1. Programming Language, Database, and Libraries

Programming Language: **Python**

Widely used programming language for AI and machine learning tasks. Ease of use of libraries for machine learning, image processing, and web development.

Specifically, libraries like **PyTorch** (for deep learning), **FAISS** (for efficient similarity search), and **CLIP** (for multimodal image-text search) are well-supported in Python.

Database: **FAISS (Facebook AI Similarity Search)**

FAISS is a powerful library for performing similarity search.

Since the goal of the project is to index image embeddings and allow efficient search by similarity It is highly optimized for performance and can handle a large number of embeddings, making it suitable for scaling the system with large datasets.

Third-party Libraries:

CLIP (Contrastive Language-Image Pre-Training) by OpenAI: A state-of-the-art multimodal model for linking images and text. CLIP allows us to encode images and text into embeddings that can be compared for similarity.

Torch (PyTorch): A widely used deep learning framework to run and manipulate models like CLIP.

FastAPI: A high-performance web framework for building APIs quickly and efficiently. It is asynchronous, making it ideal for handling concurrent requests at scale.

PIL (Pillow): A Python Imaging Library for image processing, used here for handling image downloading and transformations.

Requests: For making HTTP requests to download images.

JSON: For serializing and deserializing metadata to and from files.

2. Scalability and Maintainability

- **Scalability**:
    - **FAISS** can handle millions of vectors by using optimized indexing methods like **IVF** (Inverted File Index) and **PQ** (Product Quantization). This is essential when dealing with a large number of image embeddings.
    - **Asynchronous handling**: Using **FastAPI** allows the backend to handle many concurrent requests. This is crucial in production systems where high throughput is needed. However please set this in your environment,
      export KMP_DUPLICATE_LIB_OK=TRUE
    - **Modular design**: The system is broken into clear components—image download, indexing, and searching.

  If I had access to GCP Google vector matching engine then the scalability and availability would be better handled. As of now if being deployed the DB on cloud would need to have manual and traditional approaches for horizontal scalability.

  This can also be dockerized and the image can be pushed on cloud run as a google cloud run and the images can be pushed to a container registry on cloud linked to versions on the github repo…

- **Maintainability**:

  Clean and modular code: The system is divided into distinct classes and functions such as `ImageIndexer`, `ImageSearcher`, and utility functions like `download_image`. etc

  Here is the directory structure depicting the segregation functionalities.

```
# Directory structure (to be created manually or with a setup script)

# image-search-engine/

# ├── app/

# │   ├── main.py

# │   ├── indexer.py
```

```
# │   ├── searcher.py
# │   ├── models.py
# │   └── utils.py
# ├── frontend/
# │   └── streamlit_app.py
# ├── data/
# │   ├── images/
# │   └── embeddings.index
# ├── requirements.txt
# └── README.md
```

- **Documentation**: Each function has detailed docstrings explaining the arguments, return values, and purpose for the backend. The front end is

  streamlit, just so that it can run live with the end points working.

- **Error handling**: For instance, when downloading images, the code checks if the HTTP request succeeds using `raise_for_status`.

  If an image is corrupted we can add more error handling for it.

3. Search Quality Considerations

- **Relevance Ranking**:
  - **Cosine similarity** is used to measure the relevance of search results by comparing the vector embeddings.

    Cosine similarity is ideal for CLIP type embeddings which in our case are both for text and images.

    More here: https://openai.com/index/clip/

If for instance we only had to implement image search by searching images, euclidean distance would have also worked well.

- **Performance Optimization**:
    - **Caching**: Frequently used queries can be cached (e.g., with Redis or in-memory storage) to reduce the computational cost of repeated searches.
- **Potential Improvements for Search Accuracy**:
    - **Fine-tuning**: CLIP model can be fine tuned on domain-specific data to improve the accuracy of the image-text embeddings. (I did not have domains specific data so did not try)
    - **Multimodal Fusion**: Combining information from both text and image queries (e.g., using a hybrid search method that weights both image and text results) could improve search accuracy.
- **Documentation of Approaches**:
    - **FAISS Indexing**: The `IndexFlatIP` is used for efficient similarity search based on cosine similarity.
    - **Text-Image Search**: CLIP is used to bridge the gap between text and image modalities.

4. Validation, Error Handling, and Other Considerations

- **Validation**:
    - **File Existence and corruption**: We can validate the existence of files before attempting to load them or download new images.
- **Error Handling**:
    - **Image Download**: In the `download_image` function, `response.raise_for_status()` ensures that HTTP errors (like a 404 or 500 error) are caught. More detailed error messages could be added to return to the client (e.g., invalid image URLs, unsupported formats).

- **Rate Limiting**: In a production environment, we can implement rate limiting on image indexing or search endpoints to prevent abuse and ensure fair usage.

- **Logging**: Integrate logging to track errors, performance metrics, and system status.