GROUNDZERO
DEVELOPMENT

Technical Documentation For SimiLabs

Table of Contents

Table of Contents

Introduction

This document processing and analysis system addresses the escalating societal need for data-driven insights, particularly within the education sector. As demands for informed decision-making, improved educational outcomes, and innovation continue to rise, universities face the challenge of extracting meaningful information from documents. Designed to meet these demands, the system offers a scalable and adaptable solution, with a modular design that allows for easy integration and removal of functionalities, ensuring flexibility and relevance as new analytical tasks emerge. Ultimately, the system's aim is to efficiently handle large volumes of documents while enabling users to customise analysis features based on their specific needs, thereby enhancing decision-making processes, and driving innovation within the education sector.

This technical documentation provides a comprehensive guide to understanding and working with the system's architecture, components, and functionalities. It is designed to help developers, system administrators, and stakeholders in implementing, maintaining, and extending the system efficiently.

Technology Stack

This section looks at all the technologies that were used in the project, with a description of what they are used for, why they are necessary for the project and the reason the specific technology was chosen.

- **RabbitMQ:**
    - Description: RabbitMQ is used as the message broker for asynchronous communication between microservices. Asynchronous communication means that different parts of a system can send and receive messages without needing to wait for each other to be available or complete their tasks.
    - Why necessary: It ensures that messages are delivered even if a service is down temporarily, it decouples services and it can distribute the workload across multiple instances of a service.
    - Reason for selection: It supports complex message routing with minimal overhead and is easy to use. Unlike Kafka, RabbitMQ is designed for simpler use cases that don't require high-throughput distributed streaming.

- **Python Flask:**
    - <u>Description:</u> Flask is a lightweight web framework for Python, designed to help developers create web applications quickly and with minimal setup. Unlike larger frameworks, Flask provides only the essential tools, allowing developers to add features as needed, which makes it very flexible.
    - <u>Why necessary:</u> It enables the quick set-up of services and integrates well with other system components such as the storage solutions. The programs are also easy to containerize for deployment.
    - <u>Reason for selection:</u> The simplicity and flexibility of Flask make it popular for applications that need to be fast and modular, like APIs and microservices. It also has a strong community and ecosystem, with many libraries available for things like authentication, database connections, and caching, making it possible to build applications while keeping the core framework lightweight.

- **MinIO:**
    - <u>Description</u>: MinIO is an open-source object storage solution designed to store and manage large amounts of unstructured data, such as files, images, videos, and backups. MinIO is optimised for high-performance storage, with a lightweight design that supports rapid data access, making it ideal for big data analytics.
    - <u>Why necessary:</u> It offers scalable storage, as it supports distributed deployment across multiple servers and because MinIO is self-hosted, it gives full control over your data.
    - <u>Reason for selection:</u> MinIO is open-source, so we can build data-intensive applications that require fast and reliable storage without the need for proprietary cloud services. It's widely used in industries that handle large datasets and need flexible, performant storage solutions.

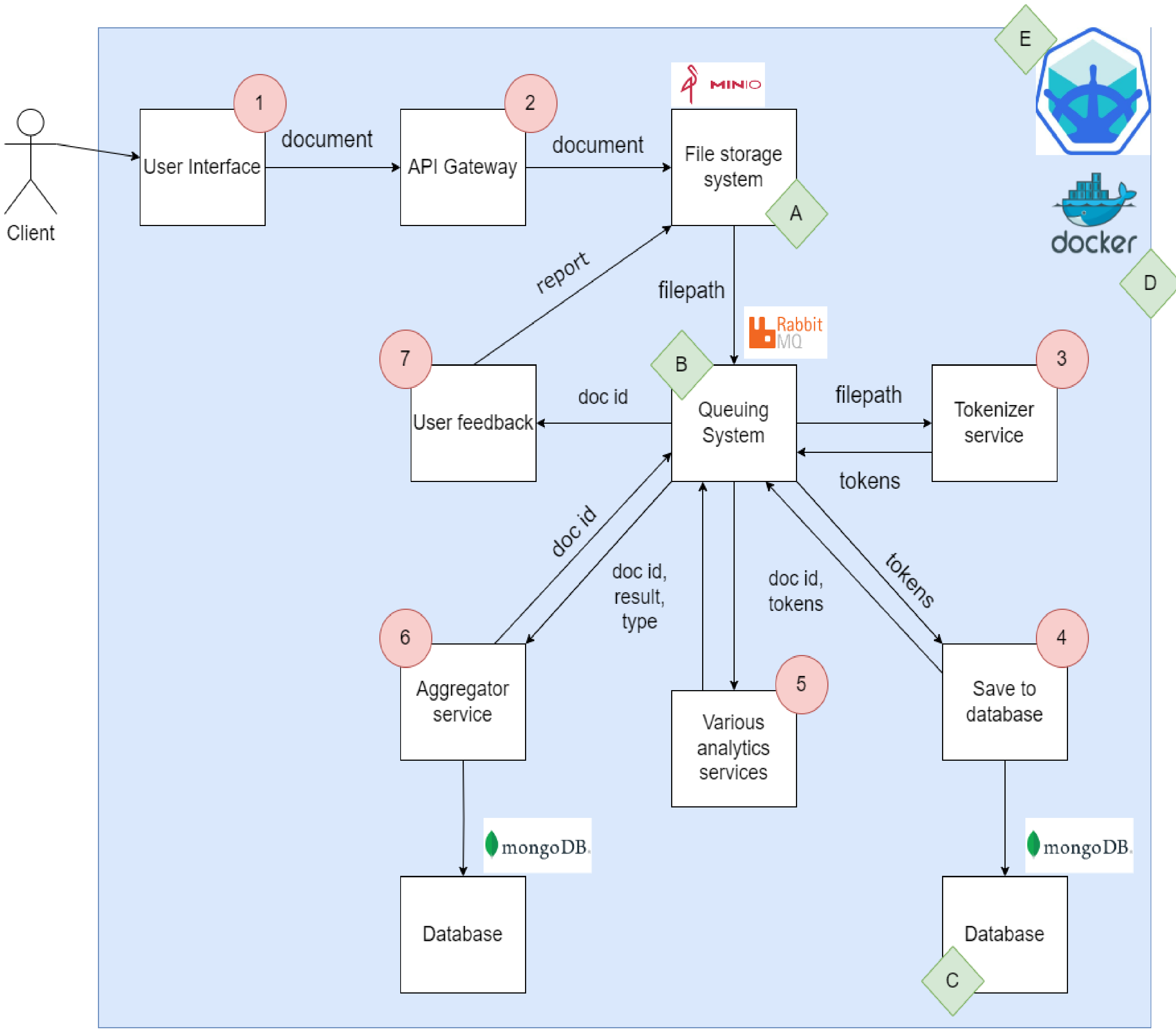
- **MongoDB:**
    - <u>Description:</u> MongoDB is used as the primary database to store and manage data in a flexible, document-oriented format. MongoDB allows for data to be stored in JSON-like structures, making it suitable for applications with varied data types and evolving schema requirements.
    - <u>Why necessary:</u> The system requires a flexible database capable of handling varied results and metadata.
    - <u>Reason for selection:</u> It is easily scalable and optimised for read/write performance.

