Multi-Modal Image Retrieval System Design

Main Components

- Data pipeline
- Feature store
- Backend/inference
- Frontend

Main features

- Separation of concerns
- Modular design
- Scalable design
- Production grade architecture and code

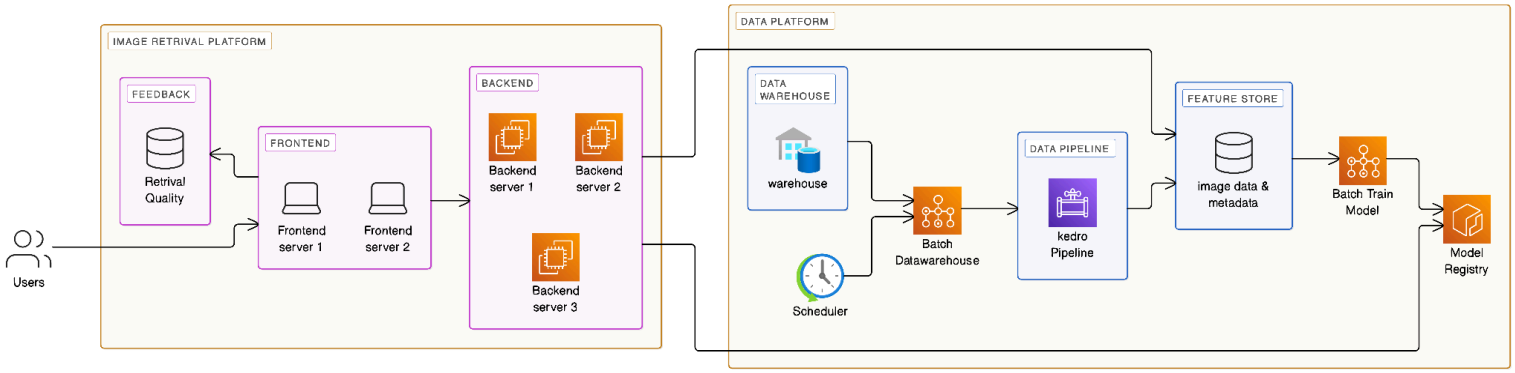


Figure 1. System architecture

The system is composed of two main platforms: one dedicated to handling data and the other focused on inference and user interaction. This design offers a modular and flexible architecture, ensuring seamless integration between the two components. The scheduler is used to kick off the pipeline and do training in a batch process as it's more appropriate for this kind of system.

Data Pipeline

A data pipeline is developed to manage feature engineering and data science tasks. Built using Kedro, the pipeline applies software engineering principles to data science projects, ensuring maintainability and production-grade quality. This approach eliminates the common issue of disorganised notebooks in data science workflows. The pipeline is scalable, allowing independent paths within its DAG to run in parallel. Kedro also emphasises data lineage, providing a visual representation of how each node processes the data. The pipeline can be integrated into a CI/CD system, as it supports test creation, adhering to software development standards.

