation to tailor the news experience (e.g., local news or context about why a story matters to the user’s area).

• **Radar API Setup:**

• Sign up for a Radar (radar.com) account to obtain an API key. Radar is a geolocation platform that can handle location tracking and geocoding.

• Decide on usage: The simplest integration for our use-case is **geocoding** (turning coordinates into a readable location, like city and state). Radar’s API offers an endpoint to reverse geocode coordinates to a place name.

• In the frontend, install Radar’s JavaScript SDK if available (Radar provides a web SDK that can simplify obtaining location). If not using the SDK, you can directly call Radar’s REST endpoint from the backend.

• **Requesting User Location:**

• In the onboarding flow (Phase 3, Step 2), when the user clicks “Use my location”, trigger the browser’s Geolocation API to get coordinates:

navigator.geolocation.getCurrentPosition(successCallback, errorCallback);

In the success callback, you receive latitude and longitude.

• With the coordinates, call Radar’s reverse-geocode API. For example, make a fetch:

GET https:*//api.radar.io/v1/geocode/reverse?coordinates=<lat>,<lon>*

Headers: Authorization: <RADAR\_SECRET\_KEY>

Radar will return a JSON with address details. Extract the city, state, and country (or whichever level is appropriate).

• Alternatively, Radar’s SDK might have Radar.reverseGeocode() which returns similar info.

• Handle errors (if user denies location or the API call fails). If denied, perhaps allow manual entry as already planned. If API fails, you could default to no location.

• **Storing Location in Profile:**

• Update the Supabase profiles table structure if needed: add fields for city, region, country, or a single location text field. For simplicity, a single text field like location (e.g. “Seattle, WA, USA”) could be stored, or store structured fields if you plan to filter by them.

• When the Radar API returns data, save the relevant location info to the user’s profile record:

• If the profile was already inserted in Phase 3, you might do an update:

supabase.from('profiles').update({ city: ..., region: ..., country: ... }).eq('id', user.id).

• If you deferred profile insert until now, combine interests and location and insert all at once.

• Ensure this operation respects RLS (the user can update their own profile).

• **Augmenting User Profile with Additional Context (optional):**

• Beyond location, consider using Radar or other means to add more profile context that could personalize news:

• For example, if Radar can detect the user’s **country** or **timezone**, you can later use this to fetch country-specific news.

• Radar also supports geofencing; while not needed now, keep in mind for notifications (Phase 14) you could use Radar to trigger events when a user enters a geofence (e.g., near a point of interest).

• If relevant, store the country code (e.g. “US”, “IN”) in profile, which can be used when querying news APIs that support country filtering.

• **Testing Radar Integration:**

• In a development environment, test on a real browser (Geolocation API might not work on insecure http, so run the dev server with https or test the logic by hardcoding a lat/lng).

• Simulate or use the browser’s location (most dev tools allow setting a fake location). Ensure that the location returned by Radar is reasonable (e.g., “San Francisco, CA, USA”).

• Verify that after onboarding, the profile entry in Supabase has the location data populated alongside interests.

• Test the case of a user skipping location: ensure the app still creates a profile without it, and maybe fill a default country or set location fields to null.

• **Profile Retrieval:** Implement a quick check after login:

• When a returning user logs in, fetch their profile from Supabase (select from profiles where id = user.id) and store it in a React state (perhaps in the AuthContext). This will be used in Phase 7 for personalization logic and also might be useful to tailor the UI (e.g., display the user’s name or allow editing interests).

• Ensure this fetch happens early (maybe in an App.js effect after supabase.auth session is set, to load profile data).

• *Deliverable:* **Enriched User Profiles** – The application now captures and stores location data (via Radar) as part of the user profile. Each user’s profile in Supabase contains their chosen interests and (if given) their geographic context. This data is available for use in tailoring news content (e.g., prioritizing local news or adding “near you” context). The onboarding flow is fully functional with location integration, and the system gracefully handles cases where location is unavailable.

**Phase 5: News Aggregator Logic and API Handling**

*Goal:* Build the core news aggregation engine that will pull in news articles from various sources. This phase focuses on integrating external News APIs (TheNewsAPI, NewsData.io, APITube, NewsAPI.org, Google Custom Search, etc.) to retrieve news content, and preparing the data for further processing. It also includes detecting trending/breaking news and ensuring the fetched data is stored in our database.

• **API Selection and Setup:**

• Evaluate the news APIs available:

• **TheNewsAPI** and **NewsData.io** offer free tiers for news articles. These can provide headlines, article metadata, and in some cases full content. Sign up for API keys for these services.

• **NewsAPI.org** (the popular news API) has a free tier but limited (often 100 requests/day and no full content). It might be used as a backup or for specific queries.

• **APITube.io** claims to aggregate many sources in real-time; obtain an API key and review its docs – it might provide a unified feed of latest news across categories.

• **Google Custom Search (CSE)** can be used to search news if other APIs fail to find certain topics (you can create a custom search engine that targets news sites). This requires an API key and custom search engine ID, and Google’s API will return relevant links (but not structured news data like others).

• Create configuration for these APIs (in code or config files): base URLs, endpoints, API keys from env variables. For example:

• TheNewsAPI endpoint for latest news: https://api.thenewsapi.com/v1/news/top?language=en&api\_token=...

• NewsData.io endpoint: https://newsdata.io/api/1/news?apikey=...&country=us&category=... etc.

• Document the rate limits for each (e.g., NewsData.io maybe 1000 requests/day, etc.) to plan usage.

• **Design Aggregation Strategy:**

• **Coverage**: Ensure we cover a broad range of news topics and user-specific interests:

• Identify a set of top-level categories (e.g. the ones used in onboarding: Technology, Sports, Politics, etc.). For each category, plan to fetch the latest articles periodically.

• Also, consider **breaking news/trending topics**: This could be a separate fetch for “Top headlines” regardless of category. Many APIs have an endpoint for top headlines or trending news.

• For personalized topics not covered by broad categories (e.g., if a user enters a custom keyword interest during onboarding), we may need on-demand search. We can plan to fetch those in a targeted way (e.g., using NewsAPI or Google CSE to search that keyword once a day).

• **Scheduling**: Decide how often to fetch new articles:

• Breaking news could be checked more frequently (e.g., every 15 minutes for top headlines worldwide).

• Category updates could be every hour or few hours since category news doesn’t all update every minute.

• Custom user keywords might be fetched daily or every few hours depending on importance.

• We will implement scheduling with Render Cron in deployment, but for now, structure the code to allow calling an aggregateNews() function with parameters for what to fetch.

• **Fallback and Redundancy**: Plan for using multiple APIs to maximize results:

• One approach: Start with one primary source (say NewsData.io) to get most articles including full content. If it returns fewer results or lacks certain categories, supplement with TheNewsAPI or APITube.

• Alternatively, use multiple sources and combine results. For example, fetch 10 articles from NewsData and 10 from TheNewsAPI, merge them.

• Be mindful of duplicates (two APIs might provide the same story). Later in this phase we’ll handle deduplication logic.

• **Implementing the Fetcher (Backend Logic):**

• Create a module or service (e.g., newsFetcher.js) that can call an API and return normalized articles. Pseudocode:

async function fetchNewsFromSource(source, params) {

*// e.g., source = 'newsdata', params = { category: 'technology' }*

*// Build API URL based on source and params*

*// Perform fetch with axios or node-fetch*

*// Parse JSON response, map to common format: { title, description, url, content, image, source, published\_at, category }*

*// Return array of articles*

}

• Implement for each API:

• For NewsData.io: call the endpoint with category or keyword, get results.

• For TheNewsAPI: similar approach.

• For APITube: if it provides a general feed, call it and filter by language or region if needed.

• For NewsAPI.org: perhaps use it for specific queries if needed (it has similar fields, but content is truncated to 200 chars).

• For Google CSE: perform a search query (needs CX id and key) – it returns items with title, snippet, and link (we might later need to scrape if we use this).

• Normalize the category/tags:

• Each API may have different category naming. Map them to our internal categories. E.g., NewsData might return "category": "science" and TheNewsAPI might have "topic": "Sci/Tech". Create a mapping table or logic to unify these (like map “Sci/Tech” -> “Technology”).

• Attach the category or categories to each article object for use in personalization and filtering.

• **Breaking News Detection:**

• For trending topics, one idea is to use an API like Google Trends or Twitter trends. This is complex and may require additional API (Twitter API is not free and Google Trends doesn’t have an official API but can be scraped). As an alternative:

• Use the top headlines from a major source as “breaking news”. For example, call NewsAPI.org top-headlines (which gives general important news) or APITube’s world news feed.

• Identify if an article is “breaking” by recency (e.g., published in last 30 min) or by it appearing across many sources (if we see same title multiple times).

• Flag such articles or handle them with higher priority in the summarization step, if needed.

• (This can be refined later, but at least ensure top headlines are fetched frequently).

• **Database Insertion (Supabase):**

• After fetching and normalizing articles, insert them into the Supabase DB:

• Define the articles table in Supabase if not done: fields like id (UUID or bigserial), title, content (or description for short text), url, image\_url, source\_name, published\_at (timestamp), category, summary (to be filled later), etc.

• Ensure a unique constraint or index on something like url or combination of title+source+date to avoid duplicates. (We will explicitly handle dedup, but a DB constraint is a safety net.)

• Use the Supabase JS client (with service key) or Supabase REST API from the backend to insert new articles:

• We might insert in batch for efficiency. Supabase allows bulk insert: supabase.from('articles').insert([array of article objects]).

• Alternatively, upsert (insert or ignore if exists) can be used if a unique key is set. Supabase supports upsert() if you provide a conflict target.

• **Deduplication logic:** Before inserting, or by using upsert, ensure we skip articles already present:

• Keep track of article URLs or a hash of title+source. For example, create a hash or simple key from each article (like const key = title.toLowerCase().slice(0,50) plus source name).

• Compare against recent articles in DB (could query Supabase for existing items of the day, or maintain an in-memory list if the job runs continuously).

• If using upsert with url as the unique key, the DB will naturally ignore or update existing ones.

• Avoiding duplicates is crucial especially when multiple APIs might return the same news. Focus on deduping within a single run *and* across runs.

• Insert only new articles and log how many were added vs skipped (this can be part of monitoring).

• **Putting it Together – Cron Execution:**

• Create a script (e.g. cron/fetchNews.js) that orchestrates the above:

• For each category of interest (and possibly one for general top news), call the fetcher function for one or more APIs.

• Merge results into one list.

• Deduplicate the list.

• Insert into Supabase.

• Optionally, after insertion, immediately trigger summarization (Phase 6) for those new articles. This could be done here or via a separate cron job.

• Print or store some log output (like “Fetched 20 Tech articles, inserted 5 new”).

• While in development, you can run this script manually (node fetchNews.js) to test and populate some initial data.

• **Testing Aggregation:**

• Run the aggregator script for a few categories (perhaps using a limited query to not hit rate limits) and verify that data appears in the Supabase articles table.

• Check that the data is correctly stored: e.g., titles are proper, categories mapped, published dates in correct format, etc.

• Test the scenario of running it twice in a short span: ensure that previously inserted articles are not duplicated (the duplicate skip logic works).

• Also test a custom interest query if applicable (e.g., if a user interest is “OpenAI” as a keyword, try a search via one of the APIs to fetch relevant articles).

• *Deliverable:* **Automated News Ingestion** – The system can fetch news from multiple external sources and populate the database with the latest articles across key categories and topics. The aggregator handles multiple APIs, merges results, and avoids duplicate entries. At this stage, Newspaper.AI has a growing repository of news content in its database, ready to be processed by AI and served to users. The foundation for catching breaking news and comprehensive coverage is established.

**Phase 6: AI Summarization Pipeline with OpenRouter (RAG for Context)**

*Goal:* Implement an AI-powered summarization pipeline that takes the raw news articles from the database and produces concise summaries. We will use an LLM through OpenRouter (or directly OpenAI API) and employ Retrieval-Augmented Generation (RAG) techniques as needed to ensure summaries are accurate and relevant. This phase makes the bulk of the content easily digestible for users.

• **OpenRouter API Setup:**

• OpenRouter acts as a gateway to various large language models (LLMs). Sign up for OpenRouter (if required) and obtain API credentials. OpenRouter can route requests to models like OpenAI’s GPT-4/GPT-3.5, Anthropic’s Claude, etc., possibly offering better pricing or unified API.

• Decide which model to use for summarization: For cost-effectiveness and speed, **GPT-3.5 Turbo** is a strong choice for summaries. For higher quality or more nuanced summaries, **GPT-4** could be used but is slower and costlier. Possibly allow configuration to switch or use GPT-4 for important articles (like breaking news) and GPT-3.5 for others.

• Familiarize with OpenRouter’s API format (likely similar to OpenAI’s). Set the API base URL and key in environment variables. If OpenRouter is not available, plan B is to use OpenAI’s API directly.

• **Article Content Retrieval for Summarization:**

• We need substantial content to feed the LLM. The articles data we have might be just title and a short description (depending on API used). If full content is available (like from NewsData’s content field), use that. If not, consider retrieving more content before summarization:

• Some APIs give a URL for the article. We could perform a **web scrape** for the full text. This can be done with a library (for example, use an open-source tool like **node-readability** or **cheerio** to pull <p> text from the article URL). Another approach is an API like Mercury Parser or Diffbot, but those may cost money or have limits.

• For MVP, focus on summarizing whatever content we have (like the description). However, for a better product, implement at least a basic scraper for major sources:

• You might attempt to fetch the article URL, strip HTML tags, and cut out boilerplate. (Ensure not to get stuck on paywalled articles; skip those or only rely on description.)

• This could be an optional enhancement: the RAG approach can help here by retrieving content in chunks.

