Real-Time Image Analytics Pipeline Using Kafka, OpenCV, Elasticsearch, and Hadoop EMR

Project Overview

As part of this project, I was given the task of creating a complete pipeline for image processing in a real time environment. The goal was to analyze 10,000 images, detect objects in them, extract dominant colors, and enable search and visualization of the results. For that, I used modern technologies that are commonly used today in the field of data processing and artificial intelligence.

I started the project locally on my computer. I processed the images using Python, with the help of OpenCV and the YOLOv5 model. I sent the image paths through Kafka, and the results of the analysis, that is which objects were detected and which colors were dominant – were saved into a local JSON file.

After that, I moved the data to the cloud, specifically to the AWS environment. There, I used EMR MapReduce to further process the data and analyze how frequently different objects appear. Once that was done, I indexed the results into Elasticsearch and used Kibana to visualize them.

I combined local and cloud processing because it allowed me to make use of the resources I already had, while also learning how such a system could work live in a real-world environment.

My pipeline consists of the following components:
• Kafka – for sending image paths
• OpenCV and YOLOv4 tiny – for object detection
• Hadoop MapReduce on AWS EMR – for analyzing results
• Elasticsearch and Kibana – for searching and visualizing data

Development environment and tools

As the first step, I created the main directory for the entire project, named:
image-pipeline-project/
I maintained the complete structure in this directory – scripts, dataset, results, and configurations.

I developed the project on the Windows operating system, using Visual Studio Code as the main editor. When I first opened a .py file, the editor automatically suggested installing the Python extension (author: Microsoft).

By clicking Install, the extension was installed and enabled me to:
• run Python scripts directly from the editor,
• automatically recognize the .venv environment.

Alternatively, the extension can be installed manually:

1. Open the Extensions tab (Ctrl + Shift + X)
2. Search for Python
3. Install the first extension by Microsoft

For all system services such as Kafka, Elasticsearch, and Kibana, I used Docker Desktop and docker-compose configurations.

Setting up the Python environment
In the root directory of the project, I created a virtual environment using the command:
python -m venv .venv

Since I work on a Windows system, the activation of the .venv environment before running any scripts was done with the command:
.venv\Scripts\activate

In the activated environment, I installed all the necessary libraries:
pip install kafka-python opencv-python-headless numpy elasticsearch scikit-learn

Installing and Using Docker

To run system services such as Kafka, Zookeeper, Elasticsearch, and Kibana, I used Docker Desktop as a lightweight and efficient solution for local service management.

Docker Installation

1. Visited the website: https://www.docker.com/products/docker-desktop/
2. Downloaded the Windows version

3. Installed Docker using the default installation flow (Next → Next → Finish)
4. After installation, I launched Docker Desktop and verified that services were running in the background

Dataset

For the implementation of this project, I used the Open Images dataset, which was downloaded from the Kaggle platform at the following address:

https://www.kaggle.com/datasets/awsaf49/coco-2017-dataset?resource=download

Although the Kaggle title says “COCO 2017 Dataset,” the actual content includes Open Images formatted to be compatible with COCO-style detection systems. For this pipeline, I used only the images, not the labels.

Dataset Contents:

- train2017/ folder containing 118,287 .jpg images
- val2017/ folder (not used)
- Other files (annotations, JSON files, label maps) were not required since the YOLO model performs detection directly from the image.

Dataset Used in This Project:

- I used only the train2017 folder
- Total images: 118,287
- I manually selected 10,000 images
- These selected images were stored locally in: *image-pipeline-project/images_subset/*

Why I Chose This Dataset

I chose the Open Images dataset because:

- It contains a large number of diverse, real-world images
- The images are high-quality and contain multiple objects per image, allowing richer detections
- It’s compatible with YOLO models using COCO labels

- It can be used without annotations, as YOLO models detect objects based on pre-trained weights

This dataset allowed me to test the pipeline under realistic conditions using images that represent everyday scenarios and a wide range of objects – from people and vehicles to furniture and animals.

Creating and Running copy_images.py

After downloading and extracting the Open Images dataset (with 118,287 images), I wanted to randomly select exactly 10,000 images for use throughout the pipeline. I wrote a script called *copy_images.py* to select and copy a defined number of images into a separate folder for processing.

Execution:

I activated the virtual environment with:

.venv\Scripts\activate

Activating the .venv environment was necessary to isolate all the libraries and dependencies used by the project, which helps avoid conflicts with global installations and ensures stable and consistent execution of scripts on any machine.

