End-to-End Image Classification system for Big Data Environments

Chiara Cintioni   Denise Falcone

Project Objectives

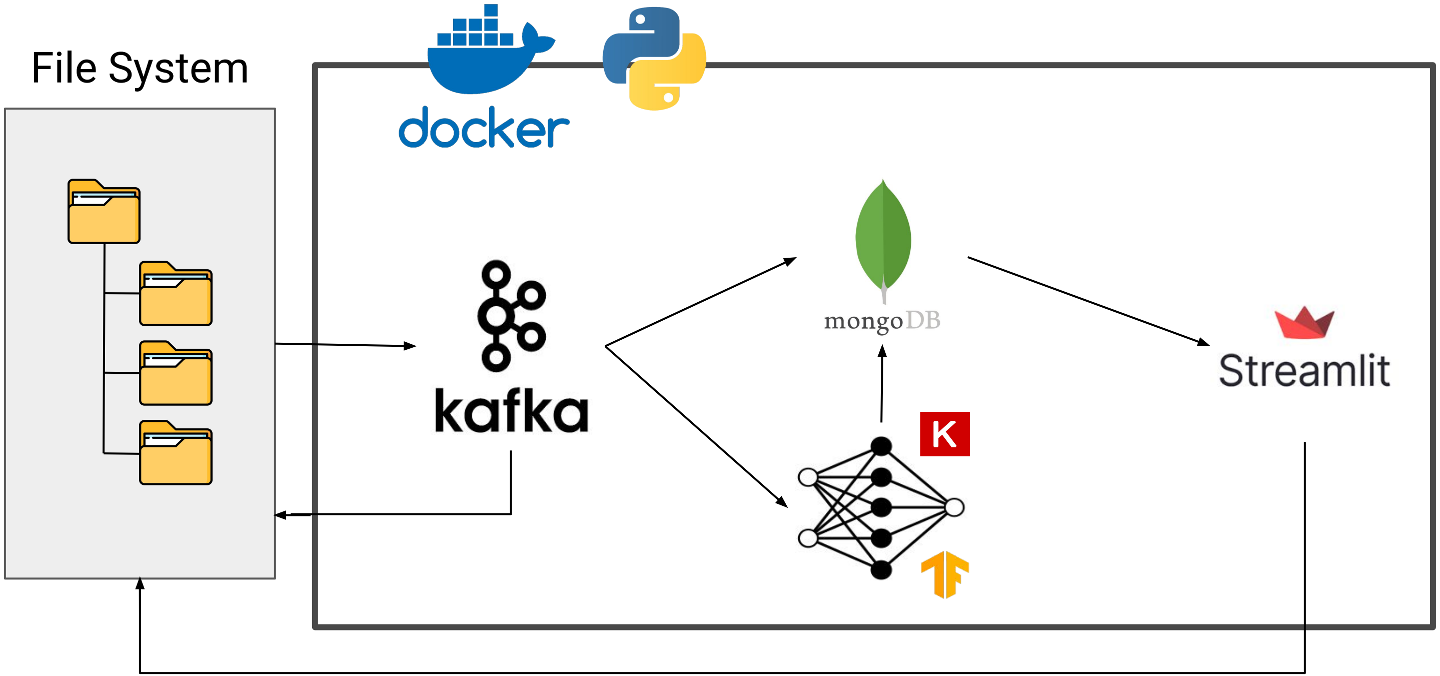
In Big Data environments, large volumes of images often need to be processed, analyzed, and classified in real time.

Traditional monolithic pipelines are not scalable, hard to maintain, and often tightly coupled to specific models or tasks.