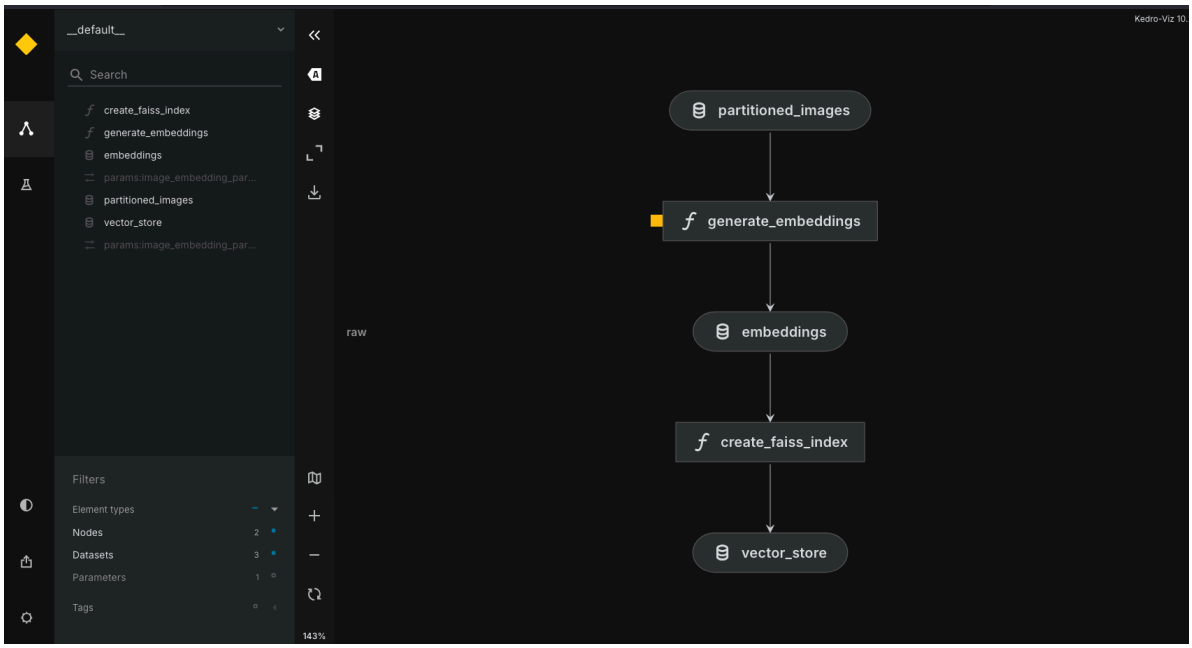
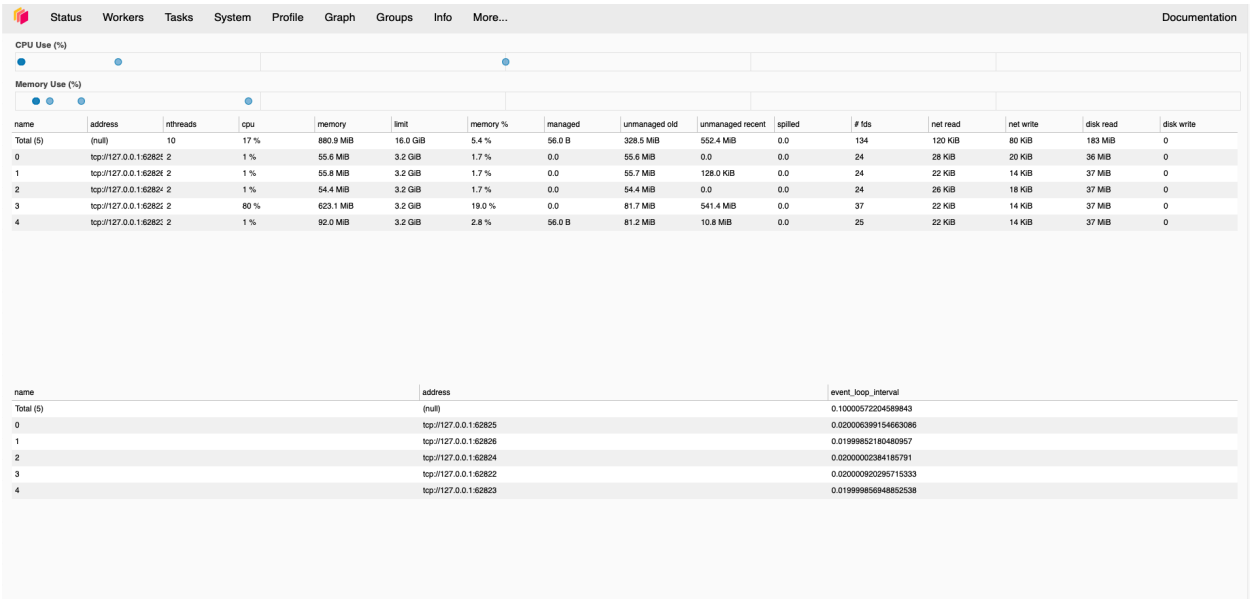


Figure 2. Kedro pipeline

An additional feature was implemented to enable distributed processing using Dask, which is particularly useful for handling memory-intensive tasks like processing multidimensional image arrays. This capability ensures the pipeline is future-proof and can be activated with a simple command, requiring no code modifications. The pipeline can also leverage Dask dataframes for parallel I/O operations, enabling efficient handling of large files in chunks.



