Language Choice

I chose **Python** because it is the language I am most comfortable with, and it provides a rich ecosystem of libraries for web development, data processing, and database interaction. This made it a natural fit for the fetch, store, and present challenge. While **Go** is my second backend language, it is better for highly concurrent applications, this project makes only limited API calls, so Python provided the right balance of simplicity and functionality.

Web Framework

I selected **FastAPI** as the framework. Initially, I considered both Flask and FastAPI, as they are widely used for lightweight web applications. I ultimately chose FastAPI because it is modern, actively maintained, and provides built-in interactive documentation via Swagger UI (/docs) and ReDoc (/redoc), which made testing and debugging API endpoints straightforward and more visible.

Server

For serving the application, I used **Uvicorn**, a reliable ASGI server that integrates seamlessly with FastAPI and is considered the standard choice for deploying FastAPI applications.

HTTP Client

I chose **httpx** over requests as the HTTP client for fetching CVE data from the NVD API. While I am currently using it in a synchronous manner and not leveraging its asynchronous capabilities, httpx offers a modern API, robust connection handling, and HTTP/2 support. Choosing it allows the project to remain future-proof: should I later want to fetch data concurrently or implement other async features, I can do so without replacing the HTTP client.

Frontend

The frontend is built with **HTML and CSS** for structure and styling, **JavaScript** for interactivity, and **Jinja2 templating** to dynamically render tables and graphs based on backend data. I considered using a frontend framework like React, but concluded it would be excessive for the scope of this project. The current stack keeps the project lightweight and maintainable.

Database

I used **SQLite**, which stores data in a local file (cves.db) and requires no additional setup. Since the dataset has a consistent tabular structure, a relational database is a natural fit. A NoSQL solution like MongoDB was unnecessary. SQLite scales adequately for this challenge by handling batch inserts efficiently and supporting vertical growth as more rows are processed. For larger datasets or concurrent users, a more robust DBMS could be considered.

Application Flow

The main menu has three options:

1. **Fetch CVEs** – Dynamically retrieves CVE data from the NVD API and streams it in batches to the database and UI.
2. **View Table** – Displays already stored CVEs in a table. This is instantaneous because it queries the local database without contacting the API. The table is static and represents the dataset as it exists at that moment.
3. **Analytics & Graphs** – Displays visualizations of the CVE dataset.

At startup, the application ensures an empty SQLite table exists to hold the CVE data.

```
@asynccontextmanager  1 usage  Jonathan Grinshpan
async def lifespan(app: FastAPI):
    # --- Startup code ---
    DB.create_tables()
    print("Database tables ensured.")
    yield
```

This code runs when the project starts. It's marked **async** so that any asynchronous tasks I might want to add later—like an API call or other startup operations—can run without blocking the app. The yield separates startup from shutdown code.

I check whether an API key is provided. With a key, requests to the NVD API are less likely to be **rate-limited**, making response times more stable (2–10 seconds compared to 2–60 seconds without a key). Before registering for an API key, I was frequently timed out when making requests to the NVD API, which made testing and fetching large datasets unreliable. This experience highlighted the importance of handling API rate limiting.

```
API_KEY = os.getenv("NVD_API_KEY")
headers = {}
if API_KEY:
    print("APIKEY received")
    headers["apiKey"] = API_KEY
else:
    print("No APIKEY given, might time out due to rate limiting.")
```

Fetch CVEs

The backend fetches all CVEs from the NVD API at once.

```
response = await client.get( url: f"{API_URL}?cpeName={CPE}",headers=headers)
```

As it processes the data, it inserts CVEs into the database **in batches of 20** and simultaneously streams each batch to the frontend using Server-Sent Events (SSE). This allows the user to start seeing results progressively while the rest of the data is still being processed.