- **Docker:**
    - Description: Docker is used to containerize applications, enabling them to run consistently across different environments. Docker packages each application and its dependencies into a container, ensuring that it can operate regardless of the underlying infrastructure.
    - Why necessary: Docker ensures each service runs consistently across different environments, allowing each to operate in its own container, which simplifies tracing and debugging. Additionally, Docker containers are lightweight, making them resource-efficient and scalable.
    - Reason for selection: Docker offers a very developed ecosystem with extensive tooling and support for a wide range of integrations, including orchestration with Kubernetes. Its community provides a repository of resources, comprehensive documentation, and prebuilt images, which accelerate development and simplify troubleshooting.

- **Minikube:**
    - Description: Kubernetes is an open-source container orchestration platform designed to automate the deployment, scaling, and management of containerized applications. Minikube is a tool that allows developers to run Kubernetes clusters locally to experiment and test without needing a multi-node cluster or cloud environment, simplifying the process of setting up a development environment for containerized applications.
    - Why necessary: Kubernetes ensures service instances run reliably, automatically restarting or replacing them if they fail. It also provides built-in service discovery for seamless microservice communication, as well as support for rolling updates. Minikube simplifies deployment by creating a local Kubernetes environment.
    - Reason for selection: Using Minikube allows for quick iteration cycles during development, as it eliminates the need for cloud infrastructure for testing. It also ensures compatibility with Kubernetes, the de-facto standard for container orchestration. Minikube is well-suited for microservice projects like this, where individual services, such as RabbitMQ, MongoDB, and MinIO, need to be deployed and tested as part of an integrated environment.

System Architecture

The system follows a microservice architecture, where each task is a separate service. The document is uploaded by a user (1), after which the gateway sends (2) it to be stored in a MinIO bucket (A). After a new document is uploaded it is sent (B) to the tokenizer service (4). This service extracts the text from the document and separates the words (tokens) after which it is sent to the service that saves it to the database (4), as well as to all the analytics services (5). The output of these services are then sent to the aggregator service that puts all the results related to a document together (6) and then adds this to the existing database record (C). Finally, the results, along with the file name, are retrieved to generate a report via the user feedback service (7), and this report is stored in a feedback bucket in MinIO. Each service (1-7) is deployed as a Docker container (D) within its own Kubernetes pod (E).



1. Components

This section provides setup instructions for components A-E, along with essential concepts of the component and common elements relevant to all services (1-7). There is one deployment.yaml file with all the component's deployment files in it, this file is explained in the "Deployment" Section of the document. When deploying these components, they have to be set up from E to A.

A. MinIO

The MinIO deployment (YAML) contains the following components:

1. Headless Service (minio-headless-svc.yaml)

   The headless service exposes the MinIO instance inside the Minikube cluster without a cluster IP, which allows other services to communicate directly with the MinIO pod.

   Key features:

   - Exposes MinIO on port 9000 within the Kubernetes cluster.
   - Allows for direct pod-to-pod communication.

2. StatefulSet (minio-statefulset.yaml)

   MinIO's StatefulSet ensures that MinIO has a stable identity and persistent storage even after pod restarts. It makes use of a PersistentVolume defined by the relevant StorageClass.

   Key features:

   - Persistent identity and storage even after pod restarts.
   - Obtains 10Gi of persistent from the StorageClass (minio-dynamic-storage)
   - Only a single replica is created (more can be added as needed)
   - Uses MinIO Headless Service for direct access to pods, and uses Kubernetes secrets (minio-creds) for secure storage of access and secret keys.

3. StorageClass (minio-storageclass.yaml)

   The StorageClass defines how persistent storage is provisioned for the MinIO StatefulSet within the Minikube environment.

   Key features:

   - Uses the Minikube hostpath provisioner, which stores data on the local Minikube node.
   - The *reclaimPolicy* is set to "retain", which allows for data recovery in the event a volume claim is deleted.
   - Volume expansion is enabled, allowing for an increase in storage capacity when needed.

4. MinIO Secrets (minio-creds.yaml)

   Creating a Kubernetes Secrets file is crucial for MinIO to operate securely. Instructions on how to accomplish this can be found under the “Deployment” section of this document.

B. RabbitMQ

The RabbitMQ deployment (YAML) contains these components:

1. ConfigMap

   This stores custom configuration for RabbitMQ, enabling specific plugins and setting configurations. It enables:

   - *rabbitmq-federation:* this plugin allows RabbitMQ to connect with other RabbitMQ clusters (this is not done now, but could be used later if there are distributed workloads or failover).
   - *rabbitmq_management:* this plugin provides a web-based management UI, allowing you to monitor and manage RabbitMQ through a browser.
     - This management UI can be accessed as follows:
       - Server side: kubectl port-forward rabbitmq-0 8080:15672
       - Your computer: ssh -L 8080:127.0.0.1:8080 cobitdev@196.252.135.195
       - http://127.0.0.1:8080/#/
   - *rabbitmq.conf:* this configuration ensures that remote users can’t connect to the UI.

2. StatefulSet

   This deploys RabbitMQ in a StatefulSet, which provides stable network identities, ordered scaling, and storage management.

   - *Service name:* this is used to ensure a network identity, which means the pod gets an identity that does not change even when the pod is restarted.
   - *Replica count:* defines a single instance of RabbitMQ in this configuration.
   - *initContainers:* runs a container to copy the ConfigMap files into specific RabbitMQ directories before the main RabbitMQ container starts, ensuring the configuration is correctly set up.
   - *volumeMounts and volumes:* each RabbitMQ pod will have its own dedicated storage that persists even if the pod restarts.

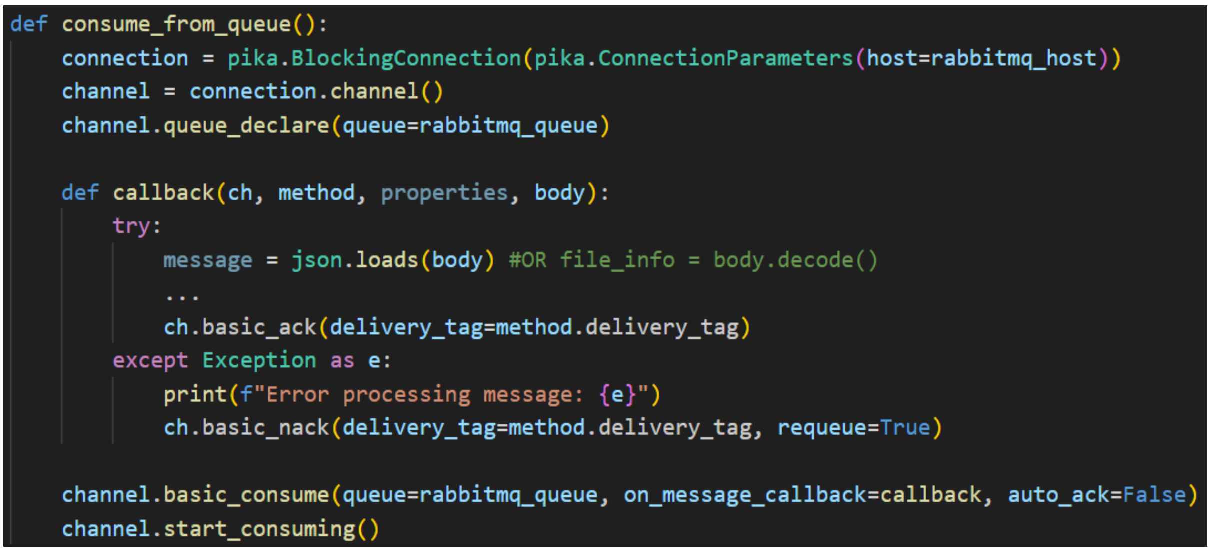
3. Service

   Exposes RabbitMQ to the network for discovery within the Kubernetes cluster.

