University of Groningen

Enterprise Application Integration

Viral Image Analysis!

Authors

Swastik Nayak	s4151968
Anil Mathew	s4056167
Filipe Capela	s4040112
Mark Soelman	s3224708

Coach:
Andrea de LUCIA

Lecturer:
prof. dr. Dimka KARASTOYANOVA

January 11, 2021

GitHub Repository:

https://github.com/FilipeCapela98/EAI_Project

Contents

1 Introduction

Any users of the social media platform Twitter[1] are able to post so-called Tweets. Tweets may consist of various media types, including text, images and videos. Furthermore, users may enrich the text by adding URLs and hashtags. The media being streamed on this platform can be very diverse. Even on a simple hashtag such as #CAT, images may be posted that are completely unrelated to cats. For this project, the authors are interested in creating a system that should be integrated with the Twitter stream on a particular hashtag, in order to receive all of these images. Then, these images should be analyzed by performing image classification to detect what objects are present in the image. Afterwards, the results of these classifications will be shown to the user's browser. Using the classifications shown to the user, the user is able to determine if there is a common trend of objects that often appear on any particular hashtag.
This report serves as documentation for the architecture of our system. The remainder of this report will have the following structure. Section 2 aims to describe the system from the perspective of the architects, by concisely listing the use cases the system should support as well as determine the functional requirements. Then, the authors provide a holistic view of the system is given in Section 3, in an attempt to make the reader's mental image of the system slightly more concrete. In Section 4, the authors elaborate on the tiered architecture. The purpose of this tiered architecture is to illustrate how the system is broken up in separate layers, that are (or rather should be) physically separated from each other. Section 5 focuses on the detailed architecture of the system, with a strong focus on the integration aspects. First, the architectural design decisions are formed on the patterns the system should use for performing the integration, after which concrete architectural views are shown. After having discussed the integration of systems, we zoom in on the individual software components as well as technologies used in Section 6. Section 7 is dedicated to discussing a proposed implementation as Minimum Viable Product of the system, by focusing on the technologies used and how the patterns were implemented. Finally, this document is concluded in Section 9.

[1] https://www.twitter.com/

2 System Description

The goal of this section is to give an overall idea of the system. We start by explaining the Use case diagram and then the Functional requirements.

2.1 Use Case Diagram

To further understand the basic concepts and functionality of the system, several use cases are mentioned. For each of these use cases, the actors, goals, and conditions are briefly described. These use cases are presented in Figure 1.

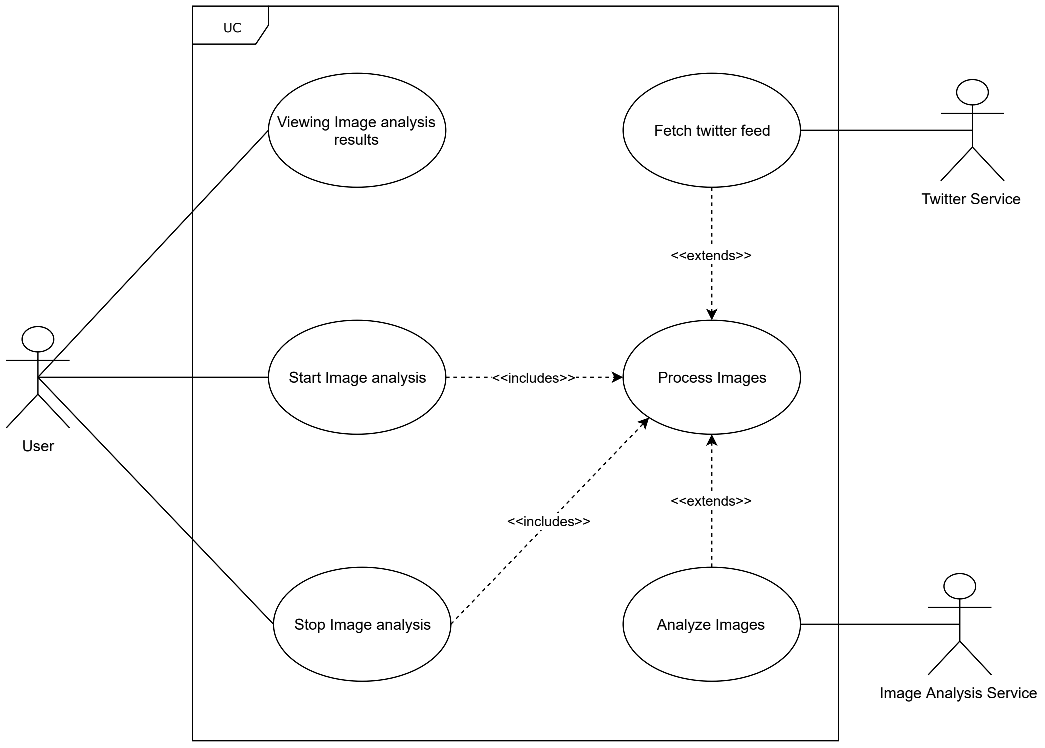


Figure 1: Use Case Diagram

UC-1	**Start Image Analysis**

Primary Actor(s)	User
Goal	User is able to trigger the Image Analysis pipeline.
Postcondition	Image analysis pipeline starts and the processed image is stored persistently in database.

