COMP0239 CW Report

Ruibo Zhang - 23073826

April 28, 2024

1 Data Analysis with BLIP Model

In this coursework, We applied the Bootstrapping Language-Image Pre-training (BLIP) model from Hugging Face for image captioning tasks [1]. As illustrated in Figure 1, the process begins when a user uploads an image to our server. The BLIP model then generates a relevant caption. We've implemented a distributed system to utilize this model for caption generation, which we've tested with a dataset of 150,000 images to ensure reliability and consistency.

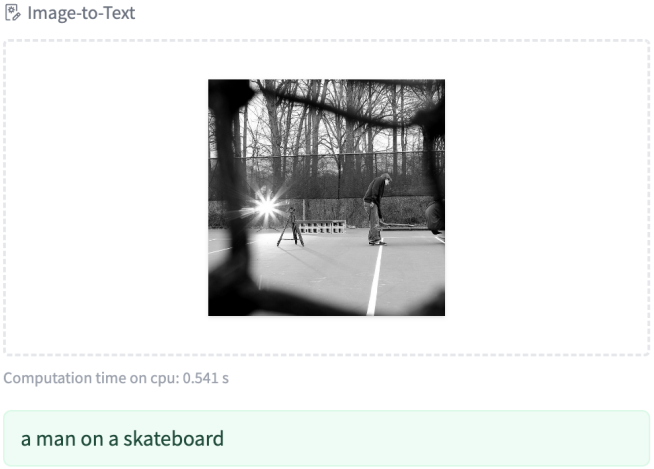


Figure 1: Caption Generation Example. The BLIP model provides a descriptive caption for the uploaded image.

2 Architecture and Tools

2.1 Introduction

We present the high-level architecture of our system, detailed with a UML diagram to depict component interactions. The section elaborates on the hardware and software used to ensure clear understanding.

Access the GitHub Repository for complete setup and execution instructions: https://github.com/RuiboZhang1/COMP0239-CW.

2.2 Architecture Overview

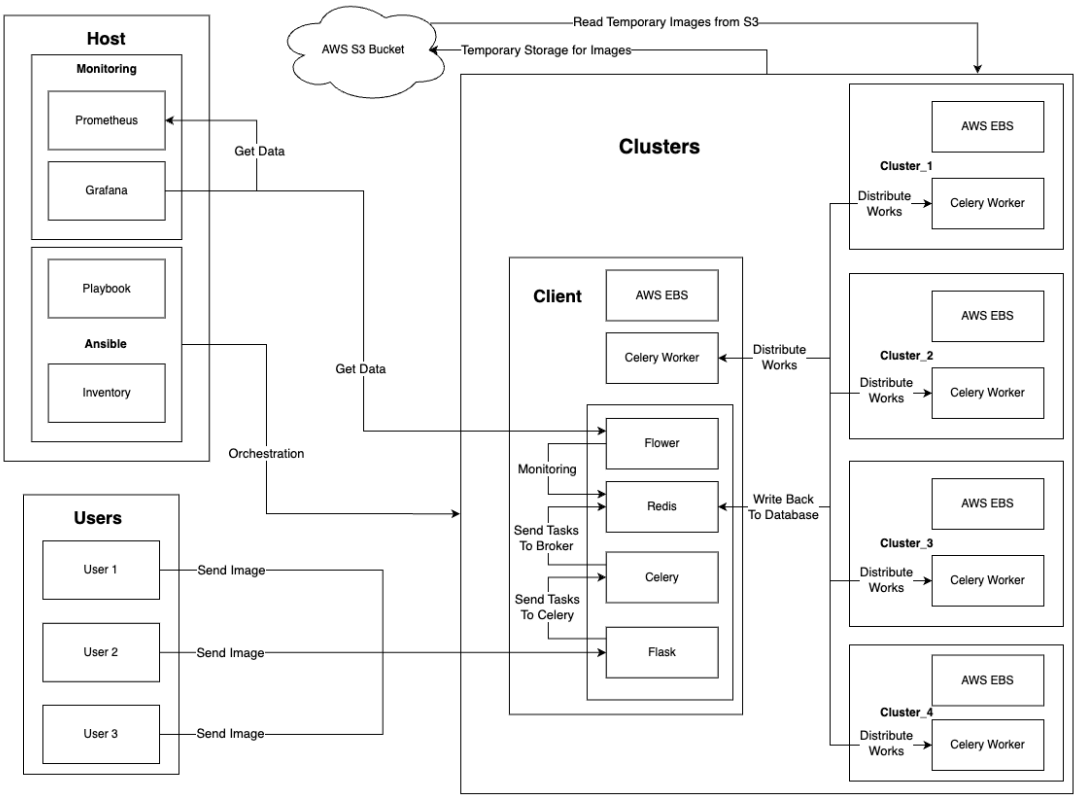


Figure 2: Architecture of the Distributed Data Analysis Service.