- *Ports:* 5672 allows for AMQP connections, which means other applications (producers and consumers) can connect to RabbitMQ and send or receive messages. Port 4369 allows different clusters to discover each other.

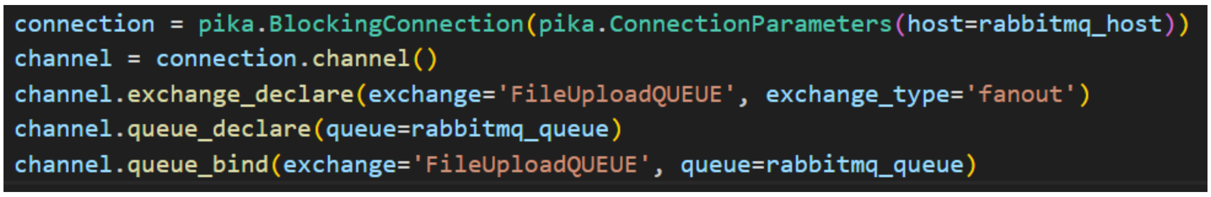
Working with RabbitMQ within the microservices

- Python Library: pika (pip install pika)
- Basics:
  - *Direct Exchange:* Each queue binds to the direct exchange using a key, and only messages with that exact key will be routed to the bound queue(s).
  - *Fanout Exchange:* This is like a public announcement, every queue listening to this exchange will receive the message. This is used when the tokens are sent to each analytics service.
  - *Acknowledgments*: We enable message acknowledgments to confirm message delivery, reducing the risk of data loss. If processing fails, RabbitMQ can route the message to a "retry" or "dead-letter" queue, where it waits for a fixed interval before a retry.
- Consumer code: Each service that listens for messages from the queue will have this function:
  - *Direct exchange:*



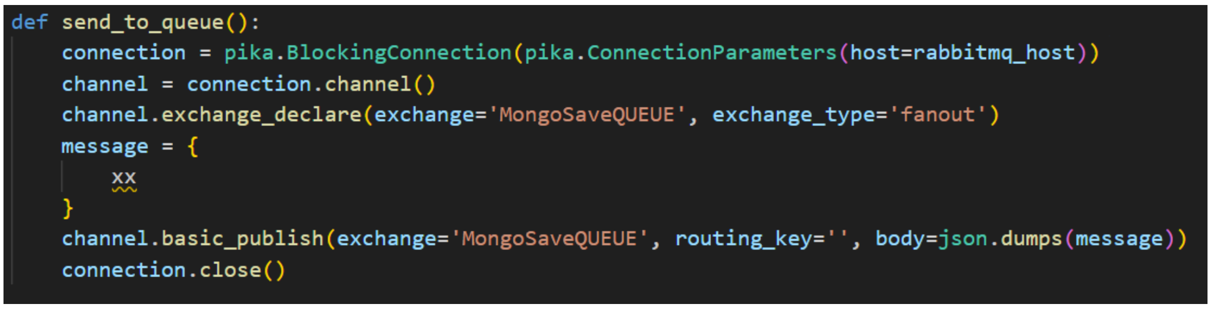
```python
def consume_from_queue():
    connection = pika.BlockingConnection(pika.ConnectionParameters(host=rabbitmq_host))
    channel = connection.channel()
    channel.queue_declare(queue=rabbitmq_queue)

    def callback(ch, method, properties, body):
        try:
            message = json.loads(body) #OR file_info = body.decode()
            ...
            ch.basic_ack(delivery_tag=method.delivery_tag)
        except Exception as e:
            print(f"Error processing message: {e}")
            ch.basic_nack(delivery_tag=method.delivery_tag, requeue=True)

    channel.basic_consume(queue=rabbitmq_queue, on_message_callback=callback, auto_ack=False)
    channel.start_consuming()
```

  - *Fanout exchange:*



```python
connection = pika.BlockingConnection(pika.ConnectionParameters(host=rabbitmq_host))
channel = connection.channel()
channel.exchange_declare(exchange='FileUploadQUEUE', exchange_type='fanout')
channel.queue_declare(queue=rabbitmq_queue)
channel.queue_bind(exchange='FileUploadQUEUE', queue=rabbitmq_queue)
```

- Publisher code: Each service that publishes something to the queue will have this code:



```
def send_to_queue():
    connection = pika.BlockingConnection(pika.ConnectionParameters(host=rabbitmq_host))
    channel = connection.channel()
    channel.exchange_declare(exchange='MongoSaveQUEUE', exchange_type='fanout')
    message = {
        xx
    }
    channel.basic_publish(exchange='MongoSaveQUEUE', routing_key='', body=json.dumps(message))
    connection.close()
```

C. MongoDB

This section explains the setup of the MongoDB database and the Mongo Express web interface in a Kubernetes Minikube environment.

The MongoDB deployment (YAML) contains these components:

1. MongoDB Headless Service (mongodb-headless-svc.yaml)
   The headless service exposes the MongoDB instance inside the Minikube cluster without a cluster IP. This allows other services to communicate directly with the MongoDB pod.
   Key features:
   - Exposes MongoDB on port 27017 within the cluster.
   - Acts as a headless service to allow direct pod communication.

2. MongoDB StatefulSet (mongodb-statefulset.yaml)
   The MongoDB StatefulSet ensures data persistence across pod restarts. It uses a Persistent Volume backed by the configured storage class.
   Key Features:
   - StatefulSet ensures that MongoDB data is retained even if the pod is restarted.
   - Volume Claim Template requests 10Gi of storage from the *mongodb-dynamic-storage* class.
   - Single replica for simplicity (can be scaled if needed).

3. Storage Class for MongoDB (mongodb-storageclass.yaml)
   The StorageClass defines how volumes are provisioned dynamically for the MongoDB StatefulSet. This is essential for persisting data within the Minikube environment.
   Key Features:
   - Uses Minikube's host path as the storage backend.
   - Volume Expansion is enabled, allowing flexibility in storage management.

- Reclaim Policy is set to *Retain*, preventing accidental data loss.

4. <u>Mongo Express Deployment and Service (mongo-express.yaml)</u>

   Mongo Express is a web-based UI for managing MongoDB. This component provides a deployment and service for exposing the UI within the Minikube cluster.

   Key Features:

   - Mongo Express runs on port 8081 within the Minikube cluster.
   - Connects to MongoDB via the *mongodb-headless-svc* on port *27017*.
   - Single replica deployment for simplicity.