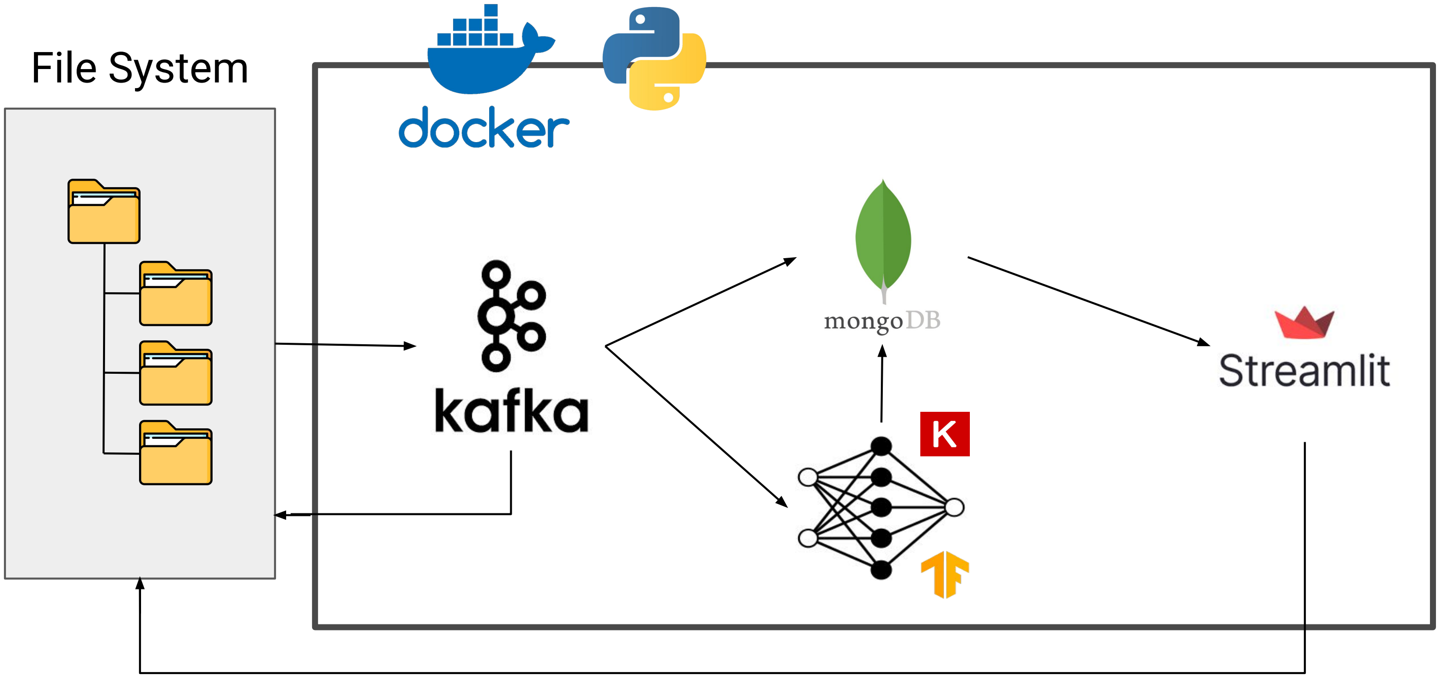
Our project objective is to:

- Design a **real-time image processing pipeline** suitable for Big Data environments.
- Ensure the system is **model-agnostic** — capable of working with any image classification or object detection model.
- Develop a **web-based interface** to visualize classification results dynamically and interactively.

Architecture



Methodology



Technologies

- **Python**: programming language used.
- **Docker**: containerized every component for portability, easy deployment, and isolation.
- **Apache Kafka**: used as a message broker to stream image paths in real time from producers to consumers. Enabled parallelism and decoupled ingestion from processing.
- **MongoDB**: NoSQL database to store classification results in a flexible document format, ideal for unstructured or semi-structured data.
- **Streamlit**: used to make a web application to visualize classification outputs in real time.
- **TensorFlow & Keras**: python libraries used to integrate a deep learning model.

Docker

Docker lets us run all components (Kafka, MongoDB, services) in isolated, reproducible environments. This avoids compatibility issues and ensures the system works the same everywhere.