The UML diagram in Figure 2 represents our service's architecture, comprising four primary components:

- **Users**: Represent the end-users who upload images for caption generation via the front-end server.
- **Host**: Serves as the system coordinator with limited computational capabilities, responsible for:
  - **Monitoring**: Utilizes Prometheus and Grafana to monitor system load and tasks status across machines.
  - **Cluster Orchestration**: Employs Ansible to manage and execute tasks across the cluster efficiently.
- **Clusters**: Contains five machines, each utilizing AWS Elastic Block Store (EBS) for storage and initializes as a Celery worker to process image captioning tasks. The clusters operate as follows:
  - **Client** - Equipped with 2 CPU cores, 4 threads, and 8GB RAM, this node is the most capable within the cluster and performs several key functions:
    - **Front-end Server**: Processes incoming image files or URLs from users, checks for duplicates via MD5 checksums, and interfaces with the Redis database.

    - **Celery**: Manages the task queue when the front-end server is operational, interfacing with the Redis broker.
    - **Redis**: As the task broker and database, Redis distributes tasks to available Celery workers and stores the processed results.
    - **Flower**: Monitors the status of Celery workers and tasks, facilitating metric exportation for visualization with Prometheus and Grafana.
  - **Cluster_{1, 2, 3, 4}** - Each node has 1 CPU core, 2 threads, and 4GB RAM. They run Celery workers that execute distributed tasks.

- **AWS S3 bucket**: The S3 bucket is a shared file storage system that can be accessed by all nodes. In this coursework, it acts as the temporary storage for images during processing. Images are uploaded by the Flask server and accessed by Celery workers as needed.

2.2.1 Software

Monitoring System Flower, Prometheus, and Grafana are integral tools used for monitoring and managing data analysis pipelines.

- **Flower** - An open-source web application for monitoring and managing Celery clusters. It provides real-time information about the status of Celery workers and tasks [2].

- **Prometheus** - As an open-source monitoring system, it collects and stores metrics as time-series data by scraping HTTP endpoints, enabling the observation of system and process performance[3].

- **Grafana** - An analytics platform that visualizes time-series data with interactive graphs and alerts, Grafana connects with data sources like Prometheus to facilitate the querying and understanding of metrics across systems[4].

Together, these applications form an integrated monitoring system, which gives comprehensive information on the cluster status and tasks status:

- **Flower** offers an operational view of Celery tasks, tracking their execution and current status.

- **Prometheus** is set up to gather Flower's metrics to assess the performance of task queues and the health of workers. It also monitors clusters' hardware performance through the Prometheus Node Exporter.

- **Grafana** leverages Prometheus data to build comprehensive dashboards, showcasing key performance indicators such as queue lengths, task durations, and resource utilization.

Orchestration System Orchestration automates the coordination, configuration, and management of complex computing environments, systems, and applications. It streamlines tasks such as server deployment, networking, and service integration. This multi-faceted process necessitates a robust orchestration system for seamless operation across various platforms.

Key advantages of orchestration include accelerated deployment, reduced manual intervention, fewer errors through consistent automation, and resource utilization optimization for cost-efficiency.

Ansible - An open-source, command-line IT automation software application. It facilitates a broad range of IT tasks, including application deployment, system updates, network setup, and operations [5]. Ansible offers distinct advantages compared to other tools like Chef, Salt, Puppet, etc.:

1. Agentless Architecture - Ansible does not require an agent to be installed on the target nodes. Users can define the Ansible inventory file in the host node, which defines the hosts and groups of hosts. It works by connecting directly via SSH and executing small "Ansible modules" on target nodes, simplifying setup and maintenance.

2. Simplicity and Ease of Use - Ansible utilises 'playbooks' to define the desired state and actions for target machines. These playbooks, written in human-readable YAML file, are straightforward to the author. They are also idempotent, ensuring that their execution on a system, already in the desired state, maintains that state without adverse effects.

Front-end Server and Distributed Task Queue Our server accepts user inputs and delegates tasks across clusters. We discuss our software choices—Flask for the server and front end, Celery for task management, and Redis for the task queue and database.