• If using RAG:

• **Retrieval-Augmented Generation** implies using a knowledge base to provide context to the LLM. For summarization, a use-case would be if the article is long or if we have multiple related articles:

• Break the article (or multiple articles on same topic) into chunks, index them in a vector store (Supabase Postgres has pgvector extension which can store embeddings).

• Use an embedding model (OpenAI embeddings or similar) to represent each chunk.

• At summarization time, query the vector store for the most relevant chunks (likely the chunks of that article itself, or related articles) to include in the prompt.

• This ensures the LLM sees the important parts of a long text even if we can’t feed the whole thing due to token limits.

• For initial implementation, RAG might be overkill if articles are short. We could instead:

• If article text > model token limit (say > 4000 tokens for GPT-3.5), then split it into halves or thirds. Summarize each part separately, then summarize the summaries together.

• This is a simpler chunking approach that can achieve a similar outcome without a vector DB. It does however accumulate error, so RAG with retrieval of key sentences could be better if needed.

• We will note RAG as an option for scaling to longer texts or multi-article summarization, but focus first on straightforward summarization per article.

• **Summarization Function Implementation:**

• Create a module summarizer.js (or integrate in the news fetch script) that takes an article (title & content) and returns a summary.

• Craft a prompt for the LLM:

• The prompt should instruct the model to produce a concise, unbiased summary of the article. For example:

"Summarize the following news article in 3-5 sentences, focusing on key facts and outcomes, and avoiding opinion or sensationalism:\n\n<ARTICLE TEXT>"

• If we want, add: "Output in a neutral tone and end with a short title or headline for the summary." (If we want the model to also suggest a headline or if we want bullet points, etc. But presumably a short paragraph summary is fine.)

• If using RAG and providing context, you would prepend the retrieved context or include it in the prompt as well.

• Call the OpenRouter (or OpenAI) API with this prompt:

• Use their completion or chat endpoint. For a chat model, the prompt would be in the user message. For example, send messages: [{role: "user", content: "<the prompt with article>"}].

• Set max\_tokens for the response (maybe ~150 tokens for summary to ensure brevity).

• Handle API response, extract the text.

• Include error handling: If the API fails or times out, catch the error and perhaps retry once or mark the article for retry later. Also be mindful of rate limits (OpenAI has requests/minute limits).

• To increase throughput, you might process multiple articles in parallel (but don’t spawn too many threads to avoid rate limiting). Perhaps summarize 3-5 articles concurrently if needed.

• **Efficiency considerations:** If a lot of articles come in, summarizing each can be costly. We could:

• Only summarize certain articles (e.g., those that are popular or match user interests) – but since our app is personalized, likely every fetched article could be relevant to someone, so better to summarize all to have them ready.

• Cache the summaries. Once summarized, store in DB (which we will do). If the same article appears again or an updated run happens, skip summarization if summary already exists.

• Use a cheaper model for initial summary and upgrade for important ones. For instance, GPT-3.5 for general use, but if an article is very important (maybe we define by being trending or requested by many users), use GPT-4 for a higher quality summary and update it.

• **Database integration:** Add a column summary in the articles table (text type) to store the AI summary. Also possibly summary\_time or a flag to mark it’s summarized.

• After obtaining the summary text, update the article record: supabase.from('articles').update({ summary: "..." }).eq('id', article.id).

• Alternatively, one could store summaries in a separate table if multiple versions or languages, but for now one column is fine.

• **Execution of Summarization:**

• Determine when this runs. Options:

• Run the summarization right after fetching new articles (in the same cron job or script). This way, as soon as articles are in DB, we call LLM and update them. This is simpler to implement sequentially in the cron workflow.

• Or run a separate scheduled job that periodically finds articles without summaries and processes them. For example, every 15 minutes, summarize any article that has summary field NULL.

• The separate job decouples fetching and summarizing (useful if one is much slower, e.g., summarizing might lag behind if a lot of news comes in). For MVP, doing it in one go is fine if performance is manageable.

• If doing sequentially in one job: after insertion of new articles, loop through them to summarize.

• Consider summarizing only new ones to avoid re-doing.

• If summarization fails for some, leave them for next run or log for manual check.

• If separate job: implement a script cron/summarizeNews.js that queries Supabase for articles missing summaries (e.g. select \* from articles where summary is null limit 10) and then processes them. This can run more frequently if needed.

• This approach also means if the fetch job runs hourly and adds 50 articles, a summarizer job running every 5 minutes can chip away at it rather than doing all 50 at once.

• For now, we’ll assume sequential to keep it simpler.

• **Testing Summaries:**

• Take a sample article (maybe insert a test row with some content) and run the summarization function in isolation to see the output. Ensure the prompt yields a well-formed summary (check for model compliance: sometimes AI might generate a headline or irrelevant text if prompt not clear).

• Adjust the prompt if needed (to enforce length or style).

• Run the pipeline on a handful of actual fetched articles. Then inspect the DB: each should now have a summary. Read those summaries to ensure they make sense and capture the essence (not too generic). Fine-tune prompt or model choice if quality is lacking.

• Monitor token usage if possible to estimate cost (OpenAI APIs return usage info). Ensure this is within budget given number of articles.

• *Deliverable:* **AI-Summarized News Content** – All news articles in the system are now accompanied by a concise AI-generated summary. The summarization pipeline is integrated such that whenever new articles are fetched, the system invokes an LLM (via OpenRouter) to generate their summaries. The use of RAG (if implemented) means even lengthy or complex stories are summarized accurately by providing the model with relevant context. This dramatically improves the user experience, as users will be able to read short summaries instead of long articles.

**Phase 7: Personalization Logic – “Why This Affects You”**

*Goal:* Implement the logic that provides each user with a personalized explanation of why a given news article is relevant to them. This feature increases user engagement by explicitly connecting news content to the user’s profile (interests, location, etc.). The outcome is a short note for each article like “Why this matters to you: …” when viewed by a specific user.

• **Utilizing User Profile Data:** Leverage the data collected in onboarding (Phase 3 & 4) – primarily:

• **Interests/Topics**: The categories or keywords the user selected.

• **Location**: The user’s city/region or country.

• These will be the main inputs to determine relevance. For example, if a user’s interests include “Technology” and an article is categorized as Technology, that’s an obvious connection. If the article mentions the user’s city or country, that’s another connection.

• If you stored additional info (like occupation or industry if that was part of onboarding, though not mentioned explicitly, some apps do ask for industry), that could be used too. (For instance, if the user works in Finance and the news is about stock markets, relevant.)

• **Defining Personalization Rules (baseline approach):**

• Start with simple rule-based matching:

• Match on **Category**: If article.category intersects with one of the user’s interest categories, then we have a direct reason: “This story is about , which is one of your chosen interests.”

• Match on **Keywords**: If we have a list of user-specific keywords or if the article’s content has keywords (e.g., if user is interested in “Artificial Intelligence” and the article summary contains “AI” or “artificial intelligence”), that’s a connection. We might scan the summary or title for any of the user’s interest keywords.

• Match on **Location**: If an article is about or datelined in the user’s city/state/country (for example, article might have metadata or content containing that location), then reason: “It’s happening in your area ().” Many news APIs give country or locale info for articles; use that if available (e.g., NewsData provides country of source).

• If an article source is a local news source corresponding to user’s location (e.g., user in UK and source domain ends in .uk or source name includes a city), that can also be inferred as local relevance.

• These rules can cover a majority of cases. Plan to compute a small text snippet for each:

• e.g., For interest match: why\_text = "Related to your interest in " + CategoryName.

• For location match: why\_text = "News from your region: " + LocationName.

• If multiple matches (say the user likes Tech and the news is Tech and also local), combine: maybe prioritize one or say “This tech news is local to you in London.”

• If no obvious match, we might not show a personalized note at all (or a generic: “Trending news you might care about”).

• **AI-Enhanced Personalization (advanced option):**

• For a more nuanced explanation, we can use an LLM to generate the “why it matters” sentence:

• Provide the model with the article summary or key points, plus the user’s profile interests and location.

• Prompt example: "Explain in one sentence why the following news might be important to a person interested in [user\_interests] and located in [location]: <summary>."

• The model might produce something like: “As a software developer in Seattle, this tech acquisition directly affects the local startup ecosystem you’re part of.”

• This can capture subtler connections (like linking an economic news piece to a user’s profession).

• The trade-off is calling the AI for each user-article pair, which is expensive if done for every item in every feed. To mitigate:

• Use it sparingly: perhaps only for the top 1-2 stories per user session, or only for premium users, etc.

• Or generate for articles in context of a generic persona and reuse. For instance, generate one “why it matters to tech enthusiasts” for a tech article, and then use that line for all users who have tech interest. But then it’s not personalized by location or individual profile.

• For MVP, it’s safer to stick to rule-based for realtime and maybe consider an AI approach triggered on demand (like if user clicks “Why?” then call AI to elaborate).

• We will implement rule-based first (fast and free), and note AI-generation as a later enhancement if time permits.

• **Implementing Personalization in Code:**

• Extend the backend or create a function that given a user profile and an article (with its summary and metadata) returns a short string explanation.

function getPersonalizedReason(article, userProfile) {

const reasons = [];

*// Interest match*

for (let interest of userProfile.interests) {

if (article.category == interest || article.summary.includes(interest) || article.title.includes(interest)) {

reasons.push(`related to your interest in \*\*${interest}\*\*`);

break;

}

}

*// Location match*

if (userProfile.city && article.summary.includes(userProfile.city)) {

reasons.push(`happening in your area (${userProfile.city})`);

} else if (userProfile.country && article.country == userProfile.country) {

reasons.push(`from ${userProfile.country}, where you live`);

}

*// Construct message*

if (reasons.length > 0) {

return "Why this matters to you: This story is " + reasons.join(" and ") + ".";

} else {

return ""; *// no personalized reason*

}

}

(The above is pseudocode; in practice, refine the string for grammar.)

• The article’s metadata like article.country or article.source\_name might be used for location matching. If our data doesn’t have explicit location, scanning the summary for city names is a heuristic (not always reliable).

• If using AI approach, implement a separate function that calls OpenAI with appropriate prompt and caches result.

• **Storing/Displaying the Personalization:**

• We have two ways:

1. **Compute on the fly in the frontend**: Fetch the user’s profile and the articles, then for each article in the React app, run a function to generate the “why” string. This keeps logic in front-end JS and uses the data available. This is feasible given the rules are light. The downside is if the logic is heavy or if using AI (can’t from frontend due to key exposure).

2. **Pre-compute and store**: E.g., when an article is fetched or summarized, you could compute which users it’s relevant for and store a mapping (but that is hard if many users; better dynamic).

3. **Hybrid**: Compute on the server per user request. For instance, when user opens their feed, an API could assemble the feed with personalized messages for each item.

• Simpler: do it in the frontend after retrieving data. We’ll proceed with that for now:

• When rendering an ArticleCard component, use the global user profile from context and the article props to call getPersonalizedReason(article, userProfile). If it returns a non-empty string, display it under the summary in a smaller italic font or so.

• Ensure to style it distinctively (maybe prefaced with an icon or different color, since it’s an AI/personal note).

• **Testing Personalization:**

• Create a test scenario: user profile with interest “Sports” and location “New York”. Article about a New York sports team. See if the logic outputs something like “Why it matters: This story is related to your interest in **Sports** and happening in your area (New York).”

• Try an article that has no overlap with interests – the function should return empty and the UI should simply not show a “Why it matters” line.

• If possible, test the AI approach offline with a sample prompt to gauge output quality. E.g., give ChatGPT a summary and fake profile and see the result. If it’s significantly better than rule-based, consider enabling it selectively.

• **Refinement:**

• Later, if analytics show users love the “why it matters” feature, we could invest in making it smarter (like the AI method).

• Also allow users to update their interests or location in profile settings and have it immediately reflect on personalization.

• *Deliverable:* **Personalized Context Messages** – The application now provides a short explanation for each news item, tailored to the user. When users scroll through their feed, they see not only summaries but also a brief note on why each article was chosen for them (e.g., it matches their interests or locale). This helps users connect with the content and understand its relevance, enhancing engagement. The personalization is currently rule-based (ensuring speed and scalability), with a framework in place that could be augmented with AI for even deeper personalization in the future.

**Phase 8: Categorization and Storage in Supabase with De-duplication**

*Goal:* Finalize the data layer by ensuring all news content is well-categorized, indexed, and free of duplicates in the Supabase database. This phase revisits the database design to refine how articles are categorized and stored, and strengthens de-duplication so users aren’t seeing repeated news. It also ensures the data model supports efficient queries for the feed.

• **Refine Database Schema:** Revisit the tables in Supabase and adjust as needed:

• **Articles Table:** by now it should have columns such as:

• id: Primary key.

• title: Text.

• url: Text (unique).

• source\_name: Text (e.g., “BBC News”).

• published\_at: Timestamp (with time zone).

• category: Text or an array of Text (if an article belongs to multiple categories, e.g., “Technology” and “Business”).

• country: Text (e.g., country code of source or news locale, if available).

• content: Text (full content or extended snippet, if stored).

• summary: Text (the AI summary).

• Optionally, image\_url: Text (thumbnail image for the article if provided by API).

• Perhaps a cluster\_id: if doing story clustering (not implemented yet, but could group related duplicates).

• **Profiles Table:** ensure it has fields:

• id (user id), interests (could be an array of text or a JSON field with list of interests), city, country, etc., as done.

• If interests is an array of text, Supabase can query that with contains operator if needed. Alternatively, you might have a join table user\_interests with one row per interest per user for easier querying. Given low complexity, an array is fine for now.

• **Saved Articles (optional)**: If planning for save-for-later in Phase 14, a saved\_articles table with user\_id and article\_id will be needed, but this can be added later.

• Apply any migrations via Supabase CLI or SQL to add missing columns (like if we hadn’t added country or image\_url yet).