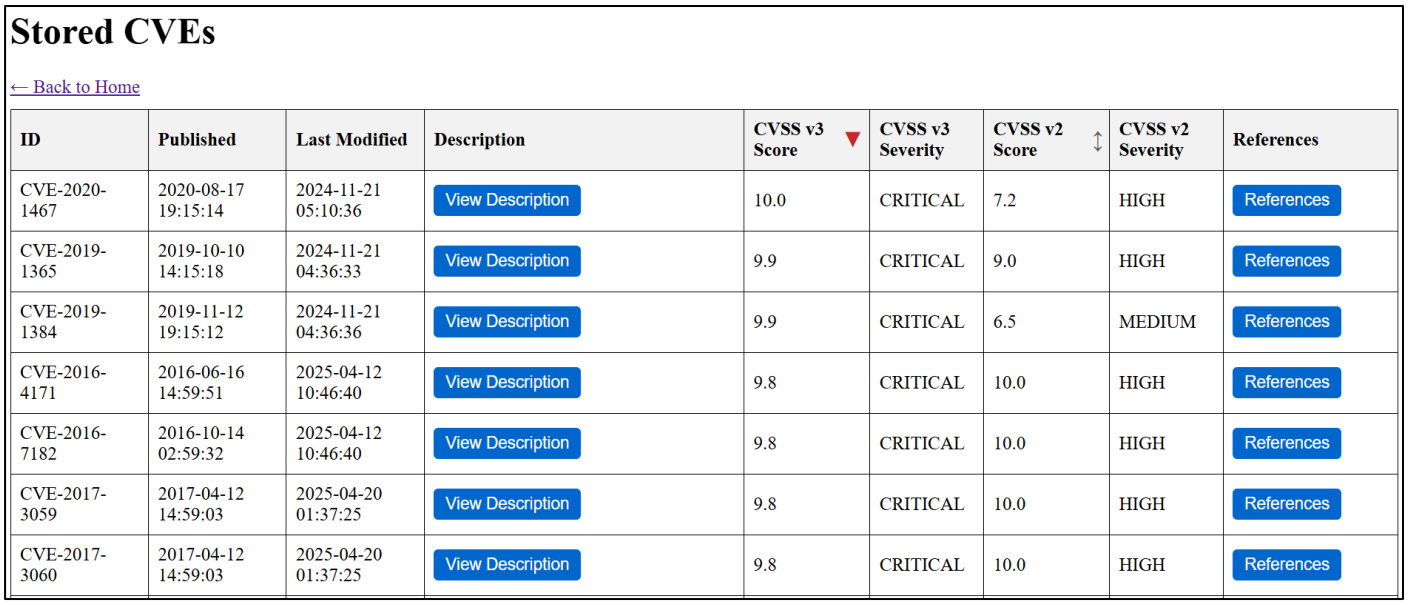
For database safety, I use **parameterized queries** to prevent SQL injection and validate all reference URLs to block unsafe protocols or internal hosts, protecting against **malicious links**.

```
placeholders = ", ".join("?" for _ in COLUMN_DICT)
sql = f"INSERT OR REPLACE INTO cve ({columns}) VALUES ({placeholders})"
```

```
safe_refs = [ref["url"] for ref in refs if is_safe_url(ref["url"])]
```

On the frontend, the table is rendered dynamically. While data loads, users see a **“Loading cves…”** indicator. Once rows arrive, they can:

- Click **“View Description”** to open vulnerability details in a modal.
- Click **“References”** to see safe external resources.
- Click **CVSS v2/v3 score headers** to sort the table by severity.



Stored CVEs

← Back to Home

ID	Published	Last Modified	Description	CVSS v3 Score	CVSS v3 Severity	CVSS v2 Score	CVSS v2 Severity	References
CVE-2020-1467	2020-08-17 19:15:14	2024-11-21 05:10:36	View Description	10.0	CRITICAL	7.2	HIGH	References
CVE-2019-1365	2019-10-10 14:15:18	2024-11-21 04:36:33	View Description	9.9	CRITICAL	9.0	HIGH	References
CVE-2019-1384	2019-11-12 19:15:12	2024-11-21 04:36:36	View Description	9.9	CRITICAL	6.5	MEDIUM	References
CVE-2016-4171	2016-06-16 14:59:51	2025-04-12 10:46:40	View Description	9.8	CRITICAL	10.0	HIGH	References
CVE-2016-7182	2016-10-14 02:59:32	2025-04-12 10:46:40	View Description	9.8	CRITICAL	10.0	HIGH	References
CVE-2017-3059	2017-04-12 14:59:03	2025-04-20 01:37:25	View Description	9.8	CRITICAL	10.0	HIGH	References
CVE-2017-3060	2017-04-12 14:59:03	2025-04-20 01:37:25	View Description	9.8	CRITICAL	10.0	HIGH	References

I selected these columns because they represent the most important information for this vulnerability dataset:

- **CVE ID** – unique identification.
- **Published / Last Modified** – track freshness and patches.
- **Description** – human-readable summary.
- **CVSS v2/v3 scores and severities** – prioritize which vulnerabilities to address first.
- **References** – quick access to vendor advisories, patches, or additional context.