Then I ran the script:

python copy_images.py

Exactly 10,000 .jpg images were successfully copied from the local train2017 directory into the existing images_subset/ folder, thereby preparing the image set for further streaming through the pipeline.

Streaming Services Setup – Kafka and Zookeeper

Once the images were ready, I proceeded to set up and launch the required data streaming services. For this purpose, I created a docker-compose.yml file that simultaneously starts Kafka and Zookeeper locally. This established efficient communication between system components.

To launch the services, I used:

docker-compose -f docker-compose.yml up -d

As a result, the Kafka service became available at localhost:9092 and was ready to send and receive messages.

Creating and Running producer.py and consumer.py

Next, I developed two essential scripts, *producer.py* and *consumer.py,* which serve as the functional core for sending and receiving data in the pipeline.

consumer.py was designed to:

- Listen to image paths from the Kafka topic image_paths
- Load images from the local directory
- Analyze them using the KMeans algorithm
- Save results in a JSON file: metadata_output.json

producer.py was responsible for:

- Reading all image files in images_subset/
- Sending their relative paths to the Kafka topic image_paths

To ensure correct real-time data processing, it was necessary to start consumer.py first, since Kafka sends messages immediately, and without additional retention configuration, they may remain unread if the consumer is not active at the time of sending.

Execution

In one terminal:

.venv\Scripts\activate

python consumer.py

In another terminal:

.venv\Scripts\activate

python producer.py

As producer.py sent image paths, consumer.py received and analyzed them in real time. The result of this phase was a structured metadata file: metadata_output.json, forming the basis for the next processing step.

Creating an S3 Bucket for Image Storage

In order to enable the storage of all images and files used in the project, it was necessary to create a dedicated S3 bucket on AWS.

The steps I followed:

1. I opened the website https://s3.console.aws.amazon.com/s3 while logged into my new AWS account.
2. I clicked the “Create bucket” button in the upper-right corner of the page.
3. In the form that appeared, I entered the following details:
    - Bucket name: image-analytics-jhk-2025
      (I carefully chose a unique name that reflects the name of my project)
    - AWS region: I left the default or selected the region closest to me (e.g., eu-central-1 – Frankfurt).
4. In the Object Ownership section, I kept the following option:
    - “ACLs disabled (recommended)”
    - Checked: “Bucket owner preferred”
5. In the Block Public Access settings for this bucket:
    - I disabled the “Block all public access” option, as I wanted the images to be publicly accessible via URL.
6. I confirmed disabling public access by checking the box acknowledging the risk.
7. I left all other settings at their default values (e.g., no versioning, no encryption since the project is in development).
8. I clicked “Create bucket” at the bottom of the form.

The S3 bucket named *image-analytics-jhk-2025* was successfully created.

After that, I uploaded all the images from the images_subset/ folder into this bucket using the AWS Management Console (by clicking “Upload” → adding all .jpg files → “Upload”).

Preparing Files for EMR Processing – Creating hadoop/ Folder and Uploading Scripts

After the images were uploaded to S3, and metadata_output.json was generated locally from the YOLO + Kafka pipeline, it was necessary to prepare the input files for processing via the Amazon EMR cluster.
To keep everything clearly and systematically organized, I decided to create a dedicated subfolder for Hadoop processing within the existing bucket image-analytics-jhk-2025.

Creating the hadoop/ folder in S3:

1. I opened the Amazon S3 console and navigated to my bucket image-analytics-jhk-2025.
2. I clicked the “Create folder” button.
3. I entered the following name for the folder:
   hadoop
4. I confirmed the creation by clicking “Create folder” at the bottom of the form.

Uploading files to hadoop/:

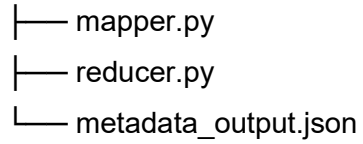
Once the folder was created, I opened it and added the following files:

- mapper.py – a script that processes the JSON input and extracts detected object labels
- reducer.py – a script that counts the frequency of each label
- metadata_output.json – the main metadata file for 10,000 images

Procedure:

1. I clicked “Upload” inside the hadoop/ folder
2. I added all three files from the local directory
3. I clicked “Upload” and waited for all files to appear in the list

I successfully prepared the structure and uploaded the files to the following path:
s3://image-analytics-jhk-2025/hadoop/



These files were later used directly during the definition of the EMR Hadoop Streaming step, which will be described in the next section of the report.

EMR MapReduce Analysis – Processing Detections on AWS

After I successfully generated metadata locally for 10,000 images and uploaded them together with the scripts mapper.py, reducer.py, and metadata_output.json to the hadoop/ folder on S3, I proceeded with setting up an Amazon EMR cluster to process this data using the Hadoop MapReduce model.

