LOGICAL APPROACH

Step 1: Get CVE Data from the API

The NVD CVE API has a ton of security vulnerability data. But instead of grabbing everything in one go, I will **fetch it in chunks** using pagination (like scrolling through pages on a website). I will store this data in a database.

Before saving, I will **clean up the data** (remove duplicates, filter out unnecessary info, and make sure everything is formatted correctly).

Step 2: Keep the Data Fresh

Since new vulnerabilities appear daily, we need a system that **updates the database regularly**. There are two ways to do this:

1. **Full refresh** – Wipe everything and pull all data again (useful occasionally but slow).
2. **Incremental updates** – Just fetch what has changed since our last update (much faster).

I will use **a scheduler** like APScheduler to run updates automatically every few hours/days.

Step 3: Build the Backend (Flask API)

Now that we have a database filled with CVE data, we need to **create APIs** that let users search and filter the data.

Some things the users to do:

- Get details of a specific CVE (e.g., /api/cves/CVE-2023-12345)
- Find CVEs from a particular year (e.g., /api/cves?year=2023)
- Filter CVEs by severity score (e.g., /api/cves?min_score=7&max_score=10)
- Show recently modified CVEs (e.g., /api/cves?last_modified=30)
- Handle pagination so users don't get overloaded with data

We'll use **Flask & Flask-SQLAlchemy** to make this happen.

Step 4: Build the Frontend (UI with JavaScript, HTML, and CSS)

I will build a **user-friendly webpage** to display this data. The first page (/cves/list) will:

- Show CVE details in a **table**
- Have a **"Total Records"** count
- Let users **choose how many records to see per page** (10, 50, or 100)
- Load new data when the user selects a different page

I will use **JavaScript to call our Flask APIs** and update the table dynamically.

Step 5: Click a Row → Go to Details Page

When a user clicks a row in the table, they should be taken to a page like /cves/CVE-2023-12345.
This page will:

- Make an API call to get full details of that CVE
- Display all its information in a well-organized way

Step 6: Security & Best Practices

Finally, I will make sure my code is **secure and well-structured** by

- Validate user inputs (avoid SQL injection & XSS attacks)
- Handle API failures (what if the NVD API is down?)
- Use **.env files** for sensitive info (like database credentials)