If no CVEs are available (e.g., API call fails or the database is empty which means it wasn't fetched first), the user sees a clear error page rather than a blank screen.

View Table

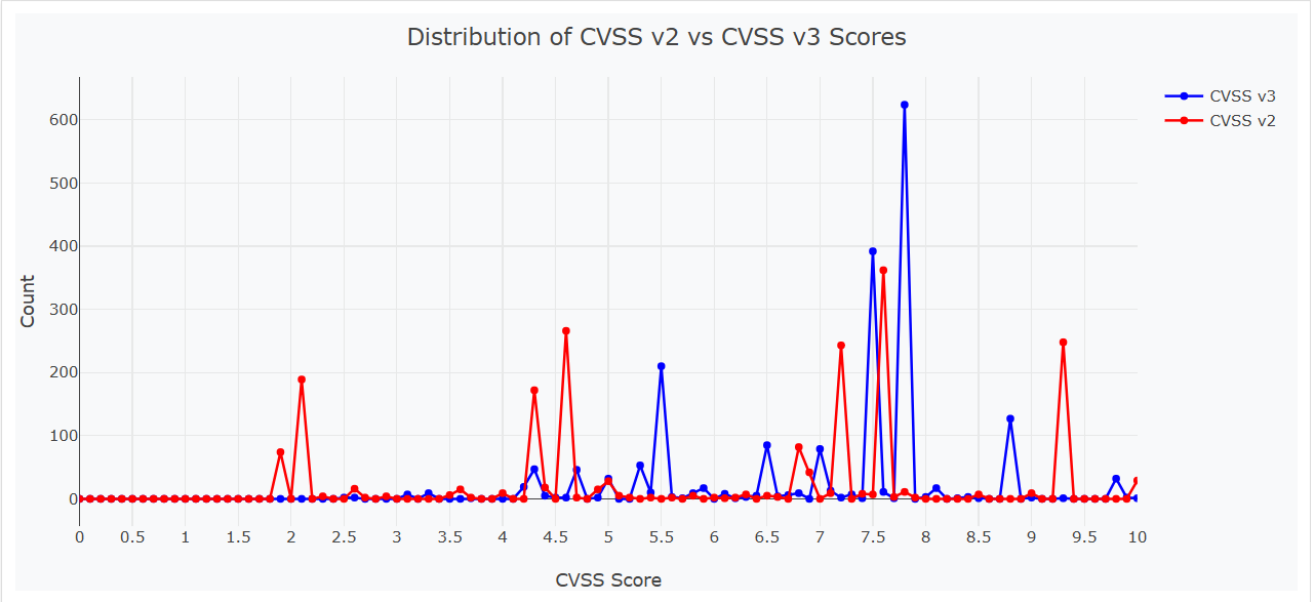
This option displays the CVEs already stored in the **local database**. Unlike **Fetch CVEs**, it does not contact the NVD API or stream results dynamically. Instead, it instantly renders the completed table, providing a **static view** but with a dynamic table of the dataset as it exists in the database at that moment.

Analytics & Graphs

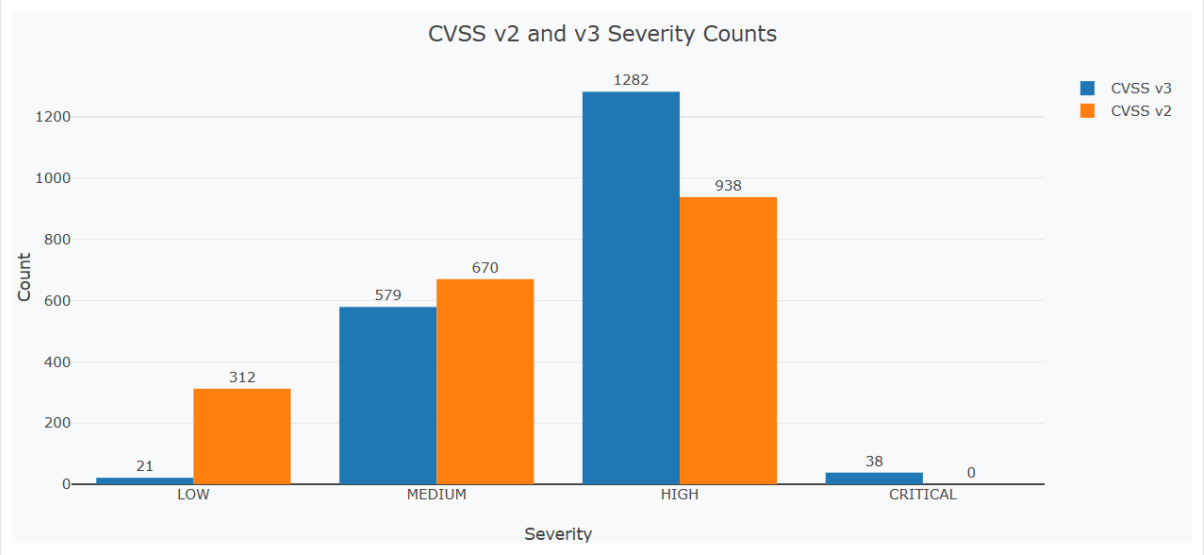
Although not required, I added this feature to provide insight into trends, distributions, and correlations that raw tables alone cannot reveal. Graphs and the word cloud are rendered dynamically in the browser using **Plotly.js** and **WordCloud2.js**, with CVE data embedded as JSON via **Jinja2**. This ensures **fast, interactive rendering** without server recomputation. Users can hover and interact with the graphs to explore the data.