• Use pgVector extension if planning any vector operations for RAG. Supabase allows enabling extensions – if RAG needed, enable vector and add a embedding column to articles where you store the article text embedding for semantic similarity queries (optional).

• **Deduplication Enhancements:**

• Ensure the unique constraint on url (or title+source if chosen) is in place in the database. In Supabase (Postgres), you can create a unique index: e.g., CREATE UNIQUE INDEX unique\_article ON articles(url);.

• Consider duplicates beyond identical URL:

• Sometimes the same news is published by multiple outlets (different URLs, different source\_name). We might want to treat those as duplicates in the feed.

• A potential solution: *Story Clustering*. This could be complex, but a simple approach:

• Compare new articles with existing ones by title similarity. For example, if the title or first sentence is 80% similar to an existing article (could use a basic string similarity or an embedding similarity if using vector), mark it as duplicate or part of the same cluster.

• If found, you might decide not to store it at all, or store but mark it as duplicate of a primary article.

• For MVP, we might not cluster aggressively, but ensure obvious duplicates are filtered.

• At least, if the title is exactly the same and date is same day, skip it. Implement this check in code if not already.

• Keep track of some identifier from sources: e.g., NewsAPI returns an id for articles if known (some don’t). If available, use it to filter.

• If implementing cluster, add cluster\_id:

• e.g., if two articles are essentially same story, assign them the same cluster\_id (perhaps the id of the first one). Then the feed could choose to show only one of them.

• This may be too much for now, but mentioning the design.

• **Categorization Consistency:**

• Ensure that the category field in each article is populated and standardized:

• If using free-form text for category (from different APIs), ensure it’s one of a known set. For example, you might have a predefined list of categories used in onboarding (Tech, Sports, etc.). Map any outside ones to these. E.g., if an API gave “economy” and you use “Business” as the category name, convert “economy” -> “Business”.

**Newspaper.AI Development Roadmap**

This roadmap outlines a **detailed, step-by-step implementation plan** for developing **Newspaper.AI**, an AI-driven personalized news aggregator. The plan is organized into phased milestones, each focusing on a key aspect of the project. Following these phases will ensure the application is **fully deployable, scalable, and ready for real-world usage**. Each phase includes objectives, tools/tech, tasks, and expected deliverables.

**Overview of Phases and Milestones**

**Phase** **Objective** **Key Tools/Tech** **Dependencies** **Deliverable**

**1. Setup** Project setup & environment configuration Node.js, React, Supabase, Git, CLI tools – (start) Base project structure & dev environment ready

**2. Arch Init** Frontend & backend architecture scaffold React (Vite/CRA), Tailwind, Node/Express, Supabase Phase 1 Running skeleton for frontend & backend (Hello World)

**3. Auth** Supabase authentication & onboarding flow Supabase Auth, React forms, Tailwind Phase 2 User sign-up/login and profile onboarding UI

**4. User Profile** Radar API integration & profile enrichment Radar (geolocation API), Supabase DB Phase 3 User profile with location/preferences stored

**5. Aggregator** News aggregator logic & external API integration TheNewsAPI, NewsData.io, APITube, Cron jobs Phase 4 (profile data available) Automated fetch of latest news into database

**6. Summarization** AI summarization pipeline (RAG for context) OpenRouter (LLM API), OpenAI, pgVector (optional) Phase 5 (articles in DB) AI-generated summaries for news articles in DB

**7. Personalization** Personalized context (“Why it matters to you”) User profile data, OpenAI (optional) Phases 4,5,6 Personalized relevance note for each article

**8. Storage & De-dup** Categorization & deduplication in DB Supabase (Postgres, SQL), indexing Phase 5 (data ingest) Structured DB with categories & no duplicate entries

**9. Frontend UI** Responsive news feed UI/UX React, Tailwind CSS, Framer Motion Phases 3,5,6 (auth & data ready) Polished web app UI (mobile-friendly, animated feed)

**10. Testing** Test-driven development & QA Vitest, React Testing Library, Playwright Phases 1–9 (features done) Comprehensive test suite (unit & end-to-end) passing

**11. Deployment** Deployment & scheduling configuration Render.com (Hosting & Cron), Docker (optional), Supabase CLI Phases 1–10 Deployed application (frontend + backend jobs) in production

**12. Monitoring** Logging and monitoring setup Sentry (error logs), PostHog or Plausible (analytics) Phase 11 (app running) Live error tracking and user analytics enabled

**13. Multilingual** Translation & multi-language support OpenAI or DeepL API, i18n libraries Phases 6–9 (content ready) News summaries available in multiple languages

**14. Enhancements** Optional features (save, notifications, dashboards) Service Workers (push), Chart libraries Phases 9–12 Added user features: bookmarks, notifications, insights

⸻

**Phase 1: Project Setup and Environment Configuration**

*Goal:* Establish the foundational development environment and repository for Newspaper.AI. This includes setting up version control, configuring the development stack, and preparing project tools to ensure all team members can develop and run the project consistently.

• **Initialize Repository & Version Control:**

• Create a new Git repository (e.g., on GitHub). Decide on a structure: a **monorepo** (single repository containing both frontend and backend code) or separate repos for each. A monorepo (e.g., using a tool like Nx or Turborepo) might simplify shared code and deployment.

• Add a README.md with project overview and setup instructions. Include a .gitignore for Node/React (ignore node\_modules, build outputs, .env files, etc.).

• Set up branch protection or a git-flow convention (like main for stable releases, dev for integration).

• **Development Environment Setup:**

• **Node.js & Package Manager:** Install Node.js (LTS version) on all dev machines. Use a version manager (nvm) and commit an .nvmrc specifying the version. Choose npm or Yarn (or pnpm) and initialize with npm init or equivalent.

• **Base Dependencies:** Add global and local dependencies:

• **Global CLI Tools:** Install the **Render CLI** (for deployment in Phase 11) via npm install -g render-cli. Also install **Supabase CLI** via npm install -g supabase.

• **Local Dev Dependencies:** Set up **TypeScript** (if using), linters, and formatters. For example, npm install -D typescript @types/node @types/react eslint prettier. Initialize tsconfig.json for both client and server if needed.

• Create a basic folder structure:

• If monorepo: a frontend/ and backend/ directory. Otherwise, single repo, but you can separate concerns within it.

• Within frontend, a src/ for source code; within backend, maybe a src/ or just top-level scripts for cron jobs.

• **Frontend Bootstrapping (React + Vite):**

• Use Vite to create a React app (with TypeScript template): e.g. npm create vite@latest frontend -- --template react-ts. This sets up a lightweight React project.

• Verify it runs: cd frontend && npm install && npm run dev. You should see the Vite dev server welcome page at http://localhost:5173 (or similar port).

• Install **Tailwind CSS** in the React project:

• npm install -D tailwindcss postcss autoprefixer and run npx tailwindcss init -p to create tailwind.config.js and postcss.config.js.

• Configure tailwind.config.js content paths to include your React src files (e.g., ./index.html, ./src/\*\*/\*.{js,ts,jsx,tsx}).

• Import Tailwind in your CSS (e.g., create src/index.css with @tailwind base; @tailwind components; @tailwind utilities; and import this in your main entry).

• Test by adding a Tailwind class to App.tsx (e.g., <h1 className="text-2xl font-bold">Hello</h1>) and ensure it styles correctly.

• Install **Framer Motion** for animations: npm install framer-motion. Do a quick test animation (like wrapping a component with <motion.div animate={{ opacity: 1 }} initial={{ opacity: 0 }}>) to verify setup.

• **Backend Bootstrapping (Node.js Service):**

• Create a simple Node.js project (could be within backend/ folder). npm init -y to get a package.json. If using TypeScript, set up tsconfig.json.

• Decide if backend will be a persistent server (Express/Node API) or just a set of scripts (for cron jobs and maybe an API route for the feed). For now, implement a minimal Express server for health-check and to possibly serve an API route:

• npm install express and create an index.js (or index.ts) that does app.get('/', (req,res) => res.send('API running')) and listen on a port (maybe 5000).

• Test by running node index.js (or ts-node index.ts if TS) and hitting http://localhost:5000 to see the response.

• This Express server can later have routes for triggering the aggregator or providing the feed to the frontend if needed. It’s also optional if the frontend can directly query Supabase or other APIs.

• Set up a basic project structure for backend scripts:

• For example, a tasks/ folder for cron jobs (like fetchNews.js for Phase 5).

• Ensure you can run those with Node independently of the Express server (this is relevant for Render’s separate cron jobs).

• **Supabase Project Initialization:**

• In the Supabase dashboard, create a new project (with a Postgres database). Note down:

• **Supabase URL** and **Anon/Public Key** (for client usage).

• **Service Role Key** (for backend admin usage).

• In the codebase, create environment variable files:

• In frontend, create a .env.local with VITE\_SUPABASE\_URL and VITE\_SUPABASE\_ANON\_KEY. (Vite exposes env variables prefixed with VITE\_ to the client code.)

• In backend, create a .env with SUPABASE\_URL, SUPABASE\_SERVICE\_KEY and any other secrets (API keys for news APIs, OpenAI, etc., will be added as we get them).

• Install Supabase client libraries:

• Frontend: npm install @supabase/supabase-js in the React project.

• Backend: also install @supabase/supabase-js to use the service key for DB operations.

• **Verification & Documentation:**

• Ensure both frontend and backend run without errors. Commit all changes to Git.

• Document in README:

• How to run the frontend (npm run dev) and backend (node index.js or npm run start).

• List the environment variables required and how to set them (without exposing secrets in the repo).

• Optionally, set up **Prettier** and **ESLint** configs and maybe a pre-commit hook (Husky) to auto-format code on commits. This ensures code consistency.

• *Deliverable:* **Project Skeleton & Env Configured** – A working base project with React/Tailwind frontend and Node backend. The development environment is standardized with configuration files and scripts. Supabase is set up and ready to use (although not yet integrated in code). At this milestone, collaborators can start the app locally (frontend and backend) and see a basic page or API response, confirming the environment is correctly configured.

**Phase 2: Frontend and Backend Architecture Initialization**

*Goal:* Establish a solid architecture for both the frontend and backend. This includes structuring the codebase, setting up common components and services, and ensuring the frontend-backend integration is prepared for upcoming features.

• **Frontend Architecture & Routing:**

• Organize the React project structure into logical modules:

• Create directories like components/ (reusable UI pieces), pages/ or routes/ (page-level components for different screens), contexts/ (for context API usage), and utils/ (helper functions).

• Install **React Router**: npm install react-router-dom. Set up a router in App.tsx or a separate Routes.tsx. Define basic routes:

• /login (LoginPage component),

• /signup (SignupPage or combined with login),

• /onboarding (for new users to select interests),

• /feed (main news feed page),

• /saved (for saved articles, optional),

• maybe /settings (to edit profile).

• Implement placeholder components for these pages (just simple headers or divs indicating the page).

• Ensure navigation works (you can add a temporary nav menu for dev to jump between routes).

• Integrate Tailwind CSS styles:

• Configure a base style (maybe in index.css) for overall app (like setting a background color, min-height, etc.).

• Use a consistent design system for spacing, font sizes, colors. Tailwind’s default is fine, but if you have a theme, configure in tailwind.config.js (e.g., custom color palette for primary/secondary).

• Ensure responsiveness by using Tailwind’s responsive utility classes (mobile-first). For example, design the feed page layout to stack on small screens and show side-by-side on larger screens.

• Plan global state management for auth and profile:

• Create an AuthContext in contexts/AuthContext.tsx that provides the current user and profile data to the app. Use React’s createContext and useState or useReducer to manage login state.

• Provide this context at the app level (wrap <Router> with <AuthProvider>).

• The context will be populated in Phase 3 when we implement login, but scaffold it now with a value like { user: null, profile: null, setUser: () => {} } to satisfy TypeScript.

• **Backend Structure & Supabase Integration:**

• If using Express, set up routes structure:

• Create a router for any needed endpoints, e.g., router.get('/feed', ...) to serve combined feed data if the frontend were to call a backend endpoint (though frontend can directly query Supabase, we might not need many backend routes besides health).

• Consider if you need any server-side secure operations (for example, computing personalized recommendations or proxying third-party API calls to hide keys). Keep endpoints ready for such cases.

• Supabase Admin Setup (Backend):

• Using the Supabase service key, test a simple DB operation. For instance, in your backend index or a test script, try:

const supabaseAdmin = createClient(SUPABASE\_URL, SUPABASE\_SERVICE\_KEY);

const { data, error } = await supabaseAdmin.from('profiles').select('\*');

This should work (though profiles is empty at start). Handle any connection errors now.

• Decide how to use Supabase from the frontend:

• You will use the anon key on the client side for auth and fetching user-specific data (Supabase’s client-side security via RLS will protect data).

• Plan to use Supabase’s real-time or subscription features (optional) for live updates of news? Supabase can send real-time on table changes, but that might be overkill for news updates. We might skip for now or consider it for notifications.

• Code Sharing (Monorepo considerations):

• If monorepo, set up a shared folder or package for types or utilities used by both front and back. For example, define a types.ts interface for Article and Profile:

interface Article {

id: string;

title: string;

url: string;

source: string;

published\_at: string;

category: string;

summary?: string;

*// ...*

}

Use this in both places to ensure consistency.

• If separate repos, you could still duplicate type definitions or use a tool to generate types from the database (Supabase can generate types via their codegen).

• **Security & CORS:**

• If the React app will call the backend (for any reason, such as protected routes), enable CORS in Express: e.g., npm install cors and app.use(cors({ origin: '\*' })) or specific origin. This allows browser to call your Node endpoints.

• However, if the frontend mostly interacts with Supabase directly and the backend is just for cron jobs, you might not need CORS (no direct calls).

• **Initial Third-Party Integrations:**

• Although full integration will happen in later phases, ensure the app is ready to incorporate:

• **OpenAI/OpenRouter**: Install their SDK or prepare to call via HTTP.