- **Flask** - A lightweight and flexible Python web framework that provides tools and features to build web applications effortlessly [6]. Compared to heavier frameworks like Django, Flask has the following pros and cons [7]:
  - **Pros**
    - Lightweight and Minimalistic: Flask is renowned for its simplicity and minimalism. This makes it an excellent choice for small to medium-sized projects, rapid development, and microservices.
    - Flexibility: It allows developers to tailor their development environment by choosing their own tools, libraries, and extensions to shape their web application precisely as envisioned.
    - Large Community and Resources: The large community behind Flask offers extensive support through a plethora of resources, tutorials, and extensions.
  - **Cons**
    - Feature-Richness: Flask's minimalism is a double-edged sword; the lack of built-in features means that it falls behind more equipped frameworks like Django, which can be a disadvantage for some projects.
    - Scalability Issues: When scaling to more complex applications, Flask may require more manual intervention and configuration to manage effectively.
    - Security and Authentication: While Flask provides the necessary tools to build secure applications, it demands a significant responsibility from developers to implement security features correctly.

  Given the scale and scope of our project, starting with Flask offers a suitable balance of control and ease of development.

- **Ray** - An open-source unified framework for scaling AI and Python applications like machine learning [8]. It provides the compute layer for parallel processing. Ray automatically scales compute resources as needed, allowing us to focus on the code instead of managing servers.

  This framework is user-friendly and we initially decided to use it for parallel processing. However, due to unclear network problems, the cluster nodes were soon offline after connecting to the ray server, which was running on the client node. More experiments should be carried out to investigate the reason for the problem.

- **Celery** - Alternatively, we choose Celery as the distributed task queue. It allows for asynchronous task execution and is designed to handle distributed processing by communicating via message passing [9]. This makes Celery an excellent tool for offloading lengthy tasks from the main thread of a Flask application, ensuring that the application remains responsive to user interaction. There are a few reasons for it to be used in this project:

  - Asynchronous Processing: It enables background processing of tasks, allowing for non-blocking operation of the main server thread.
  - Distributed System Support: With the capability to distribute tasks across multiple workers and servers, Celery supports horizontal scaling and effective workload distribution.
  - Broker and Result Backend Compatibility: Celery interfaces well with various message brokers like RabbitMQ, Redis, and Amazon SQS, and provides multiple result backends such as Redis, RabbitMQ, PostgreSQL, and MySQL.
  - Built-in Task Management: It includes mechanisms for tracking task states and retrieving results, which is especially advantageous for long-running processes.

- **Redis** - An in-memory data structure store, commonly used as a database, cache, and message broker [10]. Its key-value storage paradigm is optimized for high performance, making Redis an excellent choice for situations that require fast read and write operations. The easy-to-use and multi-functionalities of Redis make it a suitable tool for our project, starting the Redis server and then serving as both a message broker and database.

  Since we are not doing a data-intensive task, RabbitMQ is more suitable to be the message broker, which is optimized for reliability and complex routing, not for speed of data access like Redis. We are considering to upgrade the pipeline in the future.

Combining Flask, Celery, and Redis provides a powerful ecosystem for our web application's needs, particularly for tasks that require extensive processing in the background. This setup maintains a responsive front-end interface, ensuring users experience minimal latency during operation.

Storage The base storage capacity for each machine in our pipeline is 10 GB, which we found inadequate for our requirements. To address this, we integrated two Amazon storage services:

- **Amazon Elastic Block Storage (Amazon EBS)** - Amazon EBS offers block-level storage volumes for Amazon EC2 instances, functioning like raw, unformatted block devices [11]. For our project, we've utilized EBS to augment storage on each cluster machine. During the setup, we specifically use EBS to create temporary directories for Python library installations, such as those required for PyTorch, thus avoiding the limitations of the default storage. Post-installation, these temporary directories are deleted, optimizing our storage usage.

- **Amazon Simple Storage Service (Amazon S3)** - Amazon S3 is an object storage service designed for storing and retrieving data from anywhere on the web [12]. It allows multiple EC2 instances to access the storage, making it ideal for data sharing among analysts. In our setup, S3 buckets are designated as temporary repositories for images uploaded by users. Once an image is uploaded to the host, it is temporarily stored in S3 until a Celery worker is ready to process it. This strategy not only facilitates efficient data sharing but also reduces the computation and communication overhead on the host node by decoupling storage from processing tasks. After image processing, the data is cleared from S3, maintaining a clean state.

3 Main Pipeline

3.1 Introduction

This section details the dataset and the method used to test our distributed data analysis system over a 24-hour period.

3.2 Dataset

We utilized the MS COCO(Microsoft Common Objects in Context) 2017 test and unlabeled dataset [13], extracting image URLs from the annotation files for over 150,000 images. The URLs were saved to 'coco_image_urls.txt'. Accompanying Python code and the URL text file are available on GitHub. Users must install the 'cocoapi' package prior to running the code. The text file is intended for direct use in testing.

3.3 Prerequisite

It is assumed that all systems are set up according to the README on GitHub. Subsequent sections will outline the pipeline operation and monitoring metrics.