- bigdataproject-main
  - streamlit-1
  - mongo-express-1
  - mongo-1
  - zookeeper
  - image-processing-service-1
  - kafka-ui
  - broker

Docker desktop visualization.

```yaml
image-processing-service:
  image: image-processing-service
  build:
    context: ./image_processing
    dockerfile: Dockerfile
  depends_on:
    broker:
        condition: service_healthy
    mongo:
        condition: service_started
  environment:
    KAFKA_URL: 'broker:29092'
    KAFKA_PATH_TOPIC: get_path_topic
    DATA_FOLDER: /data
    FOLDER_TO_WATCH: /data/imageserver
    DATASET_FOLDER: /data/dataset
    PROCESSED_FOLDER: /data/processed_images
    TO_PROCESS_FOLDER: /data/to_process_images
    MONGO_URL: mongodb://root:example@mongo:27017/
    MONGO_DB: prova_db
    MONGO_METADATA_COLLECTION: images_metadata
  volumes:
    - ./data:/data
```

Docker custom service.

File System

data

dataset

Contains all the images

imageserver

Used to simulate real time data

to_process_images

Images ready to be classified

processed_images

Stores classified images

Real Time Data Simulation

A real-time data simulator has been implemented to emulate data real time streams.

The ***RealTimeDataSimulator class*** copies images from *DATASET_FOLDER* to *FOLDER_TO_WATCH*, which is continuously monitored by the data processing pipeline.

```python
def run(self):
    logging.info("Simulating real-time data")

    #5-minute delay to make the consumer model ready to consume the data
    #time.sleep(300)

    img_lst = os.listdir(DATASET_FOLDER)

    # Shuffle the list of files to simulate randomness
    random.shuffle(img_lst)

    for filename in img_lst:

        file_path = os.path.join(DATASET_FOLDER, filename)
        dest_path = os.path.join(FOLDER_TO_WATCH, filename)

        if os.path.isfile(file_path):

            file_func.move_file(src_path=file_path, dest_path=dest_path)

            # Simulate a 10-second delay between file copies
            time.sleep(10)
```

run() function of *RealTimeDataSimulator* class.

Apache Kafka

PRODUCER

NewImagePathProducer class

- Automatically detect new image files.
- Move them to another folder.
- Send their paths to Kafka for downstream processing.

CONSUMER

ImageProcessingConsumer class

- Consume image paths from Kafka.
- Classify the image using a deep learning model.
- Store metadata in MongoDB.

Apache Kafka Producer

```python
def run(self):
    while True:

        new_files = self.new_file_exists()

        if not new_files:
            logging.info("No new files found.")
            time.sleep(10)
            continue

        #logging.info(f"Current files: {new_files}")

        for file in new_files:
            self.send_file_path(file)

        # wait 10 seconds before checking for new files again
        time.sleep(10)

def new_file_exists(self):

    all_files = set(os.listdir(FOLDER_TO_WATCH))
    new_files = all_files - set(os.listdir(TO_PROCESS_FOLDER))

    return new_files
```

run() function of *NewImagePathProducer* class.

```python
def send_file_path(self, file):

    try:
        src_path = os.path.join(FOLDER_TO_WATCH, file)
        dest_path = os.path.join(TO_PROCESS_FOLDER, file)
        file_func.move_file(src_path=src_path, dest_path=dest_path)
        file_path = os.path.join(TO_PROCESS_FOLDER, file)
        message = {"file_path": file_path}
        self.producer.send(KAFKA_TOPIC, message)
        #logging.info(f"Sent path to Kafka: {message}")
    except Exception as e:
        logging.warning(f"Error processing {file_path}: {e}")
```

send_file_path() function of *NewImagePathProducer* class.