name	address	nthreads	cpu	memory	limit	memory %	managed	unmanaged old	unmanaged recent	spilled	# fds	net read	net write	disk read	disk write
Total (5)	(null)	10	17 %	880.9 MiB	16.0 GiB	5.4 %	56.0 B	328.5 MiB	552.4 MiB	0.0	134	120 KiB	80 KiB	183 MiB	0
0	tcp://127.0.0.1:62825	2	1 %	55.6 MiB	3.2 GiB	1.7 %	0.0	55.6 MiB	0.0	0.0	24	28 KiB	20 KiB	36 MiB	0
1	tcp://127.0.0.1:62826	2	1 %	55.8 MiB	3.2 GiB	1.7 %	0.0	55.7 MiB	128.0 KiB	0.0	24	22 KiB	14 KiB	37 MiB	0
2	tcp://127.0.0.1:62824	2	1 %	54.4 MiB	3.2 GiB	1.7 %	0.0	54.4 MiB	0.0	0.0	24	26 KiB	18 KiB	37 MiB	0
3	tcp://127.0.0.1:62822	2	80 %	623.1 MiB	3.2 GiB	19.0 %	0.0	81.7 MiB	541.4 MiB	0.0	37	22 KiB	14 KiB	37 MiB	0
4	tcp://127.0.0.1:62823	2	1 %	92.0 MiB	3.2 GiB	2.8 %	56.0 B	81.2 MiB	10.8 MiB	0.0	25	22 KiB	14 KiB	37 MiB	0

name	address	event_loop_interval
Total (5)	(null)	0.10000572204589843
0	tcp://127.0.0.1:62825	0.020006399154663086
1	tcp://127.0.0.1:62826	0.01999852180480957
2	tcp://127.0.0.1:62824	0.02000002384185791
3	tcp://127.0.0.1:62822	0.020000920295715333
4	tcp://127.0.0.1:62823	0.019999856948852538

Figure 3. Dask workers running a Kedro pipeline

With Dask we can further expand our processing over a cluster.

Model

For embedding, the Contrastive Language-Image Pre-training (CLIP) model was chosen due to its ability to map both images and text into a shared embedding space. CLIP is trained on image-text pairs and supports zero-shot prediction. To perform embedding searches, Faiss is utilised, offering efficient similarity search capabilities.

In search engines, both speed and accuracy are critical. Faiss excels in speed by storing indices in memory, enabling rapid retrievals while maintaining a minimal footprint, which helps keep operational costs low. It also supports GPU acceleration and parallel processing, striking a balance between speed and accuracy—essential for search engines. For this case study, a flat index was selected as it is the fastest option and suitable for small-scale searches (between 1,000 and 10,000 entries). However, the system can easily transition to larger-scale index types if needed.

Future enhancements may include transitioning to a full-fledged vector store like Chroma, which offers greater development flexibility and functionality, such as database-like persistence, albeit with slower query times.

https://github.com/facebookresearch/faiss/wiki/Guidelines-to-choose-an-index

https://www.capellasolutions.com/blog/faiss-vs-chroma-lets-settle-the-vector-database-debate

Feature store

The feature store houses images, embeddings, and image byte data, enabling data science teams to reuse this information without repeating EDA, feature engineering, or modeling tasks. This promotes efficiency and cost savings. Feast is used for the feature store, providing two storage options: an offline store for training and an online store for inference. The online store is optimised for fast data access, while the offline store prioritises storage capacity and throughput.