3.4 User Side

To test the system, 'test_pipeline.py' inside the pipelines directory is executed, generating a CSV file of captions called 'captions.csv'. This script simulates a user, continuously sending image URLs to the Flask server. If an image is recognized as processed, the server immediately returns its caption. Otherwise, the server requests processing and polls for results every 5000 images to manage system load and memory.

3.5 System Side

Host - The Flask server has two endpoints for the user to interact with:

- **upload**: Accepts image file or URL. It computes the MD5 checksum for each image, checks for duplicates in Redis, and returns the caption to the user directly if it exists. Otherwise, if the input is an image file, the server uploads it to the S3 bucket with its name as the key and then calls the 'process_image' task. The Server will send the jobs to the Celery workers through the Redis broker. Similar operations for the image URL input, the server first fetches the image from the given URL and uploads it to the S3 bucket before further processing.

  After submitting the task to Celery, the server sends the task ID back to the user, which can be used for retrieving the result later.

- **task/{task_id}** - Retrieves captions for processed images using the task ID returned by the server. This endpoint informs users of the processing status or delivers the final caption.

Clusters

- **Redis** - Operates as both the message broker and database on the client node, which can handle intensive transactions. It stores key-value pairs, with image MD5 checksums serving as keys and associated captions as values. This facilitates efficient retrieval of captions for previously processed images, eliminating redundant processing through checksum matching.

- **Celery Worker** - Each cluster node functions as a Celery worker. They execute tasks distributed by the message broker. Two primary tasks are:

  - **fetch_and_process_image**: Retrieves images from URLs, uploads them to the S3 bucket, and initiates the captioning process by passing the S3 object key to the next task.
  - **process_image**: The main task for captions generation, utilizes additional decorators for efficient task execution. When first called, loads the model into memory to prevent repeated loads for subsequent tasks, conserving memory and processing time. Post-generation, it saves the results to the Redis database for result retrieval.

3.6 Overview of the pipeline

The overview of the pipeline operates through the subsequent steps:

1. **Sending Data**: Continuously send a set of 150,000 image URLs to the Flask server.

2. **Check Exist**: The server retrieves the image from the URL, calculates its MD5 checksum, checks the Redis database for this checksum, and if a match is found, returns the existing caption.

3. **Fetch Image**: For new images, the server uploads the image to the S3 bucket, using the image's name as the key for subsequent retrieval.

4. **Process Image**: The broker assigns the task to an available Celery worker, passing along the S3 key. The worker, on its first call, loads the model into memory to avoid redundant loading for future tasks. The worker then retrieves the image from S3, generates a caption, saves the MD5 checksum and caption to the database, deletes the image from S3, and sends the caption back.

5. **Retrieve Result**: After processing sets of 5,000 images, the server requests the results to manage the system load and writes the image name, task id, and caption into a CSV file.

3.7 Monitoring

We employ Flower, Prometheus, and Grafana to monitor the pipeline, as described in Section 2. Grafana dashboards display metrics on Celery worker status and hardware utilization on the cluster:

- **Celery Worker Status**: Indicates if a worker is active (1) or offline (0).
- **Number of Tasks Currently Executing at Worker**: Shows the count of concurrent tasks on each worker, including minimum, maximum, average, and the count at the current moment.
- **Average Task Runtime at Worker**: Presents the minimum, maximum, average, and current runtime for tasks on each worker.
- **Task Prefetch Time at Worker**: The time interval during which a Celery worker node fetches or reserves tasks from the message broker before actually executing them.
- **Number of tasks prefetched at worker**: The number of tasks each worker process will fetch and hold while it processes other tasks.
- **Task Success Ratio**: The ratio between the number of successful tasks and the number of total tasks processed.
- **Task Failure Ratio**: The ratio between the number of failed tasks and the number of total tasks processed.
- **Task Run on Workes**: Count the number of tasks processed by each worker.
- **CPU usage (%)**: Computes CPU utilization as the inverse of idle time.
- **RAM usage (%)**: Determined by the available memory ratio.
- **Load average (1/5 minutes)**: Records system load over one and five-minute intervals, aiming for averages below 4 for clients and 2 for clusters.
- **Default Disk Usage**: Monitors default disk utilization based on the available-to-used bytes ratio.
- **Network Throughput (Receive/Transmit - 15 minutes)**: Captures the average incoming and outgoing network traffic per second over 15-minute periods.

Visualizations of these metrics, alongside output analyses, will be detailed in Section 4.

4 Results

The initial configuration with the Flask server, Celery App, and Flower hosted on the host node encountered performance bottlenecks. Monitoring data revealed periods of extreme system load, as depicted in Figure 3, which led to pipeline stalls and failures. Consequently, we redistributed the Flask server, Celery App, and Flower across the client nodes to enhance stability and performance. System monitoring outcomes and the resulting caption outputs are detailed below.