How the Components Work Together:

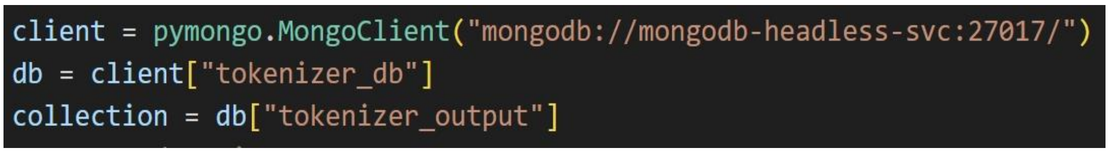
- MongoDB StatefulSet ensures data persistence with the help of the StorageClass.
- Headless Service exposes MongoDB within the cluster, allowing Mongo Express and other services to connect.
- Mongo Express Deployment provides a web interface for managing the MongoDB database, accessible on port *8081*.

Access Mongo Express

- Forward the Mongo Express service to your local machine: kubectl port-forward svc/mongo-express 8081:8081
- Now, you can access Mongo Express in your browser at: http://localhost:8081

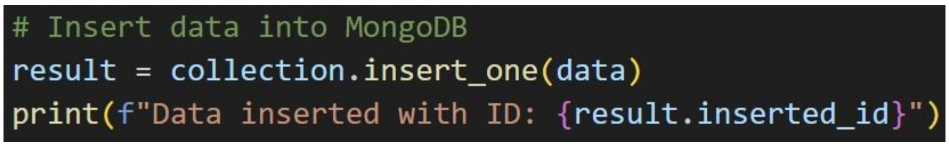
Working with MongoDB within the Microservices

- <u>Python library:</u> pymongo
- <u>Basic code</u>:
  - *Setting up the connection:* Create the connection to the MongoDB headless service and initialise the database and collection that will be used for the connection.



```
client = pymongo.MongoClient("mongodb://mongodb-headless-svc:27017/")
db = client["tokenizer_db"]
collection = db["tokenizer_output"]
```

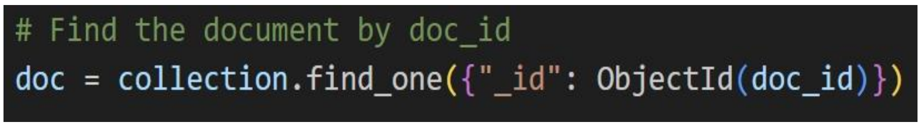
- *Saving a record* to MongoDB:



```
# Insert data into MongoDB
result = collection.insert_one(data)
print(f"Data inserted with ID: {result.inserted_id}")
```

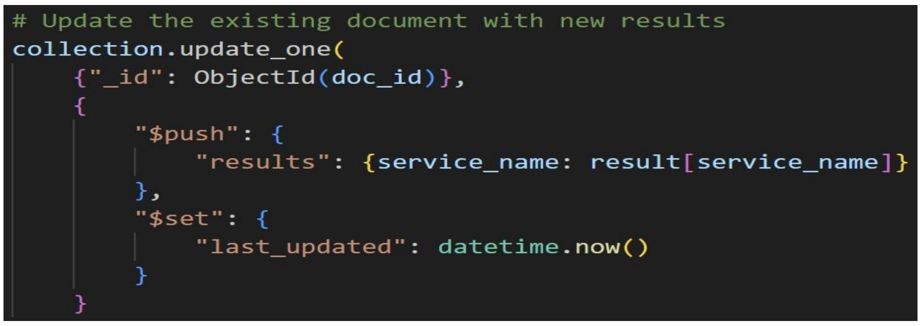
- *Updating a record* in a MongoDB database:

First, you need to find the specific record/document:



```
# Find the document by doc_id
doc = collection.find_one({"_id": ObjectId(doc_id)})
```

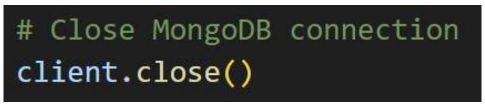
Then you can update the record/document:



```
# Update the existing document with new results
collection.update_one(
    {"_id": ObjectId(doc_id)},
    {
        "$push": {
            "results": {service_name: result[service_name]}
        },
        "$set": {
            "last_updated": datetime.now()
        }
    }
```

1. push: appends the information in the record/document.
2. set: overwrites and creates new fields in the record/document

- After writing data or reading data from the database the connection should be closed:



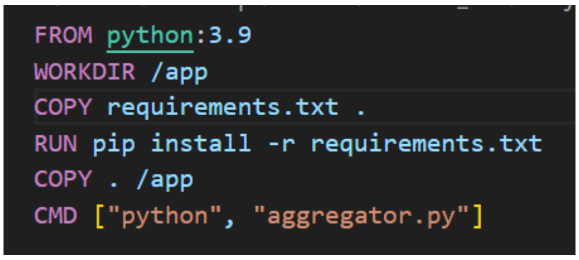
```
# Close MongoDB connection
client.close()
```

D. Docker

Each service (1-7) is deployed as a docker container, and all our images are on our DockerHub account. To create a new image:

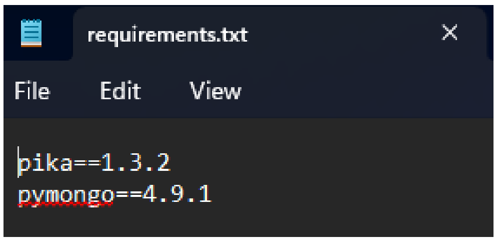
- You will have to firstly install docker on your computer/server and it has to be running when you want to use it.
- Create a Dockerfile:
    - It is a script with instructions to build a Docker image, defining the environment and steps needed to set up and run an application in a container.

- Sample Dockerfile (this should be saved with the name Dockerfile without any extension):



```
FROM python:3.9
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . /app
CMD ["python", "aggregator.py"]
```

- Create a requirements.txt file:
    - This is a plain text file with all the dependencies your program needs to run and the version if necessary.
    - For example:



```
pika==1.3.2
pymongo==4.9.1
```

- Build the image:
    - In the directory with your python script, Dockerfile and requirements file, run the command: docker build -f Dockerfile -t <your_dockerhub_username> /<image_name>:<tag> .
    - For example:
        - docker build -f Dockerfile -t similabs/mongo_save:v1 .
- Push to DockerHub:
    - docker push <your_dockerhub_username>/<image_name>:<tag>
    - The image is now available in your Docker Hub account and can be accessed for deployment.
- Deployment:
    - The images needs to be added to the respective deployment:
      For example: image: similabs/aggregator:v21 *(Needs to be updated when a new image is created or if no tag is given latest will be the default).*

E. Minikube

1. Start and Initialize Minikube Cluster

- Download and install minikube from minikube's installation guide.
- To start minikube: *minikube start --driver=docker*

- Verify it is running: *minikube status*
- Install kubectl
- To see all the components running in minikube cluster: *kubectl get all*
- To see the Kubernetes dashboard:
    - Server side: minikube dashboard --url
    - Local side:
        - ssh -L 8001:127.0.0.1:port cobitdev@196.252.135.195
        - http://127.0.0.1:8001/api/v1/namespaces/kubernetes-dashboard/services/http:kubernetes-dashboard:/proxy/#/workloads?namespace=default

2. Deploying Services with YAML Files

To apply deployment configurations and expose services, use the following commands.