• **Radar**: Possibly install Radar’s JS SDK in frontend if available via script include or npm.

• **News APIs**: No need to install, will use fetch/axios. But consider installing axios for ease of HTTP calls in Node.

• Manage these keys via .env and ensure not exposed in client (client will only have anon key, others like OpenAI keys remain in backend).

• **CI/CD Setup (Continuous Integration):**

• Set up a GitHub Action workflow (or similar CI) to run on push:

• Install dependencies, run npm run build for frontend to ensure no build errors.

• Run tsc --noEmit to catch type errors in both projects.

• Possibly run eslint . to enforce lint rules.

• This ensures that as we proceed, we maintain code quality. (Testing will be added in Phase 10.)

• **Milestone Check:** Start the application end-to-end:

• Run the backend (if it’s just a script or Express, start it).

• Run the frontend dev server.

• The frontend likely doesn’t show useful data yet, but ensure it loads without errors, and if it tries to call any API (like maybe a test call to /feed), that it’s handled or commented out.

• The structure is now in place to start building actual features (auth, data fetching, etc.).

• *Deliverable:* **Initial Architecture Established** – Both frontend and backend have a clear structure. The React app has routing and theming set up, and the Node backend is scaffolded for API routes or cron tasks. Supabase connectivity is verified from both client and server. This foundation allows developers to implement features without worrying about basic setup, and ensures the project is organized for scalability and maintainability.

**Phase 3: Supabase Authentication and Onboarding Flow**

*Goal:* Implement user authentication (sign up & login) and an onboarding flow to gather user preferences. This ensures each user has a profile in the database with information that will later drive personalization.

• **Supabase Auth Configuration:**

• In the Supabase dashboard, confirm **Email/Password** auth is enabled (this is default). Optionally enable OAuth providers (Google, GitHub, etc.) if desired: this involves obtaining client IDs/secrets for each provider and setting them in Supabase Auth settings.

• Prepare email templates for verification or password reset if using the Supabase default (for MVP, default is fine; the user will get a generic Supabase-branded email when signing up unless you customize SMTP).

• Ensure the profiles table exists (Supabase quickstart usually has it). If not, create a table profiles with columns:

• id: uuid (primary key, matches user’s auth uid).

• username: text or full\_name for display name.

• avatar\_url: text for profile picture (if you allow uploading or use Gravatar).

• interests: text[] (an array of topics the user cares about).

• location\_city: text, location\_region: text, location\_country: text to store location info (we’ll fill these in Phase 4).

• created\_at: timestamp (default now()).

• Enable Row Level Security (RLS) on profiles and write a policy: users can select and update their own profile (auth.uid() = id condition).

• **Sign Up / Login Pages (Frontend):**

• Create a **Signup form** component:

• Fields: Email, Password (and maybe Confirm Password).

• On submit, call supabase.auth.signUp({ email, password }). If using email confirmation, inform the user to check email. If not, the user is directly logged in.

• Handle errors (e.g., email already exists) and display messages.

• Create a **Login form** similarly:

• Fields: Email, Password.

• Use supabase.auth.signInWithPassword({ email, password }).

• On success, Supabase will return a session and set a cookie (if using the provided auth context).

• After login or sign-up, ensure the app knows the user is authenticated:

• The AuthContext created earlier can subscribe to auth changes. For example:

supabase.auth.onAuthStateChange((event, session) => {

*// set user in state if session exists, or null if signed out*

});

• Alternatively, after successful signIn, manually set the user in context and fetch their profile.

• Protect routes: Implement a higher-order component or use <PrivateRoute> pattern to redirect unauthenticated users. For React Router v6, you can do:

{user ? <Outlet /> : <Navigate to="/login" />}

in route definitions, to guard certain paths.

• Add a logout option: supabase.auth.signOut() (e.g., a button in a navbar or settings page).

• **Onboarding Flow:**

• After first sign-up, you want to capture user interests and possibly other data:

• One approach: use a boolean field in profile like onboarding\_complete. After sign-up, check this; if false, redirect to /onboarding.

• The onboarding page(s) ask for interests and location permission (location handled in Phase 4, but design the step now).

• **Interests Selection UI:**

• Decide on a set of interest categories to present (could be a multi-select of, say, 8–10 broad topics: World, Business, Technology, Sports, Entertainment, Science, Health, Politics, etc.).

• Represent them as a list of options with checkboxes or toggles. Use Tailwind to style (for example, a grid of cards or pills that can be toggled active).

• Allow the user to pick at least a few (or skip if they want a generic feed).

• If you have a broader idea (e.g., letting them also input custom keywords, or choose subtopics), that can be text input or additional step. But keep MVP simple with predefined categories.

• Store the selected interests in component state for now.

• **Location Opt-In UI:**

• Ask the user if they want to enable local news. Provide two main options: “Use my current location” (with a description of why it’s useful) and “Skip” or “Enter location manually”.

• If “Use current location”: you will later prompt the browser for geolocation (Phase 4 will handle the actual coordinates fetch). For now, perhaps just capture intent.

• If “Enter manually”: provide input for city and country (or zip code). This could also be handled later via a geocoding API, but simply storing their input is possible.

• If skip, they’ll still proceed without location data.

• **Completing Onboarding:**

• When the user submits the onboarding form:

• Write the profile data to Supabase. For example:

supabase.from('profiles').upsert({

id: user.id,

interests: selectedInterests,

location\_city: city,

location\_country: country

});

Use upsert so it creates or updates the profile row (Supabase may have auto-created a profile row with some default via trigger on sign up, depending on the starter template; check and adjust accordingly).

• Mark onboarding as complete. You could set a onboarding\_complete: true in profile (and have your RLS allow the user to set that).

• After saving, redirect to main /feed page.

• Ensure to handle any Supabase write errors (log or alert user).

• **Profile Fetching:**

• On app startup (or login), fetch the user’s profile to know their interests:

const { data: profile } = await supabase.from('profiles').select('\*').eq('id', user.id).single();

setProfile(profile);

• This profile info will be used for personalization later, and possibly to tailor the feed queries (e.g., fetch certain categories by default).

• **UI/UX Refinement:**

• Make the onboarding flow visually appealing and easy:

• Possibly split into multiple steps/screens (multi-step form). You can use a state machine or multi-route (like /onboarding/step1, step2, etc., or a single component that changes content).

• Use Tailwind to highlight selected interests (e.g., change color or icon when selected).

• Provide guidance text like “Select at least 3 interests to customize your news feed.”

• Mobile optimization: ensure the selection grid wraps on small screens and buttons are easy to tap.

• **Testing the Auth Flow:**

• Create a new user and walk through the process fully:

• Ensure account creation works, profile is created in DB, and feed page is reached.

• Log out, log in with the same user: you should be taken directly to feed (not onboarding, since completed).

• Try logging in with wrong password to see error handling.

• Test an OAuth login if enabled (e.g., sign in with Google) – ensure profile still gets created (you might need a trigger function or post-sign-in step to insert a profile row for OAuth users, since signUp hook is only for email).

• Ensure the app doesn’t allow navigating to feed without auth (try manually going to /feed).

• *Deliverable:* **User Authentication & Onboarding Implemented** – Users can register and log in to Newspaper.AI. New users are guided through selecting interests and location preferences. These details are stored in their profile (Supabase DB). The application now knows each user’s preferences, setting the stage for personalized news retrieval. The auth flow is smooth and secure, with protected routes and the ability to log out, meeting all basic identity management needs.

**Phase 4: Integration of Radar API and User Profiling**

*Goal:* Integrate the Radar geolocation API to enhance user profiles with location data, and finalize the storage of user preferences. This allows the app to tailor news content based on the user’s geographic context.

• **Radar API Setup & Configuration:**

• Create an account on **Radar** (radar.io) and obtain the API publishable key (for client-side use) and secret key (for server-side if needed). Radar provides SDKs for web and mobile to get user location and geofencing.

• Decide on approach: likely, use the **Radar JavaScript SDK** on the frontend to handle location permissions and geocoding:

• Install Radar SDK via script tag or npm (npm install radar-sdk-js).

• Initialize it with Radar.initialize('pub\_key') in the app (maybe during onboarding when needed).

• If not using the SDK, plan to call Radar’s REST API. But using their SDK simplifies geolocation prompts and data handling.

• **Implementing Location Capture (Frontend):**

• In the onboarding flow, when user chooses “Use my current location”:

• Call await Radar.getLocation() (or the appropriate method from their SDK) to prompt browser geolocation. This returns latitude and longitude.

• Then call Radar.reverseGeocode({ latitude, longitude }). This will return address components. Extract city, state (region), country from the response:

const result = await Radar.reverseGeocode({ latitude, longitude });

const { city, state, country } = result.addresses[0];

(Check Radar’s API docs for exact response structure).

• Handle errors: If user denies permission or it times out:

• Fallback: you might default to using their IP for an approximate location via Radar.ipGeocode() (less accurate) or just set location as null and proceed.

• Or allow manual entry as you offered: if user provided a city manually, use that.

• Immediately update the onboarding state with obtained location (so user can confirm if needed, or just silently use it).

• As part of profile upsert (from Phase 3), now include location\_city, location\_region, location\_country fields with the values from Radar.

• If user skipped location:

• You might leave those fields null, or if they entered manual location, attempt to geocode that manually later (e.g., using Radar forward geocode by address).

• **Server-side Consideration:**

• If you need server logic (not strictly necessary here), you could send the coordinates to an Express endpoint that calls Radar’s secret API. But since we can do on client with public API, it’s fine.

• However, one benefit of server-side: you might not want to expose the Radar public key or you might want to enrich data server-side (Radar can also store user location info or track them over time, which might be beyond our scope).

• We’ll proceed with client for simplicity.

• **Profile Table Adjustments:**

• Ensure the profiles table has the columns for location as planned. If not, add them via a migration or using Supabase table editor.

• Set appropriate data types: text for city and region, maybe text for country code or country name. (Could use country code for easier filtering in news APIs.)

• If needed, update RLS policy to allow updates to these fields.

• **Personalization Data Expansion:**

• Think of other profile attributes that might be useful:

• Perhaps preferred\_language (for Phase 13 translations).

• Or interests\_keywords if user adds custom interests not just categories.

• These could be added now or later. For now, stick to core interests and location.

• The Radar integration primarily gives location. Radar also has features like **place visits** or **moving/stopped state** which we probably don’t use. We mainly need static location for news relevance.

• **Verify and Test:**

• In a development environment, test the location capture:

• Use a device or browser that can simulate location (Chrome dev tools can simulate geolocation coordinates). Alternatively, temporarily use a manual call with lat/lng of a known city.

• Ensure Radar returns the expected city name. (If Radar is problematic or requires https even in dev, make sure to run dev server with https or deploy a test).

• Complete onboarding and check in Supabase that the profile row has city and country populated.

• Test updating: maybe add a way for user to update location in profile settings later. But not needed for initial release.

• Test logins of existing users (who have location set) to ensure profile fetch brings in location data properly (no issues with new fields).

• If multiple users sign up from different locations, confirm that data is isolated and correct for each.

• **Privacy Consideration:**

• Make it clear to users why you ask for location and that it’s optional. Possibly update terms or a tooltip about data usage.

• Ensure that if a user revokes location (not easily done in web without re-onboarding), your app doesn’t break (we can assume location is static unless user makes a new profile or you allow editing).

• *Deliverable:* **Location-Enriched Profiles** – The onboarding now successfully captures the user’s location (with permission) and stores city/region/country in their profile. The application has integrated Radar API for geocoding. Each user’s profile in Supabase now has both their interests and location, providing the key inputs for personalized news relevance in upcoming phases. The foundation for location-based news filtering or context (“news near you”) is laid.

**Phase 5: News Aggregator Logic and API Handling**

*Goal:* Build the core news aggregation engine to fetch news articles from multiple external sources and store them in the database. This includes integrating various News APIs, handling their responses, merging results, and scheduling regular fetches. We aim to gather a comprehensive set of news articles that will later be filtered and served to users.

• **Set Up API Access Keys:**

• Obtain API keys for the news sources to be used:

• **NewsData.io** – Sign up and get an API key (free tier allows a good amount of requests and includes full content).

• **TheNewsAPI** – Sign up for a free API token.

• **APITube.io** – If using (it offers real-time news from many sources), get an API key.

• **NewsAPI.org** – (Optional fallback, since it’s well-known but has limits on free tier).

• **Google Custom Search (News)** – Create a Custom Search Engine that includes news sites or use Google News as a site. Get a CSE ID and API key if needed for search fallback.

• Store these keys in environment variables (in backend .env): e.g., NEWSDATA\_API\_KEY, THENEWSAPI\_TOKEN, APITUBE\_KEY, etc.

• Note: Also consider **Bing News Search API** (Microsoft Azure) as an alternative; it has a free tier and might fetch diversified sources. For now, stick to the ones listed unless needed.

• **Database Schema for Articles:**

• Create a table articles in Supabase with columns:

• id: uuid (primary key, default gen\_random\_uuid()).

• title: text.

• url: text.

• author: text (if available).

• source: text (name of news source or site).

• published\_at: timestamp with time zone.

• content: text (full article content or extended summary, if available).

• summary: text (for AI summary, to fill in Phase 6).

• image\_url: text (thumbnail image, if API provides one).

• category: text (general category like “Sports”, “Tech”).

• country: text (country code or name, if relevant).

• Possibly an index on published\_at for sorting and on category for filtering.

• Add a unique constraint on url (since URLs are usually unique per article).

• If Supabase allowed it, consider a composite unique on (title, source, date) to catch cases of same title on different sites (though that can also eliminate similar titles that are actually different news).

• **Implement Fetch Logic (Backend):**

• Create a Node script (e.g., tasks/fetchNews.ts or .js) that will:

1. **Determine which API to call for what**: for each category or interest, choose a source.