Apache Kafka Consumer

```python
def run(self):

    import tensorflow as tf

    try:

        #logging.info("Consumer process started – loading model")
        self.prediction_model = tf.keras.models.load_model('model/model_0505.keras')
        logging.info("Model loaded successfully")

        for message in self.consumer:
            file_path = message.value.get("file_path")
            logging.info(f"[PID {os.getpid()}] Processing image {file_path}")
            dest_path = os.path.join(PROCESSED_FOLDER, os.path.basename(file_path))
            try:
                self.collection.insert_one(self.create_json(src_path=file_path,
                                                            dst_path=dest_path,
                                                            timestamp=message.timestamp))
            except Exception as e:
                logging.error(f"Error processing image {file_path}: {e}")

            #logging.info(f"Moving file from {file_path} to {dest_path}")
            file_func.move_file(src_path=file_path, dest_path=dest_path)

    except Exception as e:
        logging.error(f"Error: {e}")

    finally:
        self.consumer.close()
        logging.info("Consumer closed")
```

run() function of *ImageProcessingConsumer* class.

```python
def create_json(self, src_path, dst_path, timestamp):

    import utils.data_preprocessing as data_preprocessing

    img_to_predict = data_preprocessing.preprocess_image(src_path)
    logging.info(f"Image {os.path.basename(src_path)} to predict shape: {img_to_predict.shape}")
    prediction = float(self.prediction_model.predict(img_to_predict)[0][0])

    img = Image.open(src_path)
    width, height = img.size
    img.close()

    if prediction > 0.8:
        pred = "Tumor"
    else:
        pred = "No Tumor"
    return {
        "file_name": os.path.basename(src_path),
        "file_path": dst_path,
        "image_dimensions": {
            "width": width,
            "height": height
        },
        "image_size_bytes": os.path.getsize(src_path),
        "label": pred,
        "prediction_score": prediction,
        "timestamp": timestamp
    }
```

create_json() function of *ImageProcessingConsumer* class.

Parallelism in Kafka Consumer

To process multiple images simultaneously and store results in MongoDB, we used Kafka's native support for concurrent processing by:

- Using a multi-partition topic
- Creating a shared consumer group
- Running multiple consumer instances in parallel using Python multiprocessing

Each consumer handles different partitions independently, enabling scalable and efficient processing.

```
INFO - [PID 17] Processing image /data/to_process_images/000022.png
INFO - Preprocessing image: /data/to_process_images/000022.png
INFO - [PID 18] Processing image /data/to_process_images/000016.png
```

Example of two consumers working in parallel.

Deep Learning Model

While **the architecture is generic and supports various object detection or image classification tasks**, for testing and demonstration purposes we used a deep learning model that performs binary classification on breast histopathological images to determine the presence (1) or absence (0) of a tumor.

Example of classification metadata in Streamlit.

MongoDB

images_metadata

Contains metadata results from image processing.

hidden_metadata

Metadata hidden by the user on the web app.

_id	file_name	file_path	image_dimensions	image_size_bytes	label	prediction_score	timestamp
687fb74947d01057cbdc219d	000010.png	/data/processed_images/000010.png	{ "width": 96, "height": 96 }	25987	Tumor	0.8586311340332031	17532004574