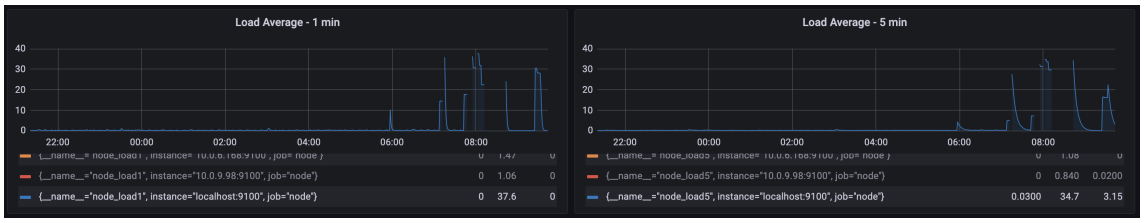


Figure 3: Host node experienced an extremely high load average, which resulted in the pipeline failure.

4.1 Grafana

Grafana visualizations from Figures 4 to 8 depict our system's performance based on the metrics previously established.

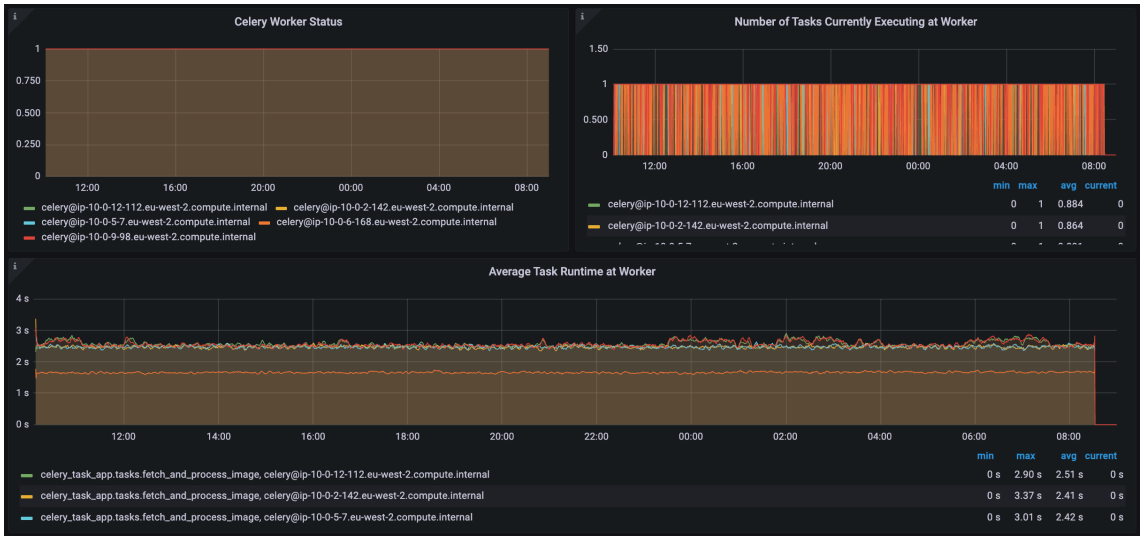


Figure 4: Displays the Celery Worker Status, the Count of Tasks in Execution at each worker, and the Average Task Runtime at the worker over the testing period.

Figure 4 presents three key metrics. The top left panel confirms that all five cluster nodes remained operational throughout the testing phase. The top right panel shows task execution, where the concurrency limit was set to one task per worker at any given time. As intended, the graph indicates that each worker processed only one task at a time. In terms of performance, while the client node, as the most powerful machine, had the fastest average runtime for fetching and generating captions, other cluster nodes required approximately 2.5 seconds for similar tasks.

Figure 5: The task prefetch time and number of tasks prefetched on each worker

Figure 5 depicts task prefetch time and the number of tasks prefetched at workers. The top panel indicates the interval between task assignment and execution, with a maximum wait of around 12 seconds and an average of approximately 3.5 seconds. The bottom panel confirms that up to four tasks were prefetched in alignment with the Celery worker's initial configuration.

Figure 6: The task success/fail ratio, number of tasks processed by each worker, and CPU usage on each machine.

Figure 6 showcases the task success ratio, failure ratio, tasks processed by each worker, and CPU usage on each machine. The top panels illustrate a 100% task success rate, indicating flawless pipeline performance throughout the testing. The bottom right panel demonstrates an even distribution of workload among workers, with each processing about 30,000 images. Meanwhile, the bottom left panel reveals that CPU usage across all workers remained below 50% on average.