- Apply a Deployment**:** *kubectl apply -f <deployment-file.yaml>*
  Example: Deploying MongoDB: *kubectl apply -f mongodb-statefulset.yaml*
- Check the Status of Deployments: *kubectl get deployments*
- List All Pods: *kubectl get pods --all-namespaces*
- Scale a Deployment (e.g., RabbitMQ) if needed: *kubectl scale statefulset rabbitmq --replicas=2*

3. Managing Docker Images within Minikube

Minikube allows local development with Docker, using Minikube's Docker environment.

- Build a Docker Image for Deployment: docker build -t similabs/mongo_save:v1 .
- Verify the Image Exists Locally: docker images

4. Monitoring and Troubleshooting

- View Logs of a Pod (e.g., RabbitMQ): kubectl logs <pod-name>
- Follow Pod Logs in Real-Time: kubectl logs -f <pod-name>
- Restart a Deployment: kubectl rollout restart deployment <deployment-name>

5. Managing Resources and Cleanup

- Delete a Specific Deployment: kubectl delete -f <deployment-file.yaml>
- Stop Minikube: minikube stop
- Delete Minikube Cluster: minikube delete

2. Document flow

This section explains the purpose of each service, how it operates, and highlights key details to remember.

2.1 User Interface

Overview: The User Interface is designed to facilitate single and batch file uploads. It is implemented using React components and communicates with a backend via HTTP requests to upload files.

- **Architecture:**
  - Frontend: React with multiple modular components.
  - Backend API: Communicates with *http://localhost:8085/upload* for file uploads.
  - UI Features: Progress bars, reset forms, batch upload support.

Component Details

a. *App.js*

The entry point of the React application that integrates the single and batch file upload components.

Functions:

- ***resetForms()***:
  - Resets the state of both the single and batch upload forms by calling the *resetForm()* method on their refs.

Structure:

- ***useRef()***: Maintains references to the *UploadForm* and *BatchUploadForm* components.
- **UI Layout**:
  - Title: *SimiLabs Linguistic Analysis*
  - Renders the two forms in a single container.

1. Usage:

The *resetForms()* function provides a way to reset both forms when necessary, ensuring users can easily restart the upload process.

b. *UploadForm.js*

A React component that enables **single-file uploads** with status and progress feedback.

State Variables:

- ***selectedFile***: Stores the currently selected file.
- ***uploadStatus***: Stores the status message of the upload.
- ***isUploading***: Tracks if the upload is in progress.
- ***progress***: Tracks the upload progress as a percentage.
- ***isUploaded***: Tracks if the upload is complete.

Functions:

- ***handleFileChange()***:
  - Updates the selected file when a user picks a file.
- ***handleUpload()***:
  - Uploads the selected file to *http://localhost:8085/upload* using **Axios**.
  - Tracks upload progress via the *onUploadProgress* callback.
- ***resetForm()***:
  - Exposed via *useImperativeHandle()* for resetting the form state.

UI Features:

- Progress bar with dynamic text colour based on percentage.
- Buttons to upload and reset the form.

c. *BatchUploadForm.js*

A React component for **batch file uploads** with progress tracking.

State Variables:

- ***selectedFiles***: Stores an array of files selected for upload.
- ***uploadStatus***: Stores the status message of the upload.
- ***isUploading***: Tracks if the upload is in progress.
- ***progress***: Tracks the overall upload progress as a percentage.
- ***isUploaded***: Tracks if the upload is complete.

Functions:

- ***handleFileChange()***:
  - Updates the selected files when a user selects multiple files.
- ***handleBatchUpload()***:

    - Sends multiple files to the backend using **Axios**.
    - Uses *FormData* to append files dynamically.
- ***resetForm()***:
    - Exposed via *useImperativeHandle()* for resetting the form state.

<u>UI Features:</u>

- Multiple file selection with a progress bar.
- Reset button to clear selected files.

d. *FileUploadPage.js*

A React component that acts as a **container** for the *UploadForm* and *BatchUploadForm* components, managing their layout and resetting their state through shared logic.

<u>State Variables:</u>

- ***resetKey***: A key used to force re-renders of the child components when their state needs to be reset.

<u>Functions:</u>

- ***resetForms()***:
    - Increments the *resetKey* to trigger re-renders and reset the forms.

<u>UI Features:</u>

- Renders both single and batch upload forms side by side.

2.2 Upload Service

Overview: The Upload Service is responsible for handling file uploads from clients, storing them in a MinIO object storage, and forwarding the file metadata to a RabbitMQ message queue for further processing by other services.

- **Language & Framework:** Python with Flask
- **File Name:** upload.py

Key Functionalities:

- <u>Environment configuration:</u>
    - Configures MinIO credentials (Kubernetes Secrets) and RabbitMQ host through environment variables.
    - The secrets can be created with the following command:

kubectl create secret generic minio-creds --from-literal=MINIO_ACCESS_KEY=your-access-key --from-literal=MINIO_SECRET_KEY=your-secret-key

Replace *your-access-key* and *your-secret-key* with your actual MinIO credentials (e.g., *minioadmin*).

- The secrets are accessed in the deployment yaml:
  - Kubenetes secrets reference in yaml file:

```
env:
        - name: MINIO_ACCESS_KEY
          valueFrom:
            secretKeyRef:
                name: minio-creds
                key: MINIO_ACCESS_KEY
        - name: MINIO_SECRET_KEY
          valueFrom:
             secretKeyRef:
                name: minio-creds
                key: MINIO_SECRET_KEY
```

- File Upload Endpoint (/upload):
  - *Method:* POST
  - *Description:* Receives a file, stores it in MinIO, and sends the file path to RabbitMQ.
  - *MinIO Client:*
    - Bucket Name: "fileupload"
    - Checks if the bucket exists and creates it if necessary.
  - *RabbitMQ Integration:*
    - Sends the file path to a RabbitMQ queue (FileUploadQUEUE) using the fanout exchange.

Deployment notes:

- The port and target port needs to be specified to the same port the Upload service is running on.
- The type of port also needs to be specified to: type: ClusterIP (This exposes the service to only be accessible within the Kubernetes cluster).

2.3 API Gateway

Overview: The Gateway API acts as a central entry point to the microservices system, routing client requests to the appropriate backend services. In this implementation, it is primarily designed to forward requests to the Upload Service.

- **Language & Framework:** Python with Flask
- **File Name:** gateway.py
- **Interacts with:** Upload service

Key Functionalities:

1. File Upload endpoint (/upload):
   - Method: POST
   - Description: Receives a file from the client and forwards it to the upload microservice.
   - Upload Microservice URL: The service URL is defined as UPLOAD_SERVICE_URL = 'http://upload-service:5001'. It forwards the request to this service.
   - Error Handling: In case of any request exceptions, the API responds with a JSON error message and a status code of 500.
   - Example of a request to upload a file (without UI): curl -X POST http://localhost:5000/upload -F file=@<file_path> (File path should be replaced with the file path on your local computer).

Deployment notes:

Gateway Deployment

- ports: - containerPort: 5000 (The port number used should correspond to the port the Gateway API is hosted on).

Gateway Service

- Use the correct ports and protocols:
  - The port and target port need to be specified to the same port the API Gateway is running on.
- The type of port also needs to be specified to: type: NodePort (This exposes the service to external clients through a port on the host node).