• e.g., For each category [“World”,“Business”,“Technology”,“Sports”,…]:

• Call NewsData.io’s endpoint with category=technology (if supported) or a keyword query.

• TheNewsAPI has an endpoint for top stories by category as well.

• APITube might not need category (it returns a firehose feed; if so, filter on our side).

• Also include a general “Top headlines” fetch for global breaking news.

• If user-specific topics (like custom keywords) are needed: we might handle those on the fly per user or as part of daily cron. Possibly out-of-scope for now, focus on broad categories.

2. **Fetch from APIs**:

• Use axios or node-fetch to call APIs. E.g.:

const url = `https://newsdata.io/api/1/news?apikey=${API\_KEY}&category=technology&language=en`;

const res = await axios.get(url);

const articles = res.data.results;

• Do similar for TheNewsAPI:

const url = `https://api.thenewsapi.com/v1/news/top?api\_token=${TOKEN}&locale=us&language=en&categories=tech`;

• Ensure to handle pagination or max results (limit to, say, 20 per category to avoid overload).

• For each API, map the response to our unified article format:

• Title, URL, Source (maybe from the response or parse the URL’s domain if not given), published\_at, content/description.

• Use the category that we fetched for as the category field (APIs might also return a category or keywords).

• If an image URL is given (NewsData and TheNewsAPI often provide an image link), include it.

• If any API fails (network issue or rate limit), catch and maybe continue with others.

3. **Merge and Deduplicate**:

• Combine articles from all sources into one list.

• Deduplicate by URL (easiest key). Also check title similarity to avoid near-duplicates:

• Implement a simple check: lower-case titles, remove punctuation, and see if two titles start with the same 10-15 words. If yes, consider them duplicates. Keep one (maybe prefer the one with more content or better source).

• Remove any non-English articles if our focus is English (or handle languages separately in Phase 13).

4. **Insert into Database**:

• For each unique new article, insert into Supabase articles.

• Use upsert with onConflict: url so that if the URL exists, it doesn’t double insert.

• Supabase JS example:

const { error } = await supabaseAdmin.from('articles').upsert(newArticles);

where newArticles is an array of article objects.

• If number of articles is large, you might batch them (Supabase can take 100 at a time).

5. **Logging**:

• Print out or log how many fetched, how many inserted, for monitoring.

• If running this script manually, it helps to see progress. In production, logs will be in Render or similar.

• **Category list decisions:**

• Likely categories: “World” (general/news), “Business”, “Technology”, “Sports”, “Entertainment”, “Science”, “Health”, “Politics”.

• If a certain API doesn’t have a category, use keywords or source targeting (e.g., for World, maybe no filter needed just top news).

• For “Breaking News”: could use APITube’s realtime feed or just rely on frequently updating top stories from one source.

• Optionally integrate a **trends API**: e.g., fetch trending keywords from Google Trends and then search those keywords via a news API. This can capture viral topics. Mark those as trending.

• **Scheduling Plan:**

• You might not implement scheduling in code, but decide frequency:

• Fetch top headlines (World/General) every 15 minutes.

• Fetch each category every hour.

• Perhaps also fetch each user’s interest daily (if custom interests beyond categories).

• We will implement scheduling via Render in Phase 11. For development, we manually run or trigger fetches.

• **Testing the Aggregator:**

• Run node fetchNews.js (or TS compile then node) locally.

• Check console output and then inspect Supabase articles table via the dashboard:

• See that rows have been inserted with correct fields.

• Verify no duplicates (URLs unique).

• Check a couple of entries: do we have content or at least a decent description? If APIs returned only short snippets, note that (we might need to extend content via scraping in summarization phase).

• If one API’s results are too limited or have missing fields, adjust or call an alternate API for that category.

• For example, if TheNewsAPI has no data for “Science”, try NewsData or vice versa.

• Test that our upsert logic doesn’t create duplicates on subsequent runs:

• Run the script again immediately; it should ideally insert zero new items because all are duplicates. If it does insert duplicates, refine the conflict key or check logic.

• Also test filtering by date: perhaps only fetch articles from last X hours to avoid the APIs giving you older news repeatedly. Some APIs allow a date filter or you can filter by checking published\_at against last stored timestamp.

• E.g., store last fetch time per category in a file or table to pass as param next time.

• Simpler: trust the APIs to give recent stuff first and skip older by dedup, but be careful not to miss new ones that are older but not seen.

• **Edge Cases:**

• If an API has usage limits (say 100/day), ensure our schedule won’t exceed that. Possibly mix usage across multiple APIs to balance.

• NewsAPI.org free plan doesn’t include full content and has small limits; maybe keep it as backup.

• If APITube is free and broad, lean on it for volume (but verify its reliability).

• Consider adding **RSS feed parsing** for sources not covered by APIs (could be a later improvement, not this phase).

• *Deliverable:* **Automated News Ingestion** – The system can pull in news articles from external sources and populate the Supabase database. This includes a variety of categories and general top news. The news fetch script handles merging multiple sources and avoids duplicate entries. We now have a growing repository of news content (titles, descriptions, etc.) stored in our database, updated on a regular schedule. This is the raw material that will be processed by AI and presented to users.

**Phase 6: AI Summarization Pipeline with OpenRouter (RAG for Context)**

*Goal:* Implement an AI pipeline to summarize news articles and generate concise headlines, making the content easily digestible for users. We will use the OpenRouter API to access advanced language models (LLMs). Additionally, consider Retrieval-Augmented Generation (RAG) techniques to provide context from the article content to the model for accurate summaries.

• **OpenRouter / LLM API Setup:**

• If you have an OpenRouter API key, configure it. OpenRouter acts as a proxy to models like GPT-4, GPT-3.5, etc., often with an OpenAI-compatible API. If not, use OpenAI API directly (with an API key from OpenAI).

• Save the API key and endpoint in environment variables (e.g., OPENROUTER\_API\_KEY and base URL if needed).

• Install any needed SDK or use fetch:

• OpenAI: npm install openai or just REST calls.

• OpenRouter might be used via the same openai SDK by pointing basePath to OpenRouter’s URL.

• Decide on model: **GPT-3.5 Turbo** (fast, cost-effective) for summarization; consider GPT-4 for very important articles (perhaps optional).

• **Content Retrieval for Summaries:**

• We need enough article content to summarize. Our articles.content from Phase 5 might be short (depending on API):

• NewsData.io provides a content field which can be full text or a large excerpt (they claim full content often). If we have that, great.

• If content is short or just a snippet, consider scraping the article URL:

• Use a library like **node-fetch** + **cheerio** to fetch the HTML of the article and extract <p> tags from the main content area.

• Alternatively, use an API like Mercury or a headless browser to get text (likely too heavy, skip for now).

• For MVP, you might attempt scraping just for a few known domains or leave this for later if summary quality is okay from the snippet.

• If not scraping, the model will be summarizing from the description/snippet. That might be okay but not ideal if snippet is incomplete.

• **RAG Approach:**

• If full content is too long for the model’s context window:

• Use RAG: break content into chunks (e.g., 500 tokens each). Store these chunks in a vector database to retrieve relevant parts for summary:

• Supabase can act as a vector store by enabling pgvector. If you want, enable the vector extension and add a column embedding vector(1536) for the content embedding.

• Use OpenAI’s text-embedding-ada-002 to create embeddings for each chunk and store them.

• At summary time, fetch the top 2-3 chunks that are most similar to maybe the title (or an initial summary question).

• But implementing full RAG might be too involved. Instead, a simpler method:

• If content > 4000 tokens, split and summarize in parts, then summarize those summaries.

• Or ask the model to focus on the first part, then second part, etc., but that’s complicated.

• Given time, we may not implement a full vector search. We can mention RAG as future scalability and ensure our architecture can accommodate it (hence storing embeddings optionally).

• **Summarization Prompt Design:**

• Craft a clear prompt for the LLM. Some possibilities:

• For a chat model:

{"role": "user", "content": "Write a brief news summary for the following article:\n\n\"\"\"\n[Full article text or content snippet]\n\"\"\"\n\nThe summary should be 3-5 sentences long, factual, and unbiased. Also provide a short headline."}

• You could instruct it to output a JSON with summary and headline keys if structuring output, but keeping it simple text is fine.

• Emphasize neutrality and conciseness (to avoid AI adding opinions or being verbose).

• If you plan to attach context, you might add after the article text: "Context: The user is interested in [user interests]." but better to handle “Why it matters” separately (Phase 7).

• Keep track of token usage: limit the input. If article content is large, consider truncating less relevant parts or focusing on first N chars (though important info might be later).

• **Batch Processing:**

• Summarize multiple articles in one go or sequentially?

• OpenAI API can handle multiple requests concurrently (respect rate limits). Maybe process 5 at a time to speed up.

• But beware of hitting request limits or running up cost if many articles. Implement a limit per run.

• Possibly prioritize summarizing more important categories first (or ones likely to be seen by users).

• If on a schedule, ensure summarization finishes before next fetch overlap.

• **Updating Database with Summaries:**

• After obtaining the summary (and maybe a generated headline if we do that), update the articles table:

await supabaseAdmin.from('articles').update({ summary: summaryText, headline: shortHeadline }).eq('id', article.id);

(Add a headline column if you want a separate AI-generated title different from original; or reuse original title if fine).

• Alternatively, store AI summaries in a separate table if you want to track versions or multi-language, but not necessary.

• Ensure that summary is not too long for the column (text type in Postgres can handle pretty large text, so fine).

• **Execution Strategy:**

• Option A: **Inline Summarization** – Integrate into the fetch script. After inserting new articles, immediately summarize them:

• Pros: simpler flow, always summarizes whatever fetched.

• Cons: if fetch frequency is high, could backlog or slow the fetch job.

• Option B: **Dedicated Summarizer Job** – separate script tasks/summarizeNews.js that runs every X minutes:

• Query Supabase for articles where summary is null (and maybe published\_at > some recent threshold to not summarize old stuff unnecessarily).

• Summarize those and update.

• This decouples fetching from summarizing.

• For now, implement Option B for clarity and reliability:

• For development, you can run it manually after fetch to populate summaries.

• In production, schedule it to run every 10-15 minutes.

• **Testing Summaries:**

• Pick a specific article from the DB that has content and run it through the summarization function (maybe build a small test harness).

• Inspect the output: does it capture the article? Is it factual? If the article content was short, the summary might be almost identical to original blurb. That’s okay but not great; perhaps in those cases, having multiple sources to combine would be useful (out of scope for now).

• If the summary is too short or too long, adjust prompt instructions or the number of sentences desired.

• If it injects any subjective phrases like “important” or “fortunately”, tweak prompt to say “just facts”.

• Test with a variety: a tech article, sports game recap, political news. The model should adapt.

• **Error Handling & Retries:**

• If an OpenAI API call fails (network or API error), implement retry logic with backoff (maybe try up to 3 times).

• If a particular article consistently fails (maybe content too large), you might mark it to skip (set summary to an empty string just to not keep retrying).

• Log any errors to a file or console for later debugging (later, Sentry will catch exceptions too).

• *Deliverable:* **AI-Summarized Articles** – Every news article stored in the database is now accompanied by a concise AI-generated summary (and optionally an AI-crafted headline). The summarization pipeline uses OpenRouter/OpenAI to condense articles into key points, making it easy for users to get the gist. The system can handle summarizing new articles as they come in, either on the fly or via scheduled batches. This adds significant value by turning raw news into digestible snippets.

**Phase 7: Personalization Logic (“Why This Affects You”)**

*Goal:* Implement personalized context for news articles, i.e., a brief explanation of why each article is relevant to the individual user. This feature leverages the user’s profile (interests and location) to increase engagement by highlighting personal connections to the news.

• **Define Personalization Criteria:**

• Use the data from the user’s profile (interests array, location city/country) to determine relevance:

• **Interest Match:** If an article’s category matches one of the user’s interests, that’s a key connection. For example, user interest “Technology” and article category “Technology”.

• **Keyword Match:** Beyond category, check if any specific keywords the user cares about appear in the article. If during onboarding you allowed custom topics (e.g., “AI”, “Climate Change”), search the article title/summary for those terms.

• **Location Match:** If the article is about the user’s region. There are a few ways:

• If article has a country field (source country or dateline), and it matches the user’s country.

• If the article title or summary contains the user’s city or state name.

• If the source is a known local newspaper for that city (this requires a mapping of sources to locations, probably too detailed for now).

• Simpler: focus on country-level. If user is in “United States” and article is US-centric (or source is US), consider it relevant. If user city is known and mentioned, even better.

• **General relevance:** Some articles might not directly match interest or location but could be universally relevant (like a pandemic news piece). For those, a generic “major news story” tag might be used, but since the feature is “why *you*”, maybe skip if no direct connection.

• Plan to combine criteria if multiple matches:

• e.g., Article is about a Tech event in the user’s city => interest + location match. The explanation could mention both.

• But keep it short (one sentence preferably).

• If more than one interest matches, just pick one to mention (the most relevant).

• **Implement Personalization Function:**

• In the frontend (likely) or backend, write a function that given an article and user profile returns a string for “why it affects you”:

function getWhyItMatters(article, profile) {

let reasons = [];

*// Interest*

if (profile.interests) {

for (let interest of profile.interests) {

if (article.category && article.category.toLowerCase() === interest.toLowerCase()) {

reasons.push(`related to your interest in \*\*${interest}\*\*`);

break; *// one interest is enough*

} else if (article.summary?.toLowerCase().includes(interest.toLowerCase())) {

reasons.push(`about \*\*${interest}\*\*, a topic you follow`);

break;

}

}

}

*// Location*

if (profile.location\_country && article.country) {

if (article.country.toLowerCase() === profile.location\_country.toLowerCase()) {

reasons.push(`happening in ${profile.location\_country}, where you live`);

}

}

if (profile.location\_city) {

if (article.title?.includes(profile.location\_city) || article.summary?.includes(profile.location\_city)) {

reasons.push(`happening near you in \*\*${profile.location\_city}\*\*`);

}

}

if (reasons.length === 0) return "";

*// Construct phrase*

let reasonStr = reasons.join(" and ");

return `Why this matters to you: This story is ${reasonStr}.`;

}

• The exact phrasing can be adjusted to be more natural. Using Markdown (\*\*) to highlight interest name if displayed in a component that renders Markdown or just as part of text.

