Added a **User-Agent** that mimics a real browser,reducing the chance of being blocked or flagged.

**async def fetch(session, url, params=None):**

Makes an asynchronous HTTP GET request using an aiohttp session. Includes: params: query parameters for the URL (like q=search+term)

headers: to simulate a browser timeout: sets max wait time for response (30 seconds) async - Allows multiple web requests to run concurrently (non-blocking), which speeds up scraping multiple pages.

**async def fetch\_page()**

Calls the fetch() function to get one BROWSER search results page (e.g., page 1, 2, 3…). Parses the page HTML using BeautifulSoup.

Fetches tittles and urls with respective to query.

**def get\_all\_text\_from\_url(url):**

Extract and clean all text content from a URL->it fetches and extracts readable text from a webpage effectively.

Sends a GET request to the given url with headers to mimic a browser. Parses the HTML using BeautifulSoup.

Removes unwanted tags (<script>, <style>) that don't contain useful content. Extracts visible text, removes extra whitespace. Returns a clean, human-readable string of text.

**def split\_text\_into\_chunks(text, chunk\_size):**

Split text into chunks of approximately equal size Input: A long block of text and a chunk\_size (usually in characters).

Output: A list of chunks, each containing complete sentences and approximately chunk\_size characters.

Logic: Splits the text into sentences using a regular expression that preserves sentence-ending punctuation.

Iteratively adds sentences to a chunk until adding the next one would exceed the target chunk size. Starts a new chunk and repeats.

**def process\_text\_content(texts, chunk\_size)**

Uses asyncio.get\_event\_loop() + run\_in\_executor to: Run split\_text\_into\_chunks() concurrently for each text input.

It splits the text at each sentence (by . ).

It adds sentences to the current chunk until adding another one would exceed the chunk\_size. Then it starts a new chunk.

**async def query\_ollama\_llm(prompt)**

Purpose: Queries Ollama's local LLM llama 3.2 to generate a response based on a prompt.

Sends POST request to /api/generate with prompt.

**async def get\_embeddings\_from\_ollama**(text\_chunks)

Gets vector embeddings for each text chunk using Ollama’s local embedding model **(nomic-embed-text**).

For each chunk: If non-empty, sends a POST request to Ollama’s /api/embeddings. Appends the embedding or a zero-vector if it fails.

ex - { "model": "nomic-embed-text", "prompt": "we will win." } Output: We get back an embedding, which is a list of float numbers (e.g, a 768-dimensional vector): [ 0.00234, -0.0317, 0.1075, ..., 0.0541 ]

 async **def query\_vector\_store**

function is responsible for querying the **FAISS index** to retrieve the top-k most relevant results based on the query embedding

The query\_embedding (a 768-dimensional vector) is converted into a NumPy array with the correct shape and data type (float32).

FAISS requires the input to be in this format for similarity search.

**async def fetch\_and\_process\_data(search\_query):**

A ClientSession is created to manage HTTP requests.

Search results are fetched for up to 3 pages (adjustable).

Results are stored in the results list.

Each text is split into smaller chunks (e.g., 1000 characters each).

Each chunk is converted into an embedding using Ollama's embedding model.

The metadata (title, URL, chunk, embedding) is stored in the data list.

The metadata is saved to a CSV file (search\_data.csv).

The embeddings are stored in a FAISS index (faiss\_index.index) for fast

similarity search.