Graphs Included:

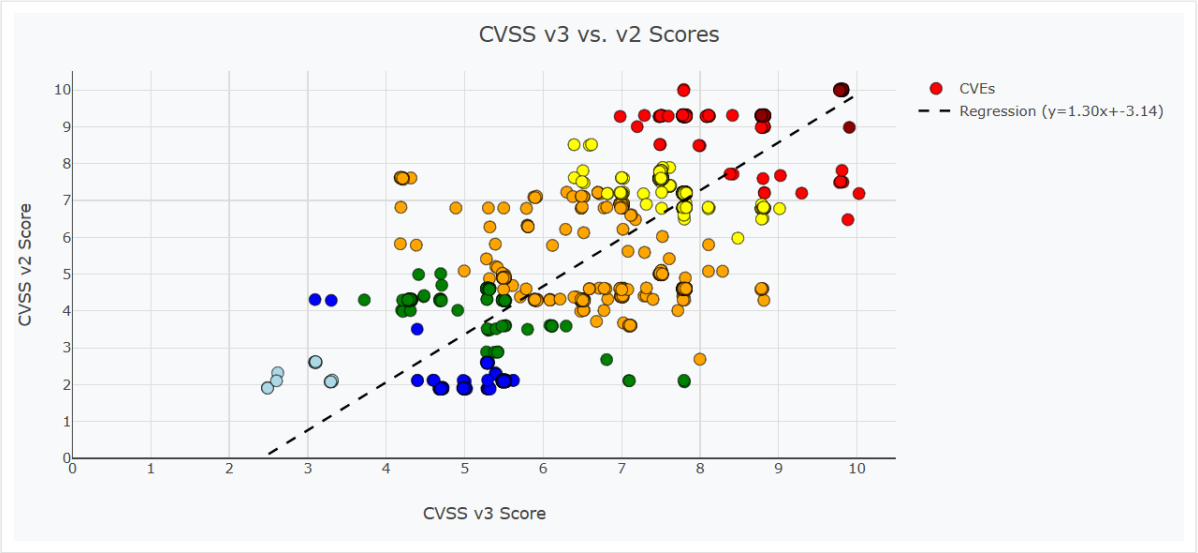
1. **Score Distribution (Histogram)**
    - Compares CVSS v2 and CVSS v3 scores side by side.
    - Helps understand whether the newer scoring system (v3) rates vulnerabilities differently from v2.