2.4 Tokenizer

Overview: This Python script receives a file path for a document saved in MinIO storage. It retrieves the document, extracts its text content, tokenizes the text into individual words, and then sends the tokenized data to a specified RabbitMQ queue for further processing.

- **Language & Framework:** Python with Flask
- **File Name:** tokenizer.py
- **Interacts with:**
  - MinIO: To retrieve the stored file.
  - RabbitMQ: As the messaging system for receiving and sending data.

Key Functionalities:

1. Environment configuration:
   - Configures MinIO credentials (Kubernetes Secrets) and RabbitMQ host through environment variables.

2. Consuming file upload events from RabbitMQ (consume_from_queue):
   - *Queue setup:* Connects to RabbitMQ FileUploadQUEUE to listen for new file upload events.
   - *Event Handling:* When a message is received:
     - Parses the message to get the MinIO bucket and file path.
     - Retrieves and processes the file.
     - Sends processed data to the MongoSaveQUEUE.

3. File retrieval from MinIO (get_file_from_minio):
   - Retrieves files from a specified MinIO bucket and loads them into memory.

4. File processing (process_file):
   - *Supported formats:* Processes .docx and .pdf files.
   - *Text Extraction:*
     - DOCX: Uses docx.Document to extract text from Word documents.
     - PDF: Uses PyPDF2 to extract text from each page in PDF files.
   - *Text Splitting:* Tokenizes the extracted text into words for easier processing in later stages.

5. Sending processed data to RabbitMQ (send_to_queue):
   - *Queue set-up:* Publishes tokenized text data to MongoSaveQUEUE via RabbitMQ, where it can be further processed or stored. This is described in the next step.
   - *Message structure:* JSON formatted message containing file_path and the tokenized_data.

2.5 Save to database

Overview: The MongoDB Save Component is a microservice designed to consume messages from RabbitMQ, process tokenized data, and store it in a MongoDB database. This component is deployed on Kubernetes Minikube.

- **Language & Framework:** Python with Flask
- **File Name:** MongoSave.py
- **Interacts with:**
    - MongoDB: For persistent data storage.
    - RabbitMQ: As the messaging system for receiving and sending data.

Key Functionalities:

1. Consume messages from RabbitMQ:
    - Retrieves tokenized data from a queue.
    - Parses the message and extracts *file_path* and *tokenized_data*.
2. Save Data to MongoDB:
    - Connects to the MongoDB service through *mongodb-headless-svc:27027 and initializes the database and collection name that will be used.*
    - Inserts data into the *tokenizer_output* collection with metadata:
        - Unique *UUID*
        - Timestamp of insertion.
3. Send Messages to Other Microservices:
    - Sends payloads through a RabbitMQ fanout exchange to notify other services.
4. Exception Handling:
    - Handles RabbitMQ and MongoDB connection errors.
    - Supports message re-queuing if processing fails.

Deployment notes:

- Environment Variable: *MONGO_URI* pointing to *mongodb-headless-svc:27017* (Needs to point to the MongoDB server created in Kubernetes Minikube with the correct port)

2.6 Various analytics

Overview: These analytics services receive input in the form of a tokenized document. This tokenized data is used to perform the required analyses on the data and provide the results in the appropriate format.

- **Language and Framework**: Python with Flask
- **File Name(s)**: various analysis scripts ending with -service.py e.g. lexical-richness-service.py

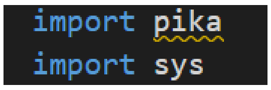
Key Functionalities:

1. Consume message from RabbitMQ:
   - Retrieves tokenized documents from the queue using the consume_from_queue() function.
   - Decodes messages and formats to expected JSON input format.
2. Process input data from message:
   - Executes the relevant function in the service to process data and provide relevant output in JSON format
3. Send output results back to RabbitMQ:
   - Sends output as a message to the aggregator queue for further handling using the send_to_aggregator() function

Adding a new microservice:

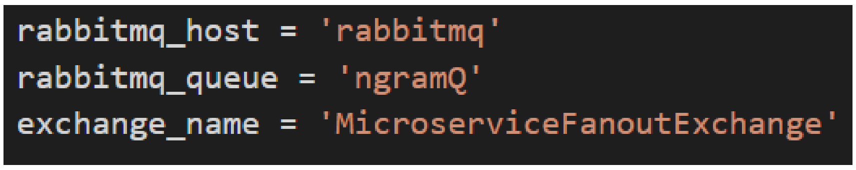
Existing microservice code needs to be modified to communicate with RabbitMQ and to ensure seamless integration with the overall system. An example of how this can be achieved is shown using the ngram-analysis-service.py code:

1. Import the **pika** and **sys** libraries (use pip install if library imports cannot be resolved, highlighted in yellow).



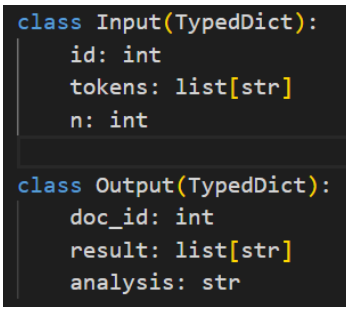
```
import pika
import sys
```

2. Define your RabbitMQ queue details in variables for ease of use. Take special care to ensure that the raabitmq_queue name is an appropriate name for the queue for your service.

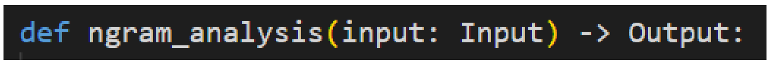


```
rabbitmq_host = 'rabbitmq'
rabbitmq_queue = 'ngramQ'
exchange_name = 'MicroserviceFanoutExchange'
```

3. Define your input and output format. While this step isn't necessary, it is helpful and recommended to ensure ease of future modification and readability. Take care to ensure that the document id (doc_id) as well as the type of analysis is also returned.

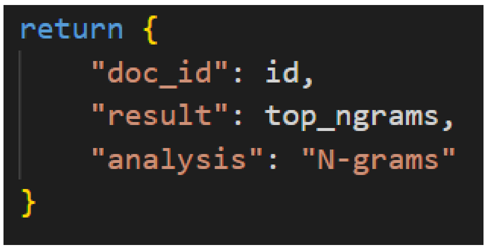


```python
class Input(TypedDict):
    id: int
    tokens: list[str]
    n: int

class Output(TypedDict):
    doc_id: int
    result: list[str]
    analysis: str
```



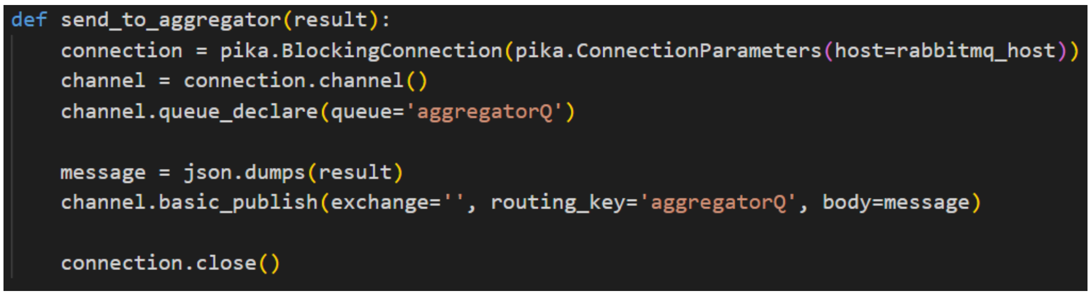
```python
def ngram_analysis(input: Input) -> Output:
```

4. **NB**: Ensure your microservice function that is called returns the correct output, and that it is correctly returned as “result” in the JSON. Also ensure that the return output matches the expected output format.