Main success scenario

1. The end user opens the interface.
2. The end user fills in the textbox with the hashtag.
3. The end user clicks the start image analysis button.
4. The system notifies the user that the pipeline has been triggered.
5. Image analysis is performed by the system (UC-4.1).
6. The user is notified regarding the status of the process.

Extensions

3a. The end user triggers the image analysis pipeline with the filled in hashtag.

  3a-1. The system verifies if the hastag that the user entered is valid in Twitter.

  3a-2. Main success scenario continues at Step 4.

UC-2	**Stop Image Analysis**
Primary Actor(s)	User
Goal	User is able to stop the Image Analysis pipeline.

Postcondition	Image analysis pipeline stops.

Main success scenario

1. The end user opens the interface.
2. The end user fills in the textbox with the hashtag.
3. The end user clicks the stop image analysis button.
4. The system notifies the user that the pipeline has been stopped.
5. Current Image analysis process is stopped by the system (UC-4.1).
6. The user is notified regarding the status of the request.

UC-3	**Viewing Image Analysis results**
Primary Actor(s)	User
Goal	User is able to view the image analysis results.
Postcondition	The analyzed and processed image analysis results can be viewed by the users

Main success scenario

1. The end user opens the interface.
2. The end users can view all the processed images.

UC-4.1	**Process Images**
Primary Actor(s)	System

Goal	Image analysis and processing is triggered or stopped. Here the tweets are filtered and the extracted images are processed using the image analysis service. Finally, the results are stored in database.
Precondition	The image analysis pipeline is triggered/stopped by the user using the interface.
Postcondition	System starts a new pipeline or stops the existing one.

UC-4.2	**Fetch Twitter feed**
Primary Actor(s)	Twitter Service
Goal	The Twitter service fetches tweets from Twitter based on the hashtag passed by the user from the interface.
Precondition	The image analysis pipeline is triggered/stopped by the user using the interface.
Postcondition	Raw tweets are fetched from Twitter.

UC-4.3	**Analyze Images**
Primary Actor(s)	Image Analysis Service
Goal	Extracted and processed images from the tweets are classified based on the detected objects.

Precondition	The image analysis pipeline is triggered by the user using the interface, and the Process Images service(UC-4.1) has some processed images.
Postcondition	Images are classified and stored in database.

2.2 Functional Requirements

This section of the document provides the list of functional requirements of the system developed.

Index	**Description**
FR-1	The user shall be able to choose a hashtag for the image analysis.
FR-2	The user shall be able to explicitly start the analysis pipeline.
FR-3	The user shall be able to explicitly stop the analysis pipeline.
FR-4	The user shall be able to view the result of the image analysis using a Graphical User Interface.
FR-5	Once a user has closed the interface and no other users are subscribed to the same hashtag, ingestion of new images into the processing pipeline should stop.
FR-6	Multiple users should be able to listen to the same hashtag, without the same images being processed twice.
FR-7	The system should pre-process the output of the Twitter stream, to convert it to a suitable format for the image analyzer.
FR-8	The image analyzer should classify the image, by detecting which objects can be observed.
FR-9	After an image has been analyzed, the results of the image analysis are stored persistently.

3 Simplified Architectural Vision

A diagram illustrating our simplified architectural vision is provided in Figure 2. The diagram starts with a user, who interacts with the system through a web application. The user does not interact with any of the other systems directly. Using this web application, the user should be able to start an image processing pipeline, view the results, and stop the processing pipeline. This yellow component in the center illustrates the main ecosystem that should provide this functionality.
As illustrated in the yellow boxes at the center, our system should integrate with the Twitter API to receive tweets containing images. The output should be preprocessed, such that the result can be ingested by the Image Analyzer. The image analyzer is a system that will not be designed from the ground up for this project, but should be a pre-existing service that will be re-used. Although the exact output representation of this service depends on the technology and service used, the output should make clear types of objects can be observed on the ingested image using text. The resulting analysis of the image will be stored, as well as forwarded to the web application to be shown to the user.
By looking at this simplified view, it should immediately become apparent that the system-to-be will not be a single piece of software. Rather, the result will be many different components that have to communicate with each other. On top of that, it is worth mentioning that our system-to-be will require integration with some pre-existing technologies, such as the Twitter API, the Cloud Storage and the Image Analyzer.