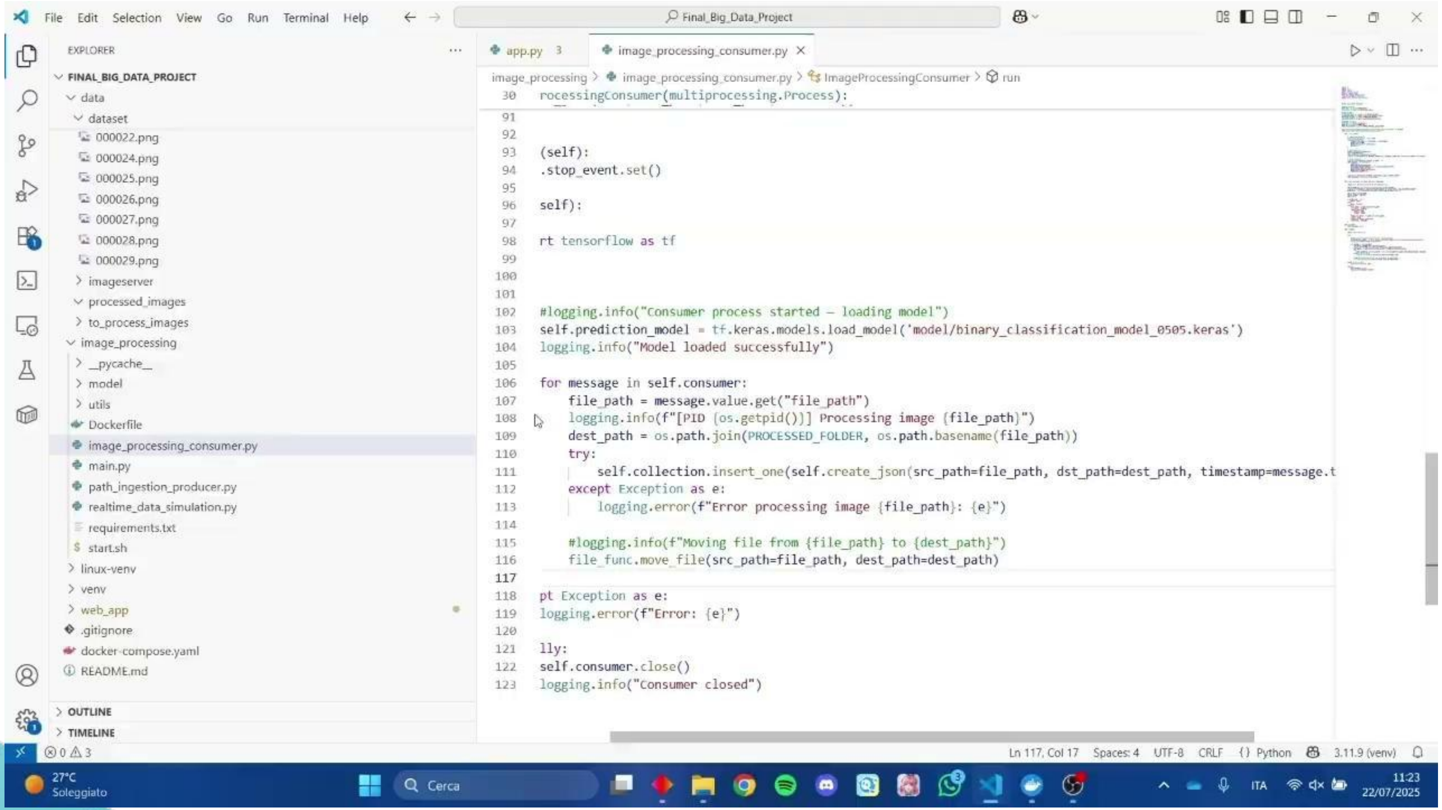
Example of metadata saved in MongoDB.

Streamlit

We used Streamlit to build a web interface for visualizing results from our pipeline.

It allows us to display predictions, metadata, and image previews in real time.

Achieved result: Demo



Conclusions & Future Works

Using Kafka, Docker, and MongoDB, the solution proposed **supports real-time ingestion, parallel processing,** and **live result visualization**.

The architecture is **model-agnostic**, making it adaptable to any image classification task. A web interface and data simulator helped demonstrate its functionality in a realistic scenario.

Future works:

- Add a feedback mechanism in the web app to improve model accuracy over time.
- Integrate an object detection model to extend functionality beyond classification.
- Scale up the system by increasing the number of Kafka consumers.
- Deploy the pipeline in a distributed cloud environment (e.g., Kubernetes) for real-world scalability.

THANK YOU
FOR THE ATTENTION