Figure 7: The load average(1/5 minutes), the default disk usage and RAM usage

Figure 7 presents load averages, default disk usage, and RAM consumption metrics. The top panels indicate that the 1-minute and 5-minute load averages remain within optimal thresholds, with client and cluster machines operating below 2 and 1, respectively. The bottom left panel shows stable disk usage, as expected, since images are not stored on local drives. On the bottom right, a moderate increase in RAM usage is observed, particularly for the client node, which correlates with Redis's in-memory data storage; as it processes more images, the memory occupied grows due to the storage of generated captions.

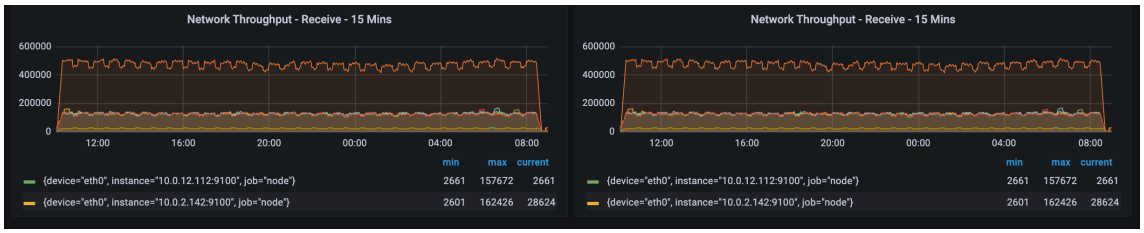
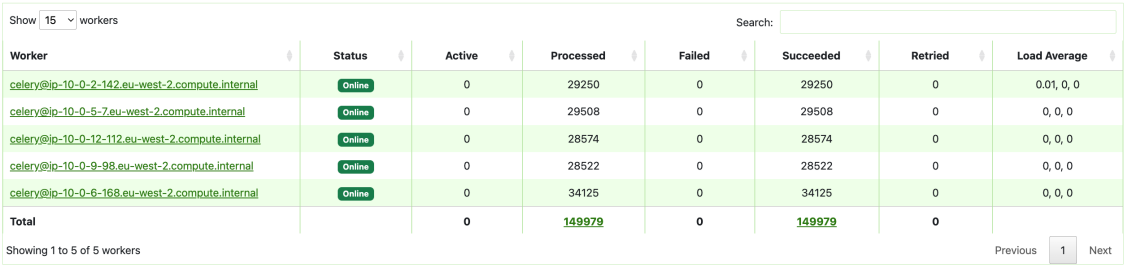


Figure 8: The average network throughput for incoming and outgoing traffic.

Figure 8 illustrates network throughput for both incoming and outgoing traffic. The graphs depict a consistent pattern of network activity, aligning with the batch processing of images. Network load peaks when new image URLs are received and drops following the batch submission, reflecting the data retrieval cycle from the Flask server.

4.2 Flower

The Flower dashboard offers insights into task distribution, noting a total count lower than the anticipated 150,000. This discrepancy likely stems from the system's efficiency in recognizing and eliminating the processing of duplicate images, as confirmed by the Redis database, thereby returning pre-stored captions directly and reducing the need for task submissions to Celery.



Show 15 workers Search:

Worker	Status	Active	Processed	Failed	Succeeded	Retried	Load Average
celery@ip-10-0-2-142.eu-west-2.compute.internal	Online	0	29250	0	29250	0	0.01, 0, 0
celery@ip-10-0-5-7.eu-west-2.compute.internal	Online	0	29508	0	29508	0	0, 0, 0
celery@ip-10-0-12-112.eu-west-2.compute.internal	Online	0	28574	0	28574	0	0, 0, 0
celery@ip-10-0-9-98.eu-west-2.compute.internal	Online	0	28522	0	28522	0	0, 0, 0
celery@ip-10-0-6-168.eu-west-2.compute.internal	Online	0	34125	0	34125	0	0, 0, 0
Total		**0**	**149979**	**0**	**149979**	**0**	

Showing 1 to 5 of 5 workers Previous 1 Next

Figure 9: The Flower interface provides a clear depiction of the active status of Celery workers and a summary of tasks processed.

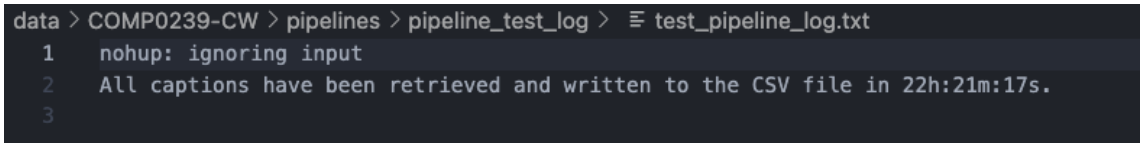
4.3 Output Files

The pipeline testing process generated three log files, capturing activities from the Flask server to the Celery Flower and including the test pipeline Python script. All logs are consolidated into the 'pipeline_test_log' directory.