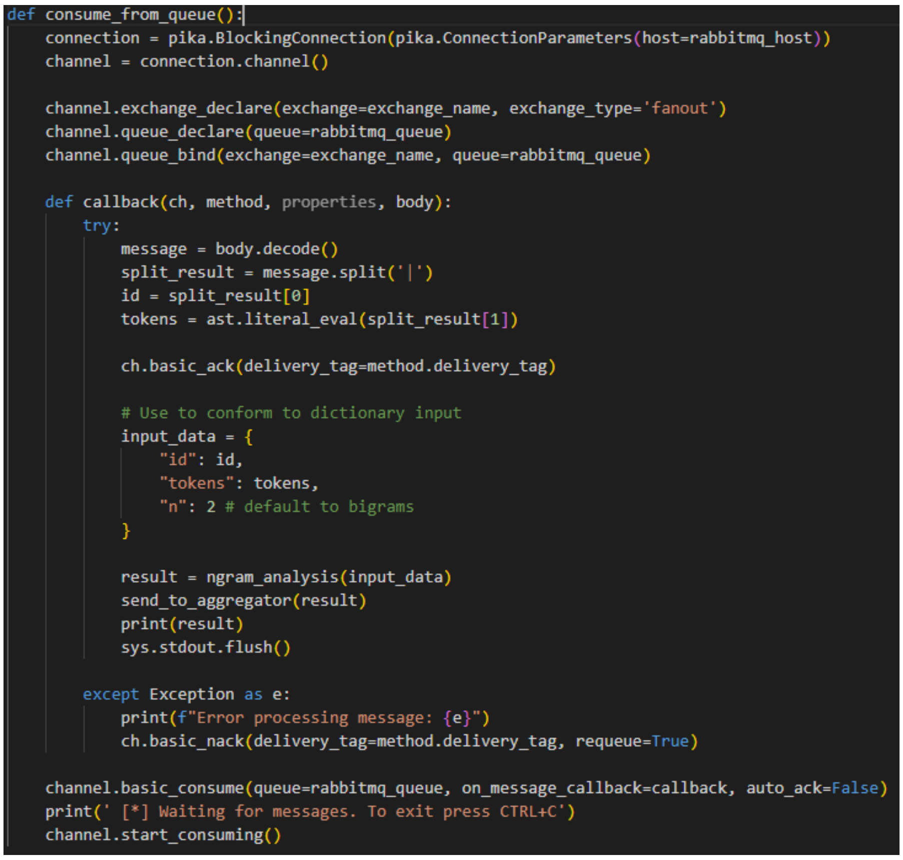
```python
return {
    "doc_id": id,
    "result": top_ngrams,
    "analysis": "N-grams"
}
```

5. Add the code that sends the result to the aggregator queue



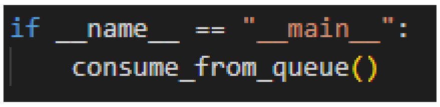
```python
def send_to_aggregator(result):
    connection = pika.BlockingConnection(pika.ConnectionParameters(host=rabbitmq_host))
    channel = connection.channel()
    channel.queue_declare(queue='aggregatorQ')

    message = json.dumps(result)
    channel.basic_publish(exchange='', routing_key='aggregatorQ', body=message)

    connection.close()
```

6. Add the code that enables the microservice to consume messages from the queue. Take note of how the input data conforms to the specified input, and how the results are obtained and sent to the aggregator shortly thereafter.



```python
def consume_from_queue():
    connection = pika.BlockingConnection(pika.ConnectionParameters(host=rabbitmq_host))
    channel = connection.channel()

    channel.exchange_declare(exchange=exchange_name, exchange_type='fanout')
    channel.queue_declare(queue=rabbitmq_queue)
    channel.queue_bind(exchange=exchange_name, queue=rabbitmq_queue)

    def callback(ch, method, properties, body):
        try:
            message = body.decode()
            split_result = message.split('|')
            id = split_result[0]
            tokens = ast.literal_eval(split_result[1])

            ch.basic_ack(delivery_tag=method.delivery_tag)

            # Use to conform to dictionary input
            input_data = {
                "id": id,
                "tokens": tokens,
                "n": 2 # default to bigrams
            }

            result = ngram_analysis(input_data)
            send_to_aggregator(result)
            print(result)
            sys.stdout.flush()

        except Exception as e:
            print(f"Error processing message: {e}")
            ch.basic_nack(delivery_tag=method.delivery_tag, requeue=True)

    channel.basic_consume(queue=rabbitmq_queue, on_message_callback=callback, auto_ack=False)
    print(' [*] Waiting for messages. To exit press CTRL+C')
    channel.start_consuming()
```

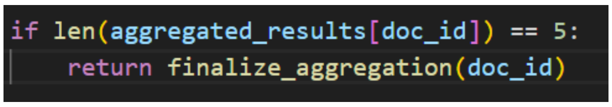
7. Ensure that your “main” function makes use of the new consume_from_queue() function.



```python
if __name__ == "__main__":
    consume_from_queue()
```

8. Once this is completed, the relevant Dockerfile and requirements.txt can be created to containerize the code and push it to Dockerhub. The appropriate microservice YAML configuration file can also be created using the correct version of the Docker Hub image to deploy onto the Minikube environment.

Note: This section of the aggregator service also needs to be changed for the amount of analysis service there are (now 5):



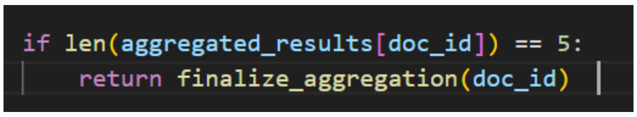
```python
if len(aggregated_results[doc_id]) == 5:
    return finalize_aggregation(doc_id)
```

2.7 Aggregator

Overview: The Aggregator service is a microservice responsible for aggregating results from multiple other services. It consumes messages from a RabbitMQ queue and stores the aggregated results in a MongoDB database when all relevant microservice results for a particular file are received. Specifically, it waits to receive five results for the same *doc_id* (file) and then finalises the aggregation.

Key Functionalities:

1. Receiving and aggregating messages:
    - The service listens on the *aggregatorQ* RabbitMQ queue for incoming messages.
    - Each message contains:
        - *doc_id*: Identifier for the file.
        - *result*: The analysis results from a microservice.
        - *analysis*: The name of the microservice that provided the result.
2. Finalising aggregation:
    - When five results for the same *doc_id* are received, the service saves the aggregated results in MongoDB. This should be changed when more services are added which sends results to the aggregator.



```
if len(aggregated_results[doc_id]) == 5:
    return finalize_aggregation(doc_id)
```

3. MongoDB integration:
    - Uses the *tokenizer_db* database and *tokenizer_output* collection.
    - If the document already exists, it updates and adds the results received from the microservice modules and sets the *last_updated* timestamp.
4. Error handling:
    - If message processing fails, the service re-queues the message for future retries.