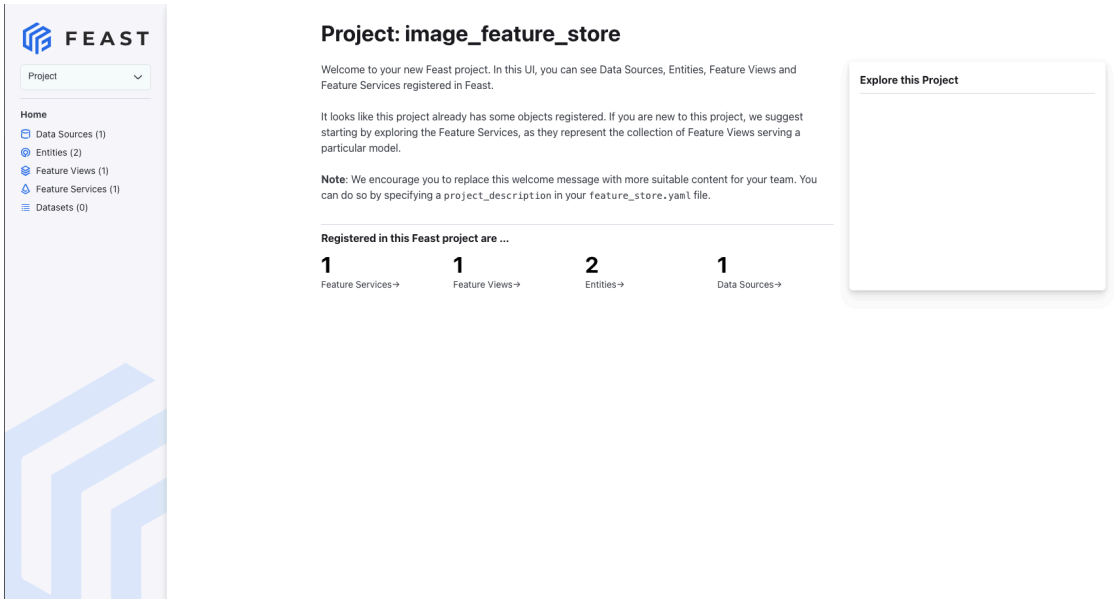


Figure 4. Feast project

For the local store we use parquet which is a column based file system designed for efficient data storage and quick data access. The online store is running on sql lite which persists the feature and can get fast access using indexes.

Model Registry

The local file system serves as the storage for models. Due to limitations in local development, external storage systems—often proprietary—are inaccessible. The local file system mimics a typical model registry, essentially functioning as a database of models. Models can be tested and approved for production readiness. The registry can be configured within Kedro using MLflow for model management.

Backend

The backend features an API for inference, built with FastAPI. This API includes basic schema validation for incoming requests and loads the model into memory to ensure quick response times.

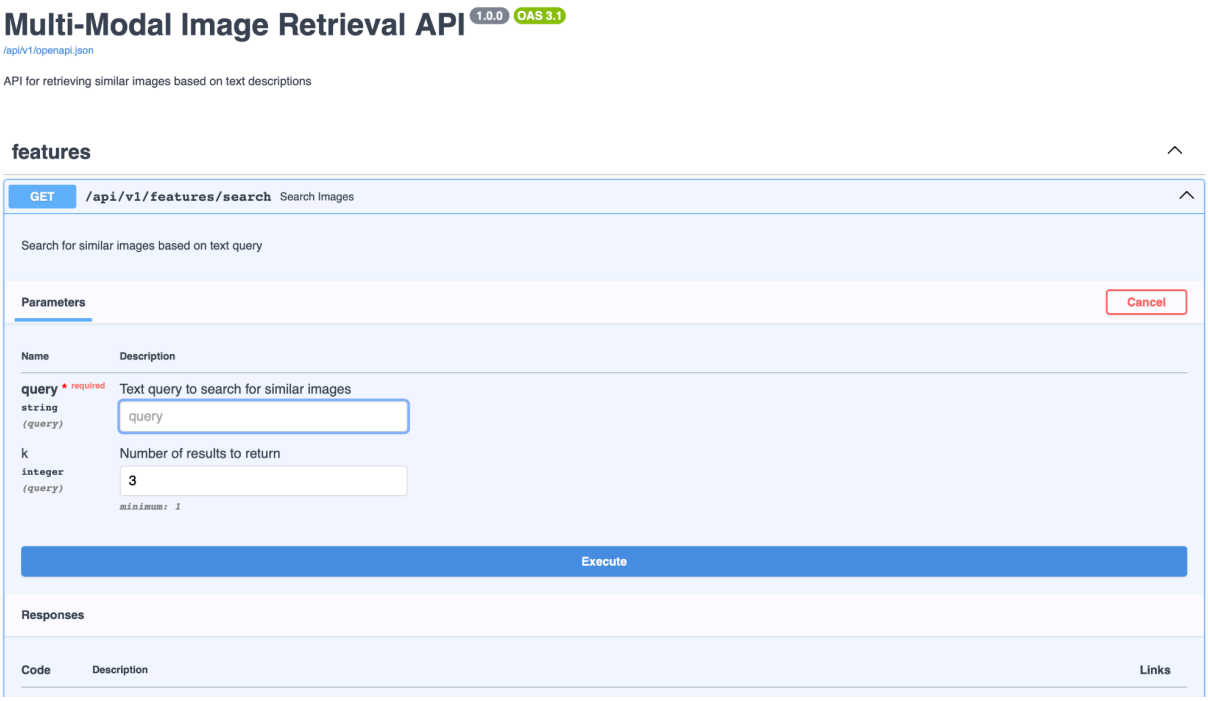


Figure 5. Project backend

The backend is designed to scale, supporting multi-model serving. This allows multiple models to be loaded and unloaded between RAM and disk, optimizing resource usage and reducing operational costs.