1. Creating the required IAM roles

Before launching the EMR cluster, it was necessary to define and assign the appropriate IAM roles that would allow the cluster to access my S3 bucket and execute the defined steps. Since I was using a new AWS account, these roles were not pre-created, so I added them manually through the IAM service.

I opened the IAM console at:
https://console.aws.amazon.com/iam

Then, I created the following two roles in order:

- **EMR_EC2_DefaultRole**
  • Role type: EC2
  • Attached policies:
    - AmazonElasticMapReduceforEC2Role
    - AmazonS3FullAccess
- **EMR_DefaultRole**
  • Role type: EMR
  • Attached policies:
    - AmazonElasticMapReduceRole
    - (optional) CloudWatchAgentServerPolicy – for monitoring and logs

Both roles were successfully created and immediately ready for use during the cluster configuration.

2. Creating the EMR cluster

After the IAM roles were ready, I opened the Amazon EMR service at:
https://console.aws.amazon.com/elasticmapreduce

I clicked on “Create cluster” and then “Go to advanced options” to be able to configure all the parameters precisely.

Cluster configuration:
• Release version: EMR 7.8.0
• Applications: Hadoop only
• Instance type: m5.xlarge
• Number of instances: 1 (Core node)
• Auto-terminate: disabled (so I could manually manage the cluster during testing)

Cluster name:
image-analytics-cluster-final

In the “Permissions” section, I assigned the previously created roles:
• EC2 instance role: EMR_EC2_DefaultRole
• EMR service role: EMR_DefaultRole

I clicked on “Create cluster” and after a short time, the cluster changed to “Waiting” status, indicating that it was ready for step addition.

3. Adding a Hadoop Streaming step

Once the cluster was ready, I opened its page and in the “Steps” section clicked on “Add step.”

Step parameters:
• Step type: Hadoop Streaming
• Name: object-label-count
• Mapper: s3://image-analytics-jhk-2025/hadoop/mapper.py
• Reducer: s3://image-analytics-jhk-2025/hadoop/reducer.py
• Input: s3://image-analytics-jhk-2025/hadoop/metadata_output.json
• Output: s3://image-analytics-jhk-2025/hadoop/output/

By clicking “Add,” the step was added to the list and automatically started.

Monitoring Step Execution and Resolving CRLF/LF Issue

After I added the Hadoop Streaming step to the cluster image-analytics-cluster-final, the first execution attempt was unsuccessful — the step ended with the status “Failed.”
By reviewing the logs through the “View logs” option in the EMR console, I noticed that the cluster was unable to execute the mapper.py and reducer.py scripts. The error message indicated a problem with the file format.

After further investigation, I realized that the issue was related to line endings — since I work on a Windows operating system, my .py files were saved with CRLF line endings (which is typical for Windows), while EMR, being a Linux-based environment, requires LF format.

I resolved the issue in the following way:

1. I opened both files (mapper.py and reducer.py) in Visual Studio Code
2. I clicked on the CRLF indicator in the bottom-right corner of the editor
3. I selected the option “Convert to LF”
4. I saved the files (Ctrl + S)
5. I uploaded them again to S3 in the same hadoop/ folder (overwriting the old versions)

I then created a new step with a different name: object-count.

Step parameters:
• Step type: Hadoop Streaming
• Name: object-count
• Mapper: s3://image-analytics-jhk-2025/hadoop/mapper.py
• Reducer: s3://image-analytics-jhk-2025/hadoop/reducer.py
• Input: s3://image-analytics-jhk-2025/hadoop/metadata_output.json
• Output: s3://image-analytics-jhk-2025/hadoop/output/

After a few minutes of execution, the status of the Hadoop step changed to “Completed,” indicating that the metadata processing on the EMR cluster was successfully completed. I accessed my S3 bucket and in the folder hadoop/output/ I found three output files:

part-00000
part-00001
part-00002

These files represent parallel output segments, as Hadoop executes the processing in a distributed way and divides the result into multiple segments depending on the number of tasks or execution units involved in the job.

After that, I downloaded all the part-* files from the S3 bucket to my local machine and placed them in the emr/ folder inside the main project directory image-pipeline-project/, in order to keep all EMR-related files organized in one place.

I merged the files directly from the terminal in Visual Studio Code using the following command:

type part-00000 part-00001 part-00002 > final_output.txt

This command merged all output segments into a single consolidated file final_output.txt, which contains the aggregated results — that is, a list of all detected objects and their total frequency in the analyzed image set.