Deployment notes:

- MongoDB URI: The environment variable *MONGO_URI* points to *mongodb-headless-svc:27017* (Needs to point to the MongoDB server created in Kubernetes Minikube with the correct port).

2.8 User feedback

Overview: Processes feedback by consuming tasks from RabbitMQ, retrieving data from MongoDB, generating PDF reports using ReportLab, and storing them in a MinIO bucket.

- **Language & Framework:** Python with Flask

- **File Name:** UserFeedback.py

Key Functionalities:

1. <u>MongoDB Query (Retrieve Data):</u>
    - Database: *tokenizer_db*
    - Collection: *tokenizer_output*
    - Functionality: Retrieves the *results* section and *file_path* from MongoDB using the provided document ID.
2. <u>PDF Generation:</u>
    - Library: ReportLab
    - Function: Generates a structured PDF report with data retrieved from MongoDB.
3. <u>MinIO Storage:</u>
    - Bucket Name: *feedback*
    - Service URL: *minio-headless-svc:9000*
    - Environment Variables: Uses *MINIO_ACCESS_KEY* and *MINIO_SECRET_KEY* for authentication.
    - Purpose: Uploads generated PDF reports to MinIO for long-term storage.
4. <u>Error Handling:</u>
    - If data retrieval from MongoDB fails, the message is requeued.
    - If PDF generation or MinIO upload fails, the process logs the error and attempts to reprocess.

Deployment notes:

- MinIO credentials (*MINIO_ACCESS_KEY* and *MINIO_SECRET_KEY*) must be configured via environment variables.

Deployment

This section outlines the necessary steps to set up the server environment, install required technologies and packages, and deploy the system successfully. Follow the instructions carefully to ensure a smooth deployment process.

A. Server Environment Setup

Before deploying the system, ensure that the server environment is configured correctly. You can automate the server environment setup by using the provided script.

Steps to setup the environment:

1. Clone the Repository:
   - Clone the application repository that contains the envSetup.sh script.

2. Make the Script Executable:
   - Make the script executable by running the following command:

```
chmod +x envSetup.sh
```

Command: chmod +x envSetup.sh

3. Run the Script:
   - Execute the script to install the required technologies and packages.

```
./envSetup.sh
```

Command: ./envSetup.sh

This script will automatically handle the installation and setup of the required technologies.

Script contents:

The envSetup.sh script performs the following:

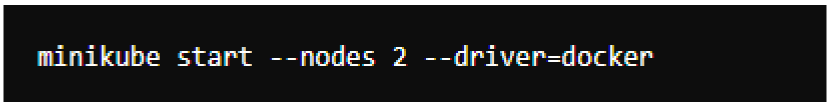
1. Remove Conflicting Docker Packages: Uninstalls existing Docker-related packages.
2. Update Package Index: Updates the local package index.
3. Install Dependencies: Instals essential packages like ca-certificates and curl.
4. Add Docker GPG Key and Repository: Adds Docker's GPG key and repository to package sources.

5. Install Docker: Instals Docker Engine, CLI, containerd, Buildx, and Docker Compose.
6. Add User to Docker Group: Grants the current user permissions to run Docker commands.
7. Install kubectl: Downloads and instals the Kubernetes CLI tool.
8. Install Minikube: Instals Minikube for local Kubernetes management.
9. Install Helm: Instals Helm via Snap for managing Kubernetes applications.

B. System deployment

Steps to deploy the system:

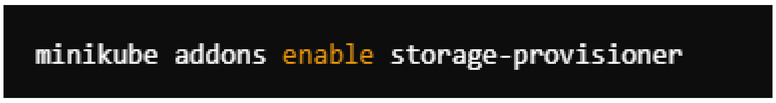
1. Start MiniKube:
   - Run the following command to start MiniKube



```
minikube start --nodes 2 --driver=docker
```

   - Command: minikube start --nodes 2 --driver=docker
   - This initiates a MiniKube cluster with two nodes using Docker as the driver.

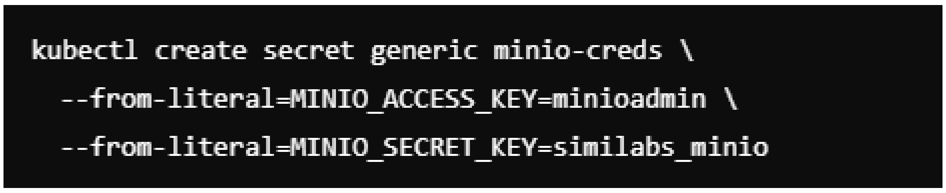
2. Ensure Dynamic Provisioning is Enabled:
   - Next, enable the storage-provisioner addon in MiniKube.



```
minikube addons enable storage-provisioner
```

   - Command: minikube addons enable storage-provisioner
   - Dynamic provisioning allows Kubernetes to automatically create persistent storage volumes when requested by applications, without requiring manual intervention.

3. Create System Secrets:
   - Create a Kubernetes secret named minio-creds that stores sensitive information for accessing MinIO, an object storage service.



```
kubectl create secret generic minio-creds \
  --from-literal=MINIO_ACCESS_KEY=minioadmin \
  --from-literal=MINIO_SECRET_KEY=similabs_minio
```

   - Command: kubectl create secret generic minio-creds \ --from-literal=MINIO_ACCESS_KEY=minioadmin \ --from-literal=MINIO_SECRET_KEY=similabs_minio

- The command uses the kubectl create secret command to generate a generic secret. The --from-literal flags define two key-value pairs: MINIO_ACCESS_KEY and MINIO_SECRET_KEY.

4. <u>Deployment:</u>
   - Deploy the system using the following command:

```
kubectl apply -f similabsSystemDeployment.yaml
```

   - Command: kubectl apply -f similabsSystemDeployment.yaml
   - This command deploys the application defined in the similabsSystemDeployment.yaml file to the Kubernetes cluster.

5. <u>Setup MinIO Bucket Policy:</u>
   - This step involves configuring the bucket policies for MinIO to make two specific buckets public. This allows external access to the data stored in these buckets, which is essential if your application needs to share files or serve assets directly from MinIO.

C. System Termination

To safely terminate and clean up the deployed system, follow the steps outlined below using the provided delete script. This ensures that all resources are removed from your Kubernetes cluster, and any associated local configurations are also cleaned up.

Steps to terminate the system:

1. <u>Make the Script Executable:</u>
   - Make the script executable by running the following command:

```
chmod +x deleteDeploymentScript.sh
```

   - Command: chmod +x deleteDeploymentScript.sh

2. <u>Run the Script:</u>
   - After making the script executable, execute it to start the termination process. Use the following command:

```
./deleteDeploymentScript.sh
```

   - Command: ./deleteDeploymentScript.sh

- This script will automatically handle the deletion of all Kubernetes deployments, services, persistent volume claims, and any associated secrets, ensuring a complete and efficient cleanup of your system.

D. Troubleshooting

For troubleshooting guidance related to common issues, please refer to the *Troubleshooting* section of the User Manual. This section provides detailed steps and solutions that can help resolve various problems encountered during the system's operation.