It also generated a 'captions.csv' file which contains the image name, task ID and corresponding captions, as shown in Figure 10. Figure 11 demonstrated efficiency by processing all 150,000 image captions in less than 23 hours — three hours ahead of the expected timeline when we initially ran on the host node.

```
149989  000000493910,56ce936b-3098-4e1a-8388-d7c568716f81,a dog chasing a sheep in a field
149990  000000328674,30157ea0-546e-43bf-af5a-4b4332cb504a,a black metal fence
149991  000000456242,71bdaebd-a432-4458-a32e-15a2376c6b7a,a traffic light on a pole
149992  000000428545,a6d0de86-c235-49ee-95e4-d6ac695dca9e,a plane on the runway
149993  000000449165,a5e5eeb5-3583-4da1-937d-72bbf5f73a12,a clear blue sky
149994  000000241686,79bae0d5-d846-46ba-a40a-b6340f3f0c72,a plane on the ground
149995  000000209876,26c3f766-b415-403f-a08a-8ad262e3e369,a bird is standing on a bench near a lake
149996  000000396012,ecc2698c-1478-4dfa-af71-c48d1c724b8b,a pink bench
149997  000000166772,e9ee44ef-4402-4bb8-bed6-a8c15bd67e8f,a plane is on the runway at the airport
149998  000000468943,4a2a731b-67aa-4297-a6d1-93cde6f82a19,a street scene with a traffic light and a sign
149999  000000104987,4a7ed1a0-7b87-44a9-8e9e-7cae4cc5fdd6,a dog and a dog are standing on a bench
150000
```

Figure 10: The complete caption file consists of 150,000 entries.



```
data > COMP0239-CW > pipelines > pipeline_test_log > test_pipeline_log.txt
1   nohup: ignoring input
2   All captions have been retrieved and written to the CSV file in 22h:21m:17s.
3
```

Figure 11: The testing pipeline concluded successfully in 22 hours, 21 minutes, and 17 seconds, showcasing the system's efficacy and consistency.

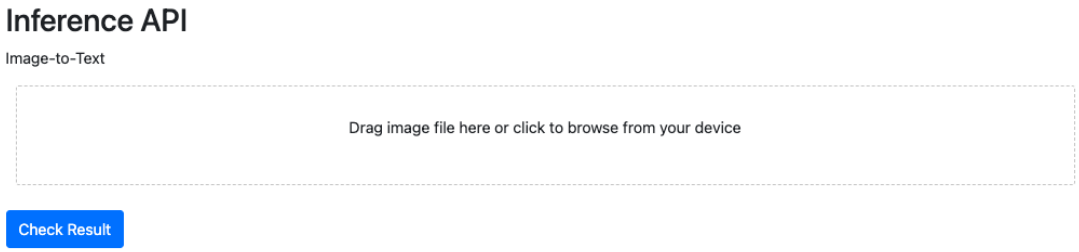
5 Frontend

A user-friendly front-end interface has been created to facilitate image uploads for caption generation. The interface is accessible at http://13.40.128.207:4506/, as shown in Figure 12a.