Frontend

To minimise redundant backend calls, search results for *k* are stored in the browser's local storage. If a user searches for fewer than *k* results for the same query, the system retrieves them from memory instead of making additional backend requests.

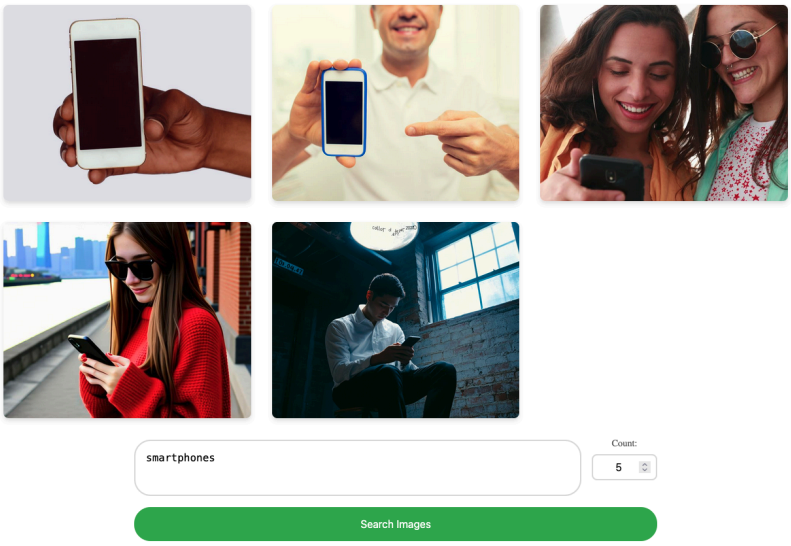


Figure 4. System frontend

In all the system can be easily deployed and containerised using the already existing docker files.

Extended System

The design can be adapted to assist visually impaired users by integrating a speech-to-text model directly into the browser, using frameworks like TensorFlow.js or ONNX Runtime Web. Running models on the client side ensures system responsiveness. Instead of returning images, the system would provide text descriptions of the top *k* images. These descriptions could then be passed to a text-to-speech engine, also running in the browser. The system could leverage browser-stored information, such as the user's IP address, to select a voice with an accent matching their geolocation.

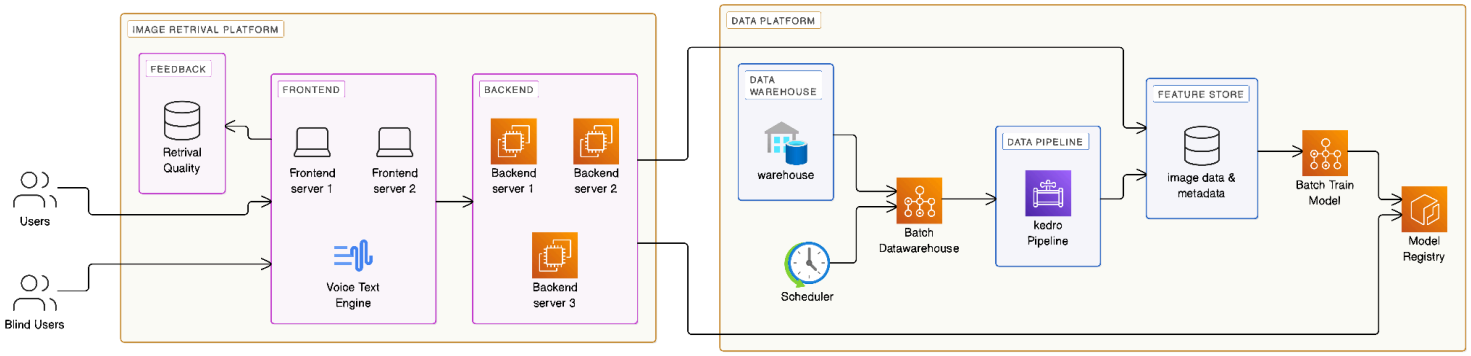


Figure 4. Extended system

We can further collect feedback from users on the quality of retrieved images with a simple thumbs up/down emotions. These can then form part of a reinforcement strategy where we can define a reward function using feedback in more to build a way to fine tune the models since we do not have access to the image-text pair.

Challenges

- Memory leaks in Dask and Faiss can lead to thread crashes, requiring careful management.
- Feast displays warnings and needs updates, such as transitioning from `on_start` events to `lifespans` in its FastAPI integration.
- Managing and sharing files between individual projects on a local file system can be cumbersome.
- The absence of image-text pairs limits the ability to fine-tune the CLIP model effectively.