• Ensure grammar: if we end up with something like “related to your interest in Tech and happening in United States, where you live.” it might be okay. If grammar is odd, refine by ordering (maybe always put location last).

• If multiple reasons, ensure the conjunction “and” is appropriate. If more than two reasons (unlikely, we limit to interest + location).

• If no reasons, return an empty string or null, meaning we won’t display the “why” line for that article.

• **Integration into Feed:**

• In the frontend feed component (the list of articles):

• When mapping articles to display, for each article, call getWhyItMatters(article, userProfile).

• If a non-empty string is returned, render it in a small font or italic below the summary.

• E.g., <p className="text-sm text-gray-600 italic">{whyText}</p> in Tailwind classes.

• If empty, omit that element entirely.

• This ensures each user sees possibly different messages for the same article depending on their profile.

• Alternatively, do this logic server-side (like when assembling a personalized feed). But doing it on the client is fine since it’s not sensitive and uses already loaded data.

• **AI-driven Enhancement (Optional):**

• (For future or optional complexity) Use an LLM to generate a more nuanced explanation:

• Provide the model with the article summary and user profile, ask: *“Explain in one sentence why a person interested in [X] and living in [Y] might care about this news.”*

• This could yield insightful connections, but it’s expensive to do for every article for every user. So likely not feasible in real-time for a feed.

• It could be triggered on-demand (e.g., user clicks a “Why?” button to generate a detailed reason), but that’s beyond MVP scope.

• Therefore, stick to rule-based as implemented.

• **Testing Personalization Output:**

• Simulate a user profile and various articles:

• User: interests [“Business”,“Sports”], location\_country “United States”, location\_city “New York”.

• Article 1: category “Sports”, country “United States” -> Expect reason mentions interest and location (”…related to your interest in Sports and happening in United States, where you live.”)

• Article 2: category “Technology”, summary mentions “New York” -> user interest not match, location city match -> “…happening near you in New York.”

• Article 3: category “Politics”, no location tie, user not interested in Politics -> no output.

• Fine tune as needed.

• Check UI layout with these strings. Make sure they don’t break the card layout or overflow oddly. Use CSS to truncate if too long or wrap as needed.

• **Performance Consideration:**

• The function is light, iterating over interests and doing string contains. Even with, say, 10 interests and a summary of few hundred characters, it’s negligible in cost. Running it on the client for, say, 20 articles is fine.

• If you had thousands of articles visible (which we likely page anyway), it’s still okay.

• No need for caching here since it’s quick, but be aware it runs each render. Could memoize results if needed.

• **Adaptation for Future:**

• If we introduce a more complex recommendation algorithm, it might rank articles by relevance score to user. That’s different from explaining why, but could feed into what we choose to show.

• For now, assume feed is all latest news with filter by interest, so explanation is straightforward.

• *Deliverable:* **User-Specific Contextual Messages** – The application now displays a short “Why this matters to you” note for relevant news articles in a user’s feed. This note is dynamically generated based on the user’s interests and location, helping users understand the personal relevance of a story at a glance. This personalization adds a human touch to the feed, increasing the likelihood that users engage with the content that’s especially pertinent to them.

**Phase 8: Categorization and Storage in Supabase with De-duplication**

*Goal:* Refine the data storage to ensure efficient retrieval and no duplication of content. This phase revisits the Supabase database schema and data to improve categorization (tagging articles with topics) and solidify de-duplication so that users do not see repeated news items.

• **Finalize Article Categorization:**

• Ensure that each article in the articles table has an assigned category or topic tags:

• We likely already assign a primary category during fetch (Phase 5). Verify consistency: e.g., use a controlled vocabulary for categories. It might be worth having a reference table or enum for categories.

• For instance, define a set: {World, Business, Technology, Sports, Entertainment, Science, Health, Politics, Other}.

• Standardize category naming in the fetch logic mapping. If an article doesn’t fit these (some APIs have odd categories), map it to the closest or “Other”.

• Optionally, store multiple categories or keywords:

• Could add a tags: text[] column for any keywords (like “AI”, “COVID-19”) extracted. This could be from API or even use AI to categorize topics. But that’s more advanced (like entity recognition).

• For now, one category is fine; possibly extend later if needed.

• If using category filter for users (i.e., user interests correspond to these categories), ensure they align exactly. E.g., if user interest is “Sci/Tech” but article category is “Technology”, ensure you’re matching the same terminology.

• Add database **indexes** on category and on published\_at:

• This will speed up queries when we select latest news per category or interest.

• In Supabase (SQL), for example:

CREATE INDEX idx\_articles\_category ON articles(category);

CREATE INDEX idx\_articles\_pubdate ON articles(published\_at DESC);

• These could be done via the SQL editor or migration scripts.

• **De-duplication Process Reinforcement:**

• Evaluate how well the deduplication has been working in practice:

• Look at the articles table after a few fetch runs. If you see any obvious duplicates (e.g., same story with slightly different title), refine logic.

• Potential improvements:

• Instead of just URL, consider using the article title normalized as a key as well:

• Create a generated column in Postgres for normalized title (e.g., lowercase, remove spaces/punctuation) and index that.

• But two different news on same topic might accidentally match if too short, so be careful.

• Maybe add a hash column:

• Compute a hash of the title and perhaps source or date, store it. Use that for conflict checking.

• This could be done in code or using a function in Postgres.

• **Article Clustering:** Implement a simple way to link related articles:

• e.g., if two articles have very similar titles (after removing stop words), assign them a cluster\_id.

• This could just be one takes the id of the other as cluster id, or generate a new uuid for cluster group.

• Not needed for MVP, but if duplicates are an issue, at least mark them. (Alternatively, just not insert duplicates at all, so cluster = unique story).

• The simplest: if title similarity > 90% and same day, skip inserting new one.

• If not already, adjust the fetch script to query existing articles by title or url first:

• For example, when processing new fetch results, for each candidate, do a quick Supabase query select id from articles where url = X or title\_ilike Y.

• However, doing a DB query per article might be slow; better approach is after fetching, filter in memory using a set of known URLs loaded from DB.

• You can load all article URLs from the past day or two into a Set, then filter new ones against it.

• This avoids hitting DB for each check and ensures even if one API gave older news, we catch it if previously stored.

• **Storage for Additional Features:**

• If implementing Save-for-later (Phase 14), prepare a table saved\_articles:

• Columns: user\_id (uuid, references auth.users), article\_id (uuid, references articles.id), saved\_at timestamp.

• Primary key can be composite (user\_id, article\_id) to avoid saving twice.

• RLS: user can insert/select their own saved items.

• This can be created now to avoid altering production DB later.

• If implementing history or read-tracking, consider a user\_article table for read status or impressions, but not necessary at this point.

• **Testing Data Integrity:**

• Perform a fresh fetch with the improved dedup logic:

• Confirm that no duplicate URLs are inserted (the unique constraint or upsert should ensure that, log should show some “conflict skipped” perhaps).

• Intentionally try adding a duplicate (maybe manually insert an article same URL and see if upsert prevents a second).

• Query some data:

• Try a query to get latest 10 articles in “Technology”:

select title, source, published\_at from articles where category='Technology' order by published\_at desc limit 10;

• This should be fast with the index and show the mix of sources, demonstrating the aggregator coverage.

• If any category seems empty, maybe our sources didn’t provide it; consider adjusting sources or add another API for that category.

• Check that all articles have summaries (Phase 6). If some don’t, either summarization hasn’t caught up or failed; ensure summarizer covers them eventually.

• **Backup & Migration:**

• Use Supabase CLI to generate migration for the schema changes (like new columns or indexes) so that the database schema is version-controlled.

• Ensure the production DB (when deployed) gets these changes. The Supabase CLI or dashboard can apply them.

• *Deliverable:* **Clean & Organized Data Layer** – The Supabase database now reliably stores news articles with proper categorization and without duplicates. The system is robust against ingesting the same story multiple times. Data is indexed for fast retrieval, which will support efficient feed queries. This gives us a solid foundation to power the user’s news feed: we have high-quality, unique, summarized articles tagged by topics and ready to be filtered by user preferences.

**Phase 9: Frontend UI/UX for Responsive News Feed**

*Goal:* Develop a polished frontend interface where users can browse their personalized news feed. The UI should be responsive (mobile-first design), visually appealing, and incorporate smooth interactions/animations (using Tailwind CSS for styling and Framer Motion for animations).

• **Design the News Feed Layout:**

• Use a mobile-first approach for layout:

• On small screens: likely a single column list of news cards that users can scroll through.

• On larger screens: can use a two-column or grid layout (e.g., main feed in center, and maybe a sidebar for additional info like trending topics or saved items).

• Implement a responsive container, e.g., <div className="max-w-3xl mx-auto"> to limit width on large screens and center the feed.

• Each **News Card** component should display:

• Headline (original or AI-generated) and source (e.g., “BBC News - 2 hours ago”).

• The AI summary text.

• The personalized “Why it matters” note (if provided by Phase 7 logic).

• Possibly a thumbnail image if available (small and left of summary or as a background).

• Actions like “Save for later” (bookmark icon) and maybe “Open full article” link.

• Use Tailwind utility classes for styling:

• Card container: e.g., bg-white dark:bg-gray-800 shadow rounded p-4 m-2.

• Headline: text-lg font-semibold.

• Summary: text-sm text-gray-700 mt-2.

• Why it matters: text-xs text-teal-600 mt-1 italic (choose a color to distinguish).

• Source/time: text-xs text-gray-500 maybe at the top or bottom of card.

• Save button: could be an icon (Tailwind icons or Heroicons, which can be installed via @heroicons/react).

• Ensure the design looks good in dark mode too (maybe later).

• **Incorporate Framer Motion Animations:**

• Animate the appearance of news cards in the feed:

• Use <AnimatePresence> and <motion.div> for the list of articles.

• Example: When new items are added to the list (or on initial load), fade them in and slide up:

<motion.div initial={{ opacity: 0, y: 10 }} animate={{ opacity: 1, y: 0 }} transition={{ duration: 0.3 }}>

...card content...

</motion.div>

• When removing items (if we implement filtering or refresh) use exit animation.

• Add interactive animations:

• Hover effects: maybe scale up the card slightly or raise shadow on hover using Framer or just CSS.

• If clickable elements (like save button), give a tap animation (Framer’s whileTap to shrink icon a bit).

• Possibly use Framer Motion for collapsible sections (like if showing full article text on expand later).

• Use Framer for any page transitions too (e.g., going from onboarding to feed could have a fade transition).

• **Feed Data Integration:**

• Connect the frontend to the data:

• Decide how to get articles for the feed. Options:

1. **Direct from Supabase**: Use Supabase JS client to query the articles table. e.g., supabase.from('articles').select().order('published\_at', { ascending: false }).limit(20).

• Apply filters: for personalization, perhaps filter by category in user interests or by country = user’s country for local news.

• Could also use Supabase’s full-text search or similarity if doing more advanced stuff.

• Ensure RLS on articles allows read to all (since news is not user-private data, you can disable RLS or allow all select).

2. **Via backend API**: Alternatively, have an Express route that assembles the personalized feed server-side (especially if doing complex logic or combining multiple queries). That endpoint could take the user id and return the filtered list.

• For simplicity, using Supabase directly from frontend is fine, since it’s public news data.

• Implement a feed fetch on component mount:

const { data: articles, error } = await supabase.from('articles').select('\*').in('category', userProfile.interests).order('published\_at', { descending: true }).limit(50);

• This fetches recent articles in categories the user cares about.

• We may also want to include some top news outside their interests to avoid blind spots (maybe always fetch some ‘World’ category news). Could union queries or just fetch interests + some top headlines category.

• If including location, maybe also filter where country = user.country for local section. Or do a separate query for local news.

• Manage the state: store the fetched articles in a React state articles, setArticles.

• Handle loading and error states: show a loading spinner or “Loading…” text while fetch occurs, and an error message if fetch fails.

• Possibly use Supabase’s real-time subscription on the articles table to get live updates (optional):

supabase.channel('news').on('postgres\_changes', { event: 'INSERT', schema: 'public', table: 'articles' }, payload => {

setArticles(prev => [payload.new, ...prev]);

}).subscribe();

This would allow new articles to appear in feed automatically if the user keeps the page open. Cool, but not essential.

• **Pagination or Infinite Scroll:**

• If the feed could be long, implement lazy loading:

• Use a “Load more” button at bottom, or infinite scroll (e.g., use IntersectionObserver to detect scroll bottom, then load older articles).

• Supabase queries can use .range() for pagination (or use the last timestamp of current list to fetch next set).

• For MVP, maybe a simple “Load more” that fetches the next 50.

• Ensure the UI performance remains good by not rendering extremely large lists without virtualization (if needed, consider a library like react-window for very long lists, though maybe unnecessary).

• **UI Polish:**

• Create a header or navbar for the app:

• Contains the app name (Newspaper.AI logo or text), maybe a greeting to user, and a logout button.

• Use a fixed top nav that collapses on mobile (hamburger menu if needed for options like Settings).

• Possibly a toggle to switch interests filter on/off or switch to “All news” view.

• At the bottom or top of feed, indicate last updated time, and a refresh button to manually fetch latest.

• Ensure “Save for later” UI (if implemented in Phase 14) has a visible icon on each card.

• Use consistent iconography (Heroicons or FontAwesome).

