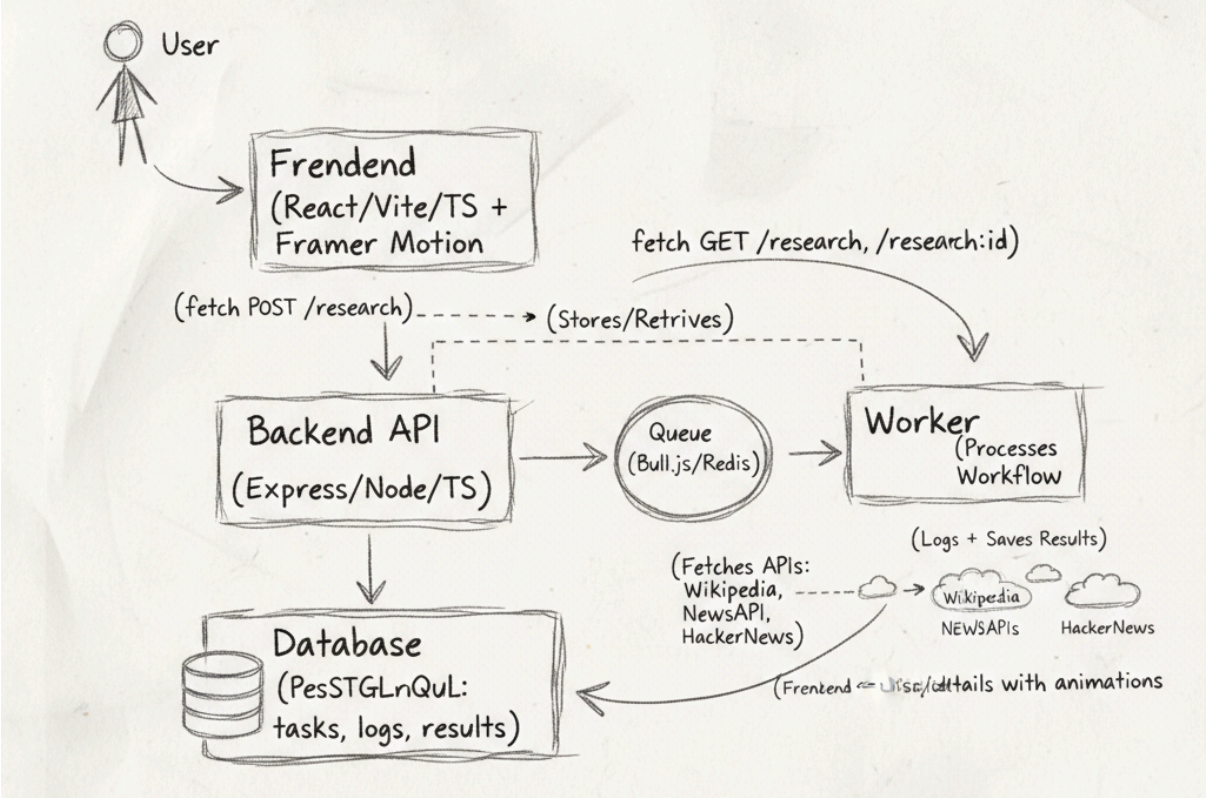
Documentation

Architecture Diagram 🏗️

The system is designed with a client-server architecture that uses an asynchronous, queue-based approach for long-running tasks. This ensures the user interface remains responsive while complex data processing happens in the background.



Explanation:

- **User Interaction (Top-Left)**:
  - The process begins with the `User` interacting with the `Frontend` application. This is typically done through a web browser. The frontend is built using technologies like React, Vite, and TypeScript, and uses Framer Motion for smooth animations.
- **Submitting a Research Request (Frontend to Backend API)**:
  - When the user submits a research topic (e.g., by filling out a form and clicking a button), the `Frontend` sends a `fetch POST /research` request to the `Backend API`. This is the initial trigger for the research workflow.
- **Backend API Processing (API to Queue/Database)**:
  - The `Backend API` (an Express/Node.js/TypeScript application) receives the request.
  - Instead of performing the long-running research directly, the API's primary job is to:

        - Validate the incoming research topic.
        - Create a new "task" record in the `Database` (PostgreSQL), marking its status as 'pending'.
        - Add a new "job" to the `Queue` (managed by Bull.js, which uses Redis). This job contains the details of the research request.
    - By offloading the heavy work to a queue, the `Backend API` can quickly respond to the `Frontend` and remain available for other user requests, preventing timeouts and ensuring a responsive application.

- **Asynchronous Worker Processing (Queue to Worker)**:

    - Separately, a `Worker` process continuously monitors the `Queue` for new jobs. This worker is also built with Node.js/TypeScript.
    - When a new job appears in the `Queue`, the `Worker` picks it up.
    -
- **Data Gathering by Worker (Worker to External APIs)**:
    -
    - The `Worker` then executes the core research logic. It makes requests to various `External APIs` (like Wikipedia, NewsAPI, HackerNews) to gather relevant information based on the research topic. This is where the bulk of the data collection happens.

- **Storing Progress and Results (Worker to Database)**:

    - As the `Worker` progresses through its steps (e.g., fetching articles, summarizing, extracting keywords), it continuously interacts with the `Database` (PostgreSQL).
    - It updates the task's status, logs detailed steps, and eventually saves the final compiled `results` (summaries, keywords, article links) back into the `Database`.