To quickly get an insight into which objects were most frequently detected, I wrote a helper script called analyze_output.py. The script uses the collections.Counter library to:
• read the contents of final_output.txt,
• count the total occurrences of all detected labels,
• and display the Top 10 most frequently present objects.

This analysis served as a local tool for quickly checking the accuracy and understanding the distribution of objects before moving on to indexing and visualizing the results.

For example, the analysis output looked like this:

person 13788
car 1817
chair 1021
bottle 886
cup 863
bowl 654
umbrella 513
book 484
donut 449
horse 433

The file final_output.txt thus becomes the final result of the reducer phase and serves as the basis for the next step — indexing and visualizing the data using Elasticsearch and Kibana.

To simplify the development process and avoid complications related to manual installation and configuration, I decided to run Elasticsearch and Kibana locally using Docker Compose. Within the project, I created a dedicated directory docker/, and inside it, a file docker-compose.elasticsearch.yml containing the configuration for both services.

I started the services with the command:

docker-compose -f docker/docker-compose.elasticsearch.yml up -d

After a few seconds, Elasticsearch became available at http://localhost:9200, while Kibana was accessible via http://localhost:5601.

Launching Elasticsearch and Kibana with Docker Compose

To enable the ingestion of metadata from the previous phase into Elasticsearch, it was necessary to adjust the format of the metadata_output.json file. Since this file was originally structured as a single list of objects, I converted it to the JSON Lines format, where each record is placed on a separate line – which is the optimal format for line-by-line data ingestion.

For this purpose, I used a simple Python script that generated the file metadata_lines.json, which contains all the information about the analyzed images – including the image path, dominant colors, and the list of detected objects.

After that, I created a script named index_to_elasticsearch.py, whose purpose was to read metadata_lines.json, add a timestamp to each record, and then index them into a new Elasticsearch index named image_metadata.

I ran the script from the activated virtual environment using the following command:

python index_to_elasticsearch.py

After running the script:

- the previous index was deleted (if it existed),
- a new index named image_metadata was created,

- all documents (10,000 of them) were successfully indexed into Elasticsearch and were ready for visualization via Kibana.

3. Visualization of results in Kibana

After all data was successfully indexed in Elasticsearch, I opened Kibana at the address http://localhost:5601 to perform the configuration and check the display of results.

It is important to note that Kibana may take a bit longer to load on the first run – in my case, I had to wait approximately 60 seconds until the interface became fully available.

Within the Kibana interface, I followed these steps:

1. I navigated to the section: Management → Stack Management → Index Patterns, where I created a new index pattern using the identifier image_metadata*.
2. During creation, I confirmed the timestamp field as the time axis, if prompted.
3. After that, I switched to the Discover tab to check whether the documents were correctly indexed – the results appeared without any issues, confirming that the indexing was successful and that Kibana was properly connected to the Elasticsearch service.

Creating the visualization – most frequently detected objects

To obtain an overview of the most frequently detected objects in the analyzed images, I created a vertical bar chart using the field recognized_objects.label.keyword as the basis for aggregation.

Through the Kibana interface, I followed these steps:

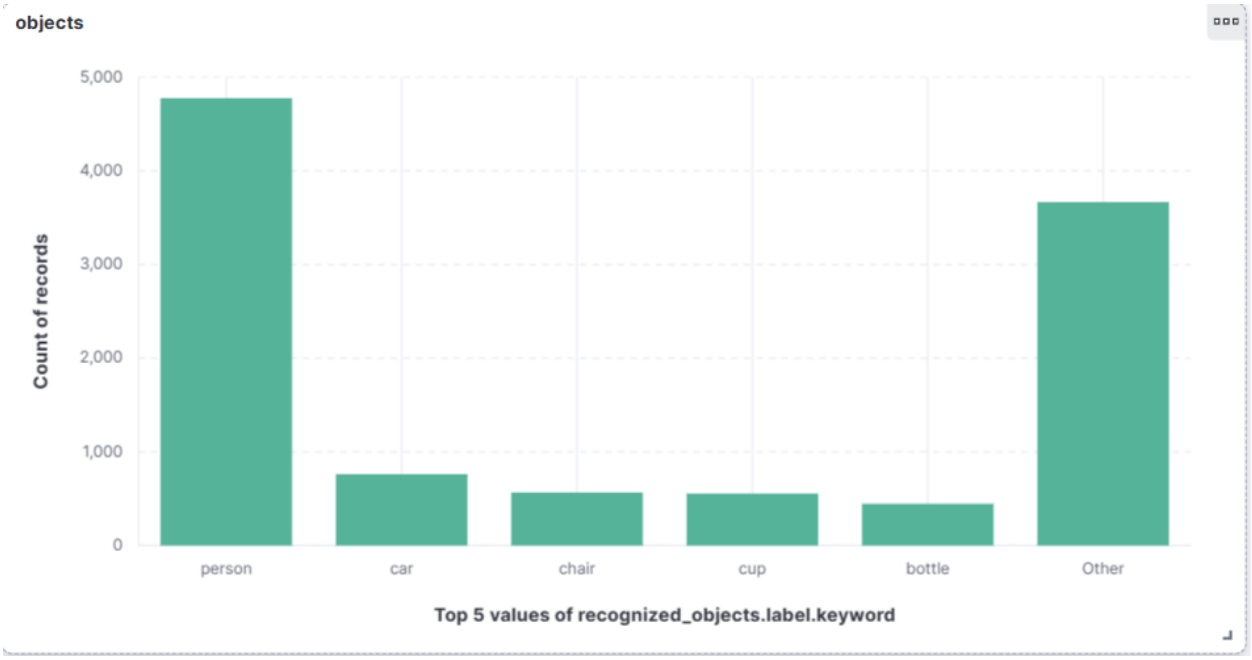
- Opened the section: Visualize → Create visualization → Vertical bar chart
- Selected the index: image_metadata
- Set the aggregation to: Count
- Defined the Bucket: Terms → recognized_objects.label.keyword

I did not manually limit the number of displayed values, so Kibana automatically displayed the five most frequently detected objects while grouping all others into the category "Other."

The chart clearly showed that the most common objects were:

person
car
chair
cup
bottle

with an additional "Other" group that contains all other less frequently detected objects.



Conclusion

Through this project, I went through all the key phases of building a real-time pipeline for image processing and analysis — from local data preparation, streaming via Kafka, image analysis using the YOLO model, to distributed result processing on the Amazon EMR cluster and their visualization in Kibana.

This was my first experience with the AWS environment, and I must emphasize that creating the S3 bucket and configuring the EMR cluster was one of the biggest challenges during the project. In order to overcome these obstacles, I spent a lot of time researching documentation, tutorials, and forums, which helped me find suitable solutions and understand how AWS functions in the context of large-scale data processing.

Each component in the system had its specific role, and only through their interconnection did the pipeline function as a whole. Among the challenges that stood out, I would particularly mention the issue with the CRLF line ending format, which caused an error during the execution of the MapReduce scripts — but I was able to identify and resolve it by adjusting the settings in Visual Studio Code.

The greatest value of this project for me was gaining a comprehensive understanding of data flow throughout the entire system — from the moment an image enters the pipeline to the moment its content is analyzed, aggregated, and visualized. The project gave me confidence that I can work with distributed systems, overcome technical challenges, and apply what I have learned in future, even more complex tasks.

Project Structure and File Organization

This project is organized into a clear and modular folder structure, with each file and directory serving a specific purpose in the pipeline:

```
image-pipeline-project/
├── .venv/                                ← Python virtual environment
├── docker/
│   └── docker-compose.yml                ← Kafka + Zookeeper
│   └── docker-compose.elasticsearch.yml  ← Elasticsearch + Kibana
├── elasticsearch/
│   └── index_to_elasticsearch.py         ← Indexes JSONL metadata
├── emr/
│   ├── mapper.py                         ← Extracts object labels
│   ├── reducer.py                        ← Counts label frequencies
│   ├── part-00000                        ← EMR partial output
│   ├── part-00001
│   ├── part-00002
│   └── final_output.txt                  ← Merged and aggregated result
├── images_subset/                        ← 10,000 selected Open Images
├── kafka/
│   ├── producer.py                       ← Streams image paths to Kafka
│   └── consumer.py                       ← YOLOv4-tiny detection & metadata extraction
├── output/
```

```
│   ├── metadata_output.json          ← YOLO detection results
│   └── metadata_lines.json           ← Converted JSONL version
├── utils/
│   ├── copy_images.py                ← Selects 10,000 random images
│   ├── analyze_output.py             ← Analyzes EMR results
│   └── convert_to_jsonlines.py       ← Converts JSON to JSONL
├── yolov4/
│   ├── yolov4-tiny.cfg               ← Model configuration
│   └── yolov4-tiny.weights           ← Pre-trained weights
└── README.md                         ← Full project documentation
└── Report.pdf
```

Each folder is self-contained, and filenames are chosen to be intuitive and meaningful — making it easy to navigate, reuse, and extend the project.