• If an article card is clicked (maybe to read more), decide behavior:

• Could open the original article in a new tab (target=\_blank).

• Or show a modal with the full summary and link (maybe with more AI context if we generate).

• Initially, maybe a simple external link on the headline.

• Add a footer with a small note or links (like “About”, “Privacy”, etc., even if placeholders).

• **Responsive Testing:**

• Use browser dev tools to simulate different screen sizes:

• iPhone 5/SE (very small) to ensure cards shrink properly, text wraps, no overflowing content.

• iPad or Tablet size.

• Desktop full HD and ultra-wide monitors (the max-w container should prevent line length from being too long).

• Check that multi-column layout on desktop (if implemented) looks good. Possibly one column for feed, one column for saved or trending.

• Check that animations still run smoothly on mobile (Framer Motion should be fine).

• **Accessibility & UX:**

• Ensure proper semantic HTML: use <article> tag for each news item, headings for titles, etc., to improve screen reader compatibility.

• Add alt text for images (if showing article thumbnail, use article title as alt, for example).

• Keyboard navigation: can tab through cards and activate links.

• Make sure color contrast is sufficient (Tailwind’s default colors are generally okay, but double-check things like teal-600 on white etc.).

• *Deliverable:* **Interactive & Responsive UI** – The Newspaper.AI web application now presents a dynamic news feed to the user. It’s visually appealing and easy to navigate on any device. News articles are displayed as cards with summaries and personal context. Animations enhance the experience without overwhelming it (e.g., smooth transitions as content loads). The feed automatically reflects the user’s interests and provides a modern app-like feel in the browser. Users can scroll and read summaries effortlessly, with the groundwork laid for further interactions (like saving articles or deeper dives).

**Phase 10: Test Driven Development (TDD) and Quality Assurance**

*Goal:* Introduce a comprehensive testing strategy to ensure the application’s reliability, correctness, and maintainability. This includes setting up unit tests for logic (especially the personalization and summarization functions), component tests for React UI, and end-to-end tests for the overall user flows.

• **Set Up Testing Frameworks:**

• **Unit/Integration Testing:** Use **Vitest** (a Vite-native test runner similar to Jest) for unit tests.

• Install Vitest and related libraries: npm install -D vitest @testing-library/react @testing-library/jest-dom jsdom.

• Configure Vitest in vite.config.ts (if using Vite) or package.json:

test: {

globals: true,

environment: 'jsdom',

setupFiles: './src/setupTests.ts', *// if needed for any global setup (like extending expect)*

}

• Create src/setupTests.ts and import @testing-library/jest-dom to have custom matchers.

• **Component Testing:** Write tests for React components using **React Testing Library** (which we installed above).

• Example: test the NewsCard component:

• Render it with sample article data and profile context, then assert that the summary and “Why it matters” text appear when appropriate.

• Use jest-dom matchers: expect(screen.getByText(/Why this matters/)).toBeInTheDocument().

• Test conditional rendering: e.g., if no reason text, it should not render that element.

• **Backend Testing:** If any complex logic in backend (like the fetch or summarizer functions), use Vitest to test those too:

• For instance, test that deduplicateArticles(list) returns unique items when given duplicates.

• Mock external API calls for summarization to not actually call OpenAI in tests (Vitest can mock modules).

• Could also test the personalization function getWhyItMatters() with various scenarios.

• **E2E Testing:** Use **Playwright** for end-to-end browser testing:

• Install Playwright: npm install -D playwright.

• Initiate it (may auto-download browsers).

• Write tests that simulate user behavior:

• Example: “User can sign up and see a feed” – launch a browser, go to the app, fill the signup form, go through onboarding (select interests, skip location), then check that the feed page shows articles.

• Use selectors or text to verify something expected (like the interest they chose appears in a “Why it matters” note).

• Another E2E: “User can log in and save an article” (if implemented).

• These tests might require seeding the database with some known articles and perhaps using a test Supabase project or local Supabase.

• You can set up environment variables for test (point to a separate Supabase or a local one via Supabase CLI).

• Or better, mock network requests in the frontend for tests to avoid hitting real DB (could intercept supabase calls, but that’s complex).

• Possibly, treat E2E more as integration with real staging environment.

• **Test Data and Mocks:**

• Create sample JSON files for articles and profiles to use in tests (to avoid hitting actual APIs).

• For summarization, mock OpenAI responses using Vitest’s mocking capabilities or dependency injection.

• Ensure no secrets are required in unit tests (maybe set dummy env keys).

• **Continuous Integration:**

• Update the CI workflow to run npm run test (Vitest tests) and possibly run Playwright in CI (Playwright has GitHub action integration).

• Running E2E in CI can be more involved (need to deploy the app or use a test server). Possibly skip E2E in CI for now or use a dummy environment.

• **TDD Approach Going Forward:**

• Encourage writing tests for new features or bug fixes. For instance, if a bug is found in personalization logic, first write a test that reproduces it, then fix the code.

• Cover critical functions:

• ensure that deduplication logic doesn’t remove unique articles (test with a list of unique inputs).

• ensure that summarization output is trimmed to desired length (maybe test our prompt handling function).

• verify that the interest filtering only shows categories user selected (simulate a user with specific interests and see feed filtering).

• **Testing Interactive Components:**

• Test that clicking “Save” triggers the correct function (maybe mock supabase call).

• Test that the login form shows errors when given wrong input (simulate typing in form fields and clicking submit).

• Use Testing Library’s fireEvent or userEvent to simulate events.

• **Bug Fix Verification:**

• If any manual testing revealed issues (e.g., location not saved if user does X, or feed not updating when interest changed), write tests to cover those flows after fixing.

• *Deliverable:* **Robust Test Suite** – The project now includes a suite of automated tests:

• Unit tests that validate business logic (deduplication, personalization, small functions).

• Component tests that ensure the UI renders expected elements given different states.

• End-to-end tests covering key user journeys (sign-up, onboarding, seeing personalized feed).

• The tests run in CI and help prevent regressions. This dramatically improves confidence in the stability of Newspaper.AI as it grows, and aids future refactoring or scaling efforts.

**Phase 11: Deployment Configuration (Render, Cron Jobs, Supabase)**

*Goal:* Prepare the application for deployment to a production environment. This involves configuring hosting for the frontend and backend (using Render.com as specified), setting up scheduled jobs for the news fetcher and summarizer, and ensuring all environment variables and database connections are correctly configured in production.

• **Render Account & Services Setup:**

• Log in to **Render.com** (or create account) and connect the Git repository.

• Set up a **Static Site or Web Service** for the frontend:

• If using Vite, you’ll have a build command npm run build which outputs to dist (static files).

• On Render, create a new Web Service (for static site, they actually have a static site service which just serves the files).

• If static site, specify the build command and publish directory (dist/). If a regular web service (like if using Next.js SSR, not the case here), you’d run a server.

• Set environment variables required for frontend (in Render, for static site, you might need to embed env at build time). Actually, since it’s static, the VITE\_SUPABASE\_URL and VITE\_SUPABASE\_ANON\_KEY should be provided during build.

• On Render, under Settings -> Environment, add VITE\_SUPABASE\_URL and VITE\_SUPABASE\_ANON\_KEY. They will be replaced in the build output if used in code.

• Domain: you can use the default <app-name>.onrender.com or set up a custom domain if you have one.

• Set up a **Background Worker or Cron Jobs** for backend tasks:

• Render has a Cron Job feature to run tasks on schedule without a continuously running server.

• For our use case, we might not even need a persistent backend server running 24/7, if all heavy lifting can be done via scheduled jobs and the frontend directly uses Supabase.

• However, consider if any server endpoints are needed (if not, we can skip deploying an Express server).

• If skipping continuous backend, use two Render Cron Jobs:

1. **fetchNews Cron:** runs node fetchNews.js on schedule (e.g., every hour).

2. **summarizeNews Cron:** runs node summarizeNews.js on schedule (e.g., every 15 minutes).

• To do this, you might need to push the code to Render via a **private service** or have a Docker container that can run these scripts.

• Render Cron Jobs allow specifying a repo and command. Possibly you create it similar to a service but choose “Cron Job” and give the schedule and run command.

• Ensure in the Cron job’s environment, you set SUPABASE\_URL and SUPABASE\_SERVICE\_KEY, plus all news API keys and OpenAI key so the script can use them.

• If using a persistent backend server:

• Deploy it as a **Web Service** (or Background Worker if it doesn’t serve HTTP requests).

• For example, run the Express server (which might have an endpoint or just serve as a container to run cron via node-cron internally).

• Could use node-cron package inside the server to schedule fetch and summarize tasks in-process.

• But Render’s native Cron is simpler and decoupled.

• For clarity and resource efficiency, using Render Cron Jobs is ideal:

• Create one Cron Job for each script with appropriate cron expression.

• E.g., fetch job: 0 \* \* \* \* (top of every hour), summary job: \*/15 \* \* \* \* (every 15 minutes).

• Watch out: if summary job overlaps with fetch job, could cause contention. But summary is quick vs fetch maybe longer. Possibly shift them so fetch at 0:00 and summary at 0:05 etc.

• **Supabase Configuration for Production:**

• The Supabase project created earlier is already remote and cloud-hosted. Ensure your Render services have the correct Supabase URL and keys.

• You might want to use the service role key only in the Cron jobs or backend (never expose it to frontend).

• The anon key goes to frontend.

• If using any Supabase Edge Functions or Supabase cron, not needed since Render handles logic.

• Monitor Supabase usage limits to ensure free tier can handle the read/write (likely fine initially).

• **Environment Variables on Render:**

• Set all needed env vars for each service:

• For the Frontend: only public ones (Supabase anon, perhaps a flag or API URL if any).

• For Cron jobs: Supabase URL, service key, news API keys, OpenAI key, Radar key (if used server-side), etc. Keep these secret.

• Render has a nice interface to add those. Double-check names match what code expects (process.env.X).

• **Deployment Process:**

• Trigger a deploy (manually or via git push if auto-deploy enabled on commit).

• Ensure the build passes. If any errors, fix (especially things like missing env or build commands).

• Once deployed, test the frontend in the Render-provided URL:

• Go through sign up, onboarding. It should communicate with Supabase (make sure the domain is allowed in Supabase’s settings if using Auth email links).

• Check Supabase Auth redirect: If email confirmations are on, set the SITE URL in Supabase Auth settings to your deployed frontend URL for deep linking.

• After onboarding, see if feed loads. Likely, initially no data if the fetch cron hasn’t run yet. Might need to run fetch manually or wait.

• You can manually invoke the fetch script by triggering the Cron job on Render (there might be an option to run now) or run it locally and push data to Supabase.

• Once some articles are in DB, ensure they appear on frontend.

• Test that summarization ran (summaries present).

• **Scheduled Job Verification:**

• After scheduling, monitor that Cron jobs execute:

• Render provides logs for each run. Check logs to see if any errors, and that “inserted X articles” or “summarized Y articles” messages appear.

• Check Supabase to confirm new rows are added over time.

• If issues, debug (maybe env not set properly or script crash due to missing dependency).

• **Supabase and Render Communication:**

• Supabase is external to Render. Ensure no network issues (Render should be able to reach any external internet by default).

• If Supabase restricts API calls by IP, that could be an issue (it doesn’t by default).

• If needed, you could deploy an Edge Function on Supabase for heavy summarization, but we already have our pipeline.

• **Domain and SSL:**

• If using a custom domain for frontend, configure it in Render and update DNS. Ensure it has SSL (Render handles LetsEncrypt automatically).

• If frontend is at custom domain, update Supabase redirect URI for auth (in Auth settings).

• **Deployment Checklist:**

• Turn off debug logs or dev-only features for production (like verbose console logs or any test routes).

• Ensure no hard-coded localhost URLs remain.

• Build is in production mode (Vite does this automatically for npm run build).

• Possibly set NODE\_ENV=production in Render for clarity.

• *Deliverable:* **Production Deployment** – Newspaper.AI is now deployed and accessible to users. The frontend is live on a public URL, and the backend processes (news aggregation and summarization) run automatically on schedule via Render’s cron jobs. All secrets and configurations are properly set in the environment, and the app connects to the production Supabase database. We have validated that a new user can sign up, receive updated news, and the system is functioning end-to-end in a live setting.

**Phase 12: Logging and Monitoring (Sentry, Analytics)**

*Goal:* Implement error logging and user behavior monitoring to maintain the health of the application and gain insights into usage. Sentry will capture runtime errors, and an analytics tool like PostHog or Plausible will track user interactions and page views in a privacy-compliant way.

• **Sentry Integration (Error Tracking):**

• Create a Sentry project for both frontend and (if applicable) backend.

• Add Sentry SDK:

• Frontend: npm install @sentry/react @sentry/tracing.

• Initialize Sentry at app startup (e.g., in index.tsx or App.tsx):

Sentry.init({

dsn: "https://<key>@sentry.io/<project>",

integrations: [new Sentry.BrowserTracing()],

tracesSampleRate: 0.1, *// adjust for performance monitoring sample*

environment: import.meta.env.MODE

});

• This will automatically capture exceptions and unhandled rejections. It can also track performance (route changes, component render times) if needed.

• Make sure to set a low sample rate if not on a paid plan to avoid hitting event quotas.

• Also configure Sentry to scrub PII if any (though our app doesn’t handle sensitive personal data except email, which Sentry auto scrubs by default).

• Backend: If we have a Node process, npm install @sentry/node. Initialize similarly with your DSN in the Express app or before running cron logic:

Sentry.init({ dsn: "...", environment: process.env.NODE\_ENV });

*// If Express, use Sentry.Handlers.requestHandler and errorHandler middlewares.*

• For cron scripts, could wrap main logic in try/catch and Sentry.captureException on error.

• Or simply import Sentry and init, then if any uncaught exception occurs, Sentry will log it (if the process doesn’t exit too fast).

• Test Sentry in dev by forcing an error and verifying it shows up in Sentry dashboard.

• Sentry will help catch issues like frontend runtime errors or if a cron job throws (perhaps Out of memory or an API error not caught).

• Document in the team how to check Sentry and respond to alerts.

• **Analytics Setup (PostHog or Plausible):**

• Decide on analytics platform:

• **Plausible Analytics** is simpler (pageview-focused, lightweight script, privacy-friendly).

• **PostHog** is more comprehensive (events, funnels, open-source but heavier to self-host).

• For MVP, Plausible might suffice to get page views and referrers.

• Plausible setup:

• Create a site on plausible.io for your domain.

• They give a script include; add it to index.html or as a React effect.

• It automatically tracks page views. For custom events (like article save or share), you can use window.plausible("CustomEventName").

• Ensure not to include on local dev or provide an opt-out for users if necessary (Plausible is GDPR compliant, no cookies by default).

• PostHog setup (if chosen):

• If using a hosted PostHog or a self-hosted instance, include their snippet or use their JS package.

• PostHog can track user identity (we might not want to send email or such; could send user ID or just treat everyone as anonymous).

• Track events like “Article Viewed”, “Article Saved”, “Signup Completed”, etc., to analyze feature usage.

• Use feature flags if needed (PostHog has that, but not needed now).

• **Log Aggregation (Server logs):**

• Render provides logs in their dashboard, but for long-term log storage or querying, consider plugging into a service:

• Could integrate with LogDNA, Papertrail, or use an open-source like ELK stack (overkill now).

• Not mandatory at MVP scale, but as scaling, having centralized logs from cron jobs and any backend could help debugging.

• Perhaps ensure the cron job outputs are saved (Render might keep last 30 days).

• **Monitoring Cron Health:**

• If possible, set up an alert if cron jobs fail or haven’t run (Render might have some status, or you could have Sentry alert on missed schedule).

• Alternatively, implement a simple health check: e.g., create a table heartbeat and have each cron update a timestamp when it runs, then an alert if not updated recently (just an idea).

• For now, rely on Sentry for error and manual check.

• **Performance Monitoring:**

• Sentry’s performance tracing can show slow API calls in frontend. This might reveal if Supabase queries are slow or if a component is re-rendering too much.

• Evaluate this data after deployment to optimize if needed.

• Also monitor Supabase dashboards for query performance and usage.

• **User Feedback Loop:**

• Consider adding a simple feedback form or email link in the app where users can report issues or suggestions. (Not asked, but part of a good production readiness).

• If implemented, tie it to an email or a logging channel.

• *Deliverable:* **Observability in Place** – The deployed application is now equipped with tools to observe and troubleshoot its behavior. Sentry is catching any errors (front or back) and providing stack traces for developers to fix issues proactively. Analytics are recording how users interact with the app (e.g., page views, clicks), giving insight into engagement and feature usage. With logging and monitoring, the team can ensure stability and continue improving Newspaper.AI by responding to real-world usage patterns and errors.

**Phase 13: Translation and Multilingual Support**

*Goal:* Expand the application’s reach by supporting multiple languages. Implement translation of news summaries (and possibly interfaces) so that users can read news in their preferred language. We’ll use an API (OpenAI or DeepL) for translations and handle content delivery in different languages.

• **Identify Target Languages:**

• Decide which languages to support initially (based on potential user base or available news sources). Common ones: Spanish, French, German, Chinese, etc.

• Also consider that some news APIs provide multi-language content already (NewsData.io has articles in many languages). But our summaries are in English because the LLM summarized the English content.

• We can either:

• Translate the summary and “why it matters” text for each article to the user’s preferred language.

• Or attempt to fetch news in the user’s language from the start and summarize that (which might require multi-language summarization).

• Simpler: stick to summarizing in English then translate summary.

• In user profile, add a field preferred\_language (e.g., “en”, “es”, “fr”). On onboarding, perhaps allow user to pick a language (default to browser locale or English).

• Or if not in onboarding, add an option in settings to change language.

• If no profile language, assume English.

• **Translation API Integration:**

• Options:

• **OpenAI**: the GPT models can translate text. E.g., prompt GPT-3.5: “Translate the following English text to Spanish: [text]”. This works but costs similar to generating text (not too high for short summaries).

• **DeepL**: a dedicated translation API with high quality, especially for European languages. It has a free tier up to 500k chars/month (then $20 per million).

• **Google Translate API**: well-known, but costs $20 per million chars and is not free after a small trial.

• **Azure or AWS translate**: also options, but let’s pick DeepL or GPT for ease.

• If using DeepL:

• Sign up for an API key. Note usage limits.

• Use their client or call via fetch: https://api-free.deepl.com/v2/translate?auth\_key=XXX&text=Hello&target\_lang=ES.

• If using GPT:

• Could use the same OpenRouter/OpenAI pipeline: just prompt for translation. But that’s an extra API call per article per language.

• Actually, maybe better: if user’s language != English, when generating the summary in Phase 6, directly prompt the LLM to output in the target language.

• This would save a separate translation step. For example: “Summarize the following article in Spanish.” and done.

• Could do that if we know the user language at summarization time. But summarization is currently not user-specific (we summarize once and serve to all).

• So likely, summarization is always in English (as a base), then translate on demand.

• **Architecture for Multi-language Delivery:**

• Approach 1: **On-the-fly translation** – When a user is reading the feed, if their language is not English, call translation API for each summary as needed.

• This could introduce latency for each article displayed.

• We can mitigate by translating in batch or caching results.

• Perhaps translate all visible articles when feed loads, then render.

• Implement a cache: maybe store translations in the database. e.g., have columns summary\_es, summary\_fr, etc., or a separate table keyed by article and language.

• Or use a client-side cache (if user navigates away and back, etc.).

• Storing in DB ensures we translate each summary at most once per language, which is efficient but increases storage.

• Approach 2: **Pre-translate popular languages in cron job** – e.g., after summarization, automatically translate the summary to a few languages and store them.

• Could choose top 3 languages we support and always translate every new summary to those.

• That ensures immediate availability, but could waste API quota on translations for languages some articles may never need (depending on user distribution).

• Perhaps a hybrid: pretranslate the ones we expect (maybe based on user profiles count per language).

• For MVP, do on-the-fly with caching in DB:

• Add a table translations with columns: article\_id, language, translated\_summary.

• Or expand articles table to have JSONB or separate text columns per language (not very scalable if many languages).

• Table is more flexible:

CREATE TABLE translations (

article\_id uuid references articles(id),

language text,

summary text,

primary key (article\_id, language)

);

• When a user requests feed:

• For each article, if user.lang != ‘en’:

• Query translations for that article & lang.

• If found, use it.

• If not, call translation API (or OpenAI) to get it, then store it, and use it.

• Could incorporate this logic into a Supabase Edge Function for efficiency (so the client just calls “getFeed” and the function returns translated summaries).

• But doing it in client is okay if careful with concurrency.

• Make sure to also translate “Why it matters” text or generate it in target language. Those strings are short; can also feed to translator.

• The UI chrome (like button labels, headings) can be handled via i18n library:

• Use something like i18next or react-intl for interface translations if needed. For now, our UI is minimal text (“Save”, “Logout”, “Why this matters to you:”), which we can hardcode translations easily if supporting a few languages.

• Possibly just conditionally render Spanish text if lang == ‘es’, etc. Simple approach for MVP.

• **Implementing Translation Calls:**

• In the client or an API route, call the chosen API:

• With DeepL: craft URL or use their SDK. Make sure to handle rate limit (e.g., if translating 20 summaries at once, might be fine).

• With OpenAI: use same openai.createChatCompletion but with a prompt for translation (this may be slower and cost a bit more).

• Leaning towards DeepL for efficiency. Use free tier initially.

• **Testing Multilingual Output:**

• Change your profile language to Spanish. Load feed, confirm summaries appear in Spanish.

• Check that the translation is accurate and not awkward. DeepL is usually good. GPT might produce more natural phrasing but likely similar.

• Test switching language (simulate user change language and reload feed). See that it fetches new translations.

• Ensure fallback: if translation API fails, just show English with a note or try another service (maybe OpenAI as fallback).

• **UI for Language Selection:**

• In settings or profile (maybe a simple dropdown in the navbar or profile page).

• Allow user to pick from list of languages we support. On change, update profile in Supabase and update the app state.

• Immediately re-fetch feed or translations so user sees effect.

• Also possibly allow multi-language news: some might want to see content from multiple languages. That complicates things, likely unnecessary for MVP.

• **Edge: Multi-language news sources:**

• If we wanted, we could fetch news from other languages and summarize in that language. But that basically multiplies our pipeline per language. Instead, translating summaries covers many use cases (someone can read about foreign news in their language).

• Google News already does multi-language aggregation, but our unique angle is summarization + personalization.

• *Deliverable:* **Multilingual Support** – Users can now receive news summaries in their preferred language. The app detects the user’s language setting (from profile) and displays translated summaries and context messages accordingly. This broadens the accessibility of Newspaper.AI to non-English speakers. The implementation ensures translated content is cached to avoid repeated work, and the interface can accommodate multiple languages seamlessly.

**Phase 14: Optional Features (Save-for-later, Notifications, Dashboards)**

*Goal:* Introduce additional features to enhance user engagement and provide extra value. These are optional stretch goals that can be implemented resources permitting. They include saving articles for later reading, push notifications for breaking news, and visual dashboards for news trends or user activity.

• **Save-for-Later (Bookmarking):**

• **Backend:** Use the saved\_articles table (from Phase 8 setup) to store saved items.

• Ensure RLS allows only the user to insert/select their own saved articles.

• **Frontend UI:**

• Add a bookmark icon (e.g., an outline star or bookmark) on each news card.

• Clicking it:

• If not saved yet: call supabase.from('saved\_articles').insert({ user\_id: user.id, article\_id: article.id }). Optimistically update UI (fill icon).

• If already saved (we can fetch saved list to know): clicking again could remove (implement remove if desired, or just allow duplicates prevention via PK).

• Alternatively, treat it as one-way (just save, and manage unsave separately via a profile page).

• Provide feedback on save: maybe a toast “Saved!” or change icon state.

• Create a “Saved” page or section:

• A route /saved that lists the user’s saved articles, possibly using the same NewsCard component.

• Query: join saved\_articles and articles: e.g., supabase.from('saved\_articles').select('article: articles(\*)').eq('user\_id', user.id).

• Display in chronological order of saved\_at or article date.

• Allow unsave here as well.

• **Testing Save:**

• Save an article, refresh page, ensure it’s still marked saved and appears in Saved page.

• Try saving the same article twice (should either error due to PK or be ignored). Our PK stops duplicates.

• **Notifications (Web Push or Email):**

• **Web Push Notifications:**

• Implement using Service Workers for web push:

• Need user permission via Notifications API.

• Need VAPID keys for push service (if not using a service like Firebase).

• Alternatively, integrate with a service like OneSignal or Firebase Cloud Messaging for easier handling.

• Use-case: Notify users of a major breaking news or a daily digest.

• Possibly set up a Render background task or Supabase function that triggers push to all users (who opted in) if a big event occurs. But detecting “big event” is another challenge (maybe if many articles about same topic? or just daily at 8am).

• For MVP, might not implement push fully due to complexity. Instead, maybe email notifications:

• Supabase can trigger emails via its triggers or use external like Mailgun.

• But that requires email content (maybe send a daily summary email with top 5 news).

• If implementing web push minimally:

• Frontend: ask for notification permission after onboarding.

• Use the Notifications API to show a notification. (Without a server to push, we can simulate by client scheduling a daily check or something).

• For actual push, need server component – possibly too heavy for now.

• **In-app Notifications/Indicators:**

• Simpler: a bell icon in UI showing number of new articles since last visit.

• Use local storage to store last read timestamp. Next visit, highlight articles newer than that or show a badge.

• Mark articles as read when clicked. Could have user\_article table to track read status if needed.

• **Visual Dashboards:**

• Ideas: show charts of topics the user read the most, or trending categories globally.

• Could use a chart library like Chart.js or D3 to display:

• A pie chart of user’s interests vs what they read.

• A line graph of number of articles per day (their activity).

• Or a tag cloud of popular keywords in saved articles.

• These could go in a “Insights” or profile page.

• Data sources:

• Use Supabase to aggregate data (SQL queries grouping by category count, etc.) or fetch in client and use JS to count.

• E.g., count articles by category in user’s saved or read list, then display.

• Not core to functionality but nice for engagement, letting user reflect on their news diet.

• Also, an admin or general trends dashboard:

• Perhaps a page showing top 5 trending topics today (e.g., based on frequency of keywords in fetched articles).

• Could do this by scanning article titles or an external trends API.

• Display on a sidebar or a “Trending” section in feed.

• **Performance and Polishing:**

• Ensure these features don’t slow down main use:

• Lazy load dashboards or heavy data after main content.

• For push, ensure it doesn’t annoy user; provide easy opt-out.

• Use local notifications for daily reminder if push not implemented:

• Some browsers support scheduled notifications via service worker or Page Visibility API to detect returning.

• For each optional feature, if time is short, implement partially or as stub for future.

• **Deployment Consideration for New Features:**

• If using service workers (for PWA or push), need to configure and serve correct files. React can register a serviceWorker if using CRA or you do manually with Workbox.

• If using email, need to set up email service credentials.

• Ensure to update environment variables and test these flows in production too.

• *Deliverable:* **Enhanced User Features** – Additional capabilities are added to Newspaper.AI:

• Users can save articles to read later and manage their saved list.

• The groundwork for notifications is laid (potentially allowing users to get alerts for new content).

• Insightful dashboards or visualizations provide extra value, such as showing trending news or personal reading habits.

• These features are optional but can significantly increase user retention and interaction when used. Each is implemented in a scalable way, integrated with the existing platform (Supabase and the React frontend) and can be iterated on in the future.