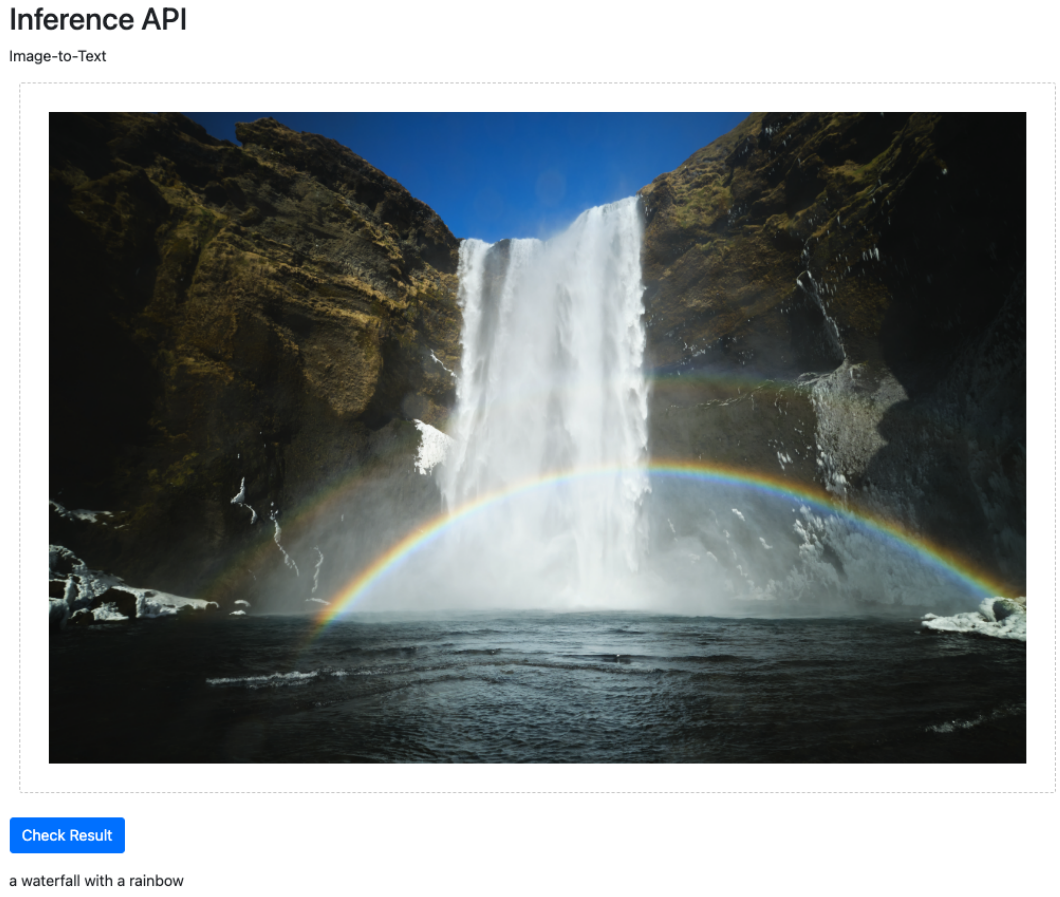
When a user uploads an image, the server checks whether it has been processed previously. If so, the message "Image has been processed. Click 'Check Result' to view the caption." is displayed, prompting the user to obtain the caption. By clicking the 'Check Result' button, the user can retrieve the caption for the image.

For new uploads, the interface will indicate "Image uploaded and awaiting processing. Click 'Check Result' to view the caption later." Users can then periodically click 'Check Result' to query the processing status from the server. Once the image processing is successful, the server will display the generated caption to the user upon the next 'Check Result' action, as shown in Figure 12b.

More Images are available for testing in the 'More Test Images' folder.



(a)



(b)

Figure 12: (a) The default landing page where users can upload images for caption generation. (b) Displays the user interface post-caption generation, with the caption visible beneath the uploaded image.

6 Conclusion

In this project, we successfully designed and implemented a distributed data analysis pipeline with the specific aim of generating captions for images. Our testing

procedures, which involved sequential input of tasks, have confirmed the pipeline's operational consistency and reliability. The results confirm that our system capably handles tasks in a sequential manner and completes them efficiently.

Throughout the testing phase, the system demonstrated a remarkable ability to process a large volume of data without significant delays or errors. This efficiency is a testament to the thoughtful architecture and the seamless integration of various components, including the Flask server, Celery workers, and Redis database, all of which have been instrumental in achieving our goals.

Nevertheless, we recognize that there is potential for further enhancements. To facilitate better deployment and distribution, we plan to dockerize the entire pipeline. Dockerization will not only streamline the setup process for end-users but also enhance the portability and scalability of our application.

Future updates will also focus on improving user interaction with the front-end interface. We envision integrating a history log feature, allowing users to review their previously uploaded images and their corresponding captions. This would not only improve the user experience by providing a record of past interactions but also reinforce transparency in data processing.

Furthermore, we are considering the development of more intuitive features for real-time monitoring of the processing status. This could include progress indicators or notifications that inform users when their task is queued, processing, or completed. By providing users with visibility into the task flow, we aim to build a more user-centric platform that empowers them with information and control.

References

[1] https://huggingface.co/Salesforce/blip-image-captioning-base

[2] https://flower.readthedocs.io/en/latest/

[3] https://prometheus.io/docs/introduction/overview/

[4] https://grafana.com/docs/grafana/latest/introduction/

[5] https://www.ansible.com/overview/how-ansible-works

[6] https://flask.palletsprojects.com/en/3.0.x/

[7] https://www.linkedin.com/pulse/exploring-pros-cons-flask-web-development-2023-deepak-ranjan-4ypnf/

[8] https://docs.ray.io/en/latest/ray-overview/index.html

[9] https://docs.celeryq.dev/en/stable/getting-started/introduction.html

[10] https://redis.io/docs/latest/get-started/

[11] https://aws.amazon.com/ebs/

[12] https://aws.amazon.com/s3/

[13] https://cocodataset.org/#home