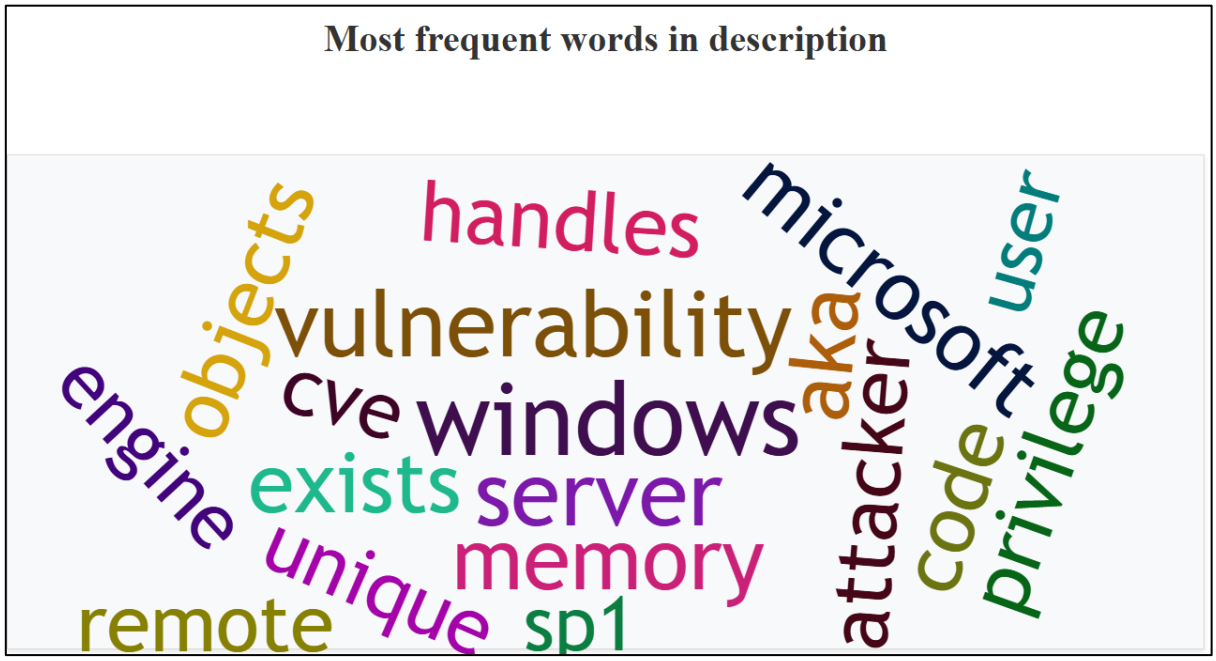
2. **Severity Counts (Bar Chart)**
    - Displays the count of CVEs by severity level (LOW, MEDIUM, HIGH, CRITICAL) for both v2 and v3.
    - Useful for quickly gauging the risk distribution of vulnerabilities.



3. **CVSS v3 vs v2 (Scatter Plot + Regression Line)**
    - Plots each CVE's v3 score against its v2 score, color-coded by average severity.
    - Includes a **regression line** to show overall correlation between v2 and v3 scores.
    - Adds **jitter** to avoid overlapping points, making clusters easier to see.



4. **Word Cloud (Description Keywords)**
    - Extracts the most frequent words from CVE descriptions, excluding common stopwords.
    - Highlights recurring vulnerability types (e.g., “attacker”, “remote”, “vulnerability”, “server”).



Lastly, I have created a **requirements.txt** file featuring all the Python dependencies required for the project, ensuring that the application can be installed and run consistently across environments.

I have also included a **Dockerfile** that sets up a lightweight Python container, installs the dependencies, copies the project files, exposes the appropriate port, and runs the FastAPI server. This allows the entire application to be easily deployed and executed as a Docker container, providing a reproducible and isolated environment for testing and production

I downloaded Docker Desktop and ran the command:

docker build -t cve-webapp .

```
(.venv) C:\Users\Tal\PycharmProjects\fullstack_challenge>docker build -t cve-webapp .
[+] Building 1.9s (10/10) FINISHED
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 506B
 => [internal] load metadata for docker.io/library/python:3.12-slim
 => [internal] load .dockerignore
 => => transferring context: 207B
 => [1/5] FROM docker.io/library/python:3.12-slim@sha256:abc799c7ee22b0d66f46c367643088a35e048bbabd81212d73c2323aed38c64f
 => [internal] load build context
 => => transferring context: 39.00kB
 => CACHED [2/5] WORKDIR /app
 => CACHED [3/5] COPY requirements.txt .
 => CACHED [4/5] RUN pip install --no-cache-dir -r requirements.txt
 => [5/5] COPY . .
 => exporting to image
 => => exporting layers
 => => writing image sha256:5353be505a86f7812c2cc6a995549c9a03ab26a4c88942d50f2fd211c2eda20f
 => => naming to docker.io/library/cve-webapp
```

And then this command:

docker run --env-file .env -p 8080:8080 cve-webapp

```
(.venv) C:\Users\Tal\PycharmProjects\fullstack_challenge>docker run --env-file .env -p 8080:8080 cve-webapp
INFO:     Started server process [1]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://0.0.0.0:8080 (Press CTRL+C to quit)
```