- **Retrieving Status and Results (Frontend to Backend API to Database)**:

    - Meanwhile, the `Frontend` periodically sends `fetch GET /research` (for the list of all tasks) and `fetch GET /research/:id` (for details of a specific task) requests to the `Backend API`.
    - The `Backend API` retrieves the latest task status, logs, and results directly from the `Database` and sends them back to the `Frontend`.

- **Displaying Information to User (Frontend)**:

    - The `Frontend` receives the updated information and `Displays list/details with animations` to the user, reflecting the progress or completion of their research task. When a task is complete, the detailed results are shown.

-

Workflow Explanation ⚙️

When a user submits a research topic, the following sequence of events is triggered:

1. **Input Parsing**: The backend API receives the topic. It uses a validation library like Zod to ensure the input is not empty. A new entry is created in the `tasks` table with a status of `'pending'`, and an initial log is saved. The job is then added to the Bull.js queue.

2. **Data Gathering**: The background worker picks up the job from the Redis queue. It begins gathering data by making parallel requests to multiple external sources using `node-fetch`:

    - **Wikipedia**: Uses the search API to find relevant pages.
    - **NewsAPI**: Searches for top headlines and articles related to the topic.
    - **HackerNews**: Queries for relevant stories and discussions. All results are collected and combined into a single dataset.

3. **Processing**: The worker processes the raw data. It selects the top 5 most relevant articles based on a simple scoring metric (e.g., how closely the title matches the topic). For each article, it performs:

    - **Summarization**: A simple summary is generated by truncating the article content to the first 200 words. (An LLM could be integrated here for higher-quality summaries as a bonus).
    - **Keyword Extraction**: Keywords are identified by calculating word frequency and selecting the top 5 most common (and relevant) words.

4. **Result Persistence**: After each major step (e.g., "Data Gathering," "Processing"), a log entry is inserted into the `logs` table, linked to the task's ID. Once the entire workflow is complete, the final results (a JSON object containing the articles, summaries, and keywords) are saved to the `results` table. Finally, the task's status in the `tasks` table is updated from `'processing'` to `'completed'`.

5. **Return to Frontend**: The frontend polls the `GET /research` endpoint to refresh the task list. When it sees a task is `'completed'`, the "View Details" button becomes active. Clicking it fetches the specific task's data from `GET /research/:id`, which includes all associated logs and the final results from the database.

Notes on Trade-offs 🤔

Every technical decision involves trade-offs. Here are the key choices made for this project and their rationale.

- **Node.js vs. FastAPI**:
    - **Choice**: **Node.js/TypeScript**.
    - **Reasoning**: Using TypeScript on the backend creates consistency with the frontend stack, enabling code sharing and a more unified development experience. Node.js's asynchronous, event-driven nature is well-suited for I/O-heavy operations like making multiple external API calls.
    - **Trade-off**: Python/FastAPI might offer better performance for CPU-intensive tasks and has a more mature ecosystem for data science and AI. However, for this project's scope, the benefits of a full-stack TS environment outweighed the potential performance gains.

- **Vite vs. Next.js**:
    - **Choice**: **Vite**.
    - **Reasoning**: Vite provides an extremely fast development experience and is optimized for building modern single-page applications (SPAs). Since this project doesn't require Server-Side Rendering (SSR) or advanced SEO, Vite's simplicity and speed were ideal.
    - **Trade-off**: **Next.js** offers powerful features like SSR, static site generation, and file-based routing, which are crucial for SEO and perceived performance in larger applications. However, this adds complexity that was unnecessary for this internal-tool-style dashboard.

- **Database Choice (PostgreSQL)**:
    - **Choice**: **PostgreSQL**.
    - **Reasoning**: PostgreSQL is a robust, open-source relational database with strong support for JSONB, which is perfect for storing the unstructured results from the research workflow. It's production-ready and scales well.
    - **Trade-off**: **SQLite** would be simpler for local development (no separate service needed), but it's less suitable for production environments, especially with concurrent writes from the API and worker. PostgreSQL provides a more realistic setup from the start.

- **Queue System (Bull.js/Redis)**:
    - **Choice**: **Bull.js with Redis**.
    - **Reasoning**: The research process can take several seconds. Performing it directly in the API request would block the server and lead to a poor user experience (and potential request timeouts). A queue system decouples the long-running task from the initial request, ensuring the API remains

responsive. Bull.js provides a robust feature set (e.g., retries, progress tracking) on top of Redis.
    - **Trade-off**: The architecture is more complex than a simple synchronous approach. It requires managing two extra services (Redis and the worker). For a very fast workflow, a queue might be overkill, but here it's essential for reliability and scalability.

- **UI Libraries (Framer Motion)**:

    - **Choice**: **Framer Motion**.
    - **Reasoning**: It provides a simple, declarative API for adding fluid animations (e.g., fade-ins for loading states, list item transitions). This significantly improves the user experience by making the interface feel more dynamic and responsive.
    - **Trade-off**: Adding any new dependency increases the final bundle size. For an application where performance is absolutely critical, one might opt for pure CSS animations, but the developer experience and power of Framer Motion justified the slight size increase.

- **Deployment (Render/Firebase)**:

    - **Choice**: **Render** for the backend (API + worker) and **Firebase** for the frontend.
    - **Reasoning**: Both platforms offer generous free tiers that are perfect for portfolio projects. Render's support for Docker and distinct service types (Web Service, Background Worker) is a perfect match for the backend architecture. Firebase Hosting is incredibly simple and fast for deploying static frontends like a Vite build.
    - **Trade-off**: **Vercel** is another excellent choice for frontend hosting and is often considered faster for global delivery. However, Firebase integrates seamlessly with other Google services (like Authentication and Firestore) if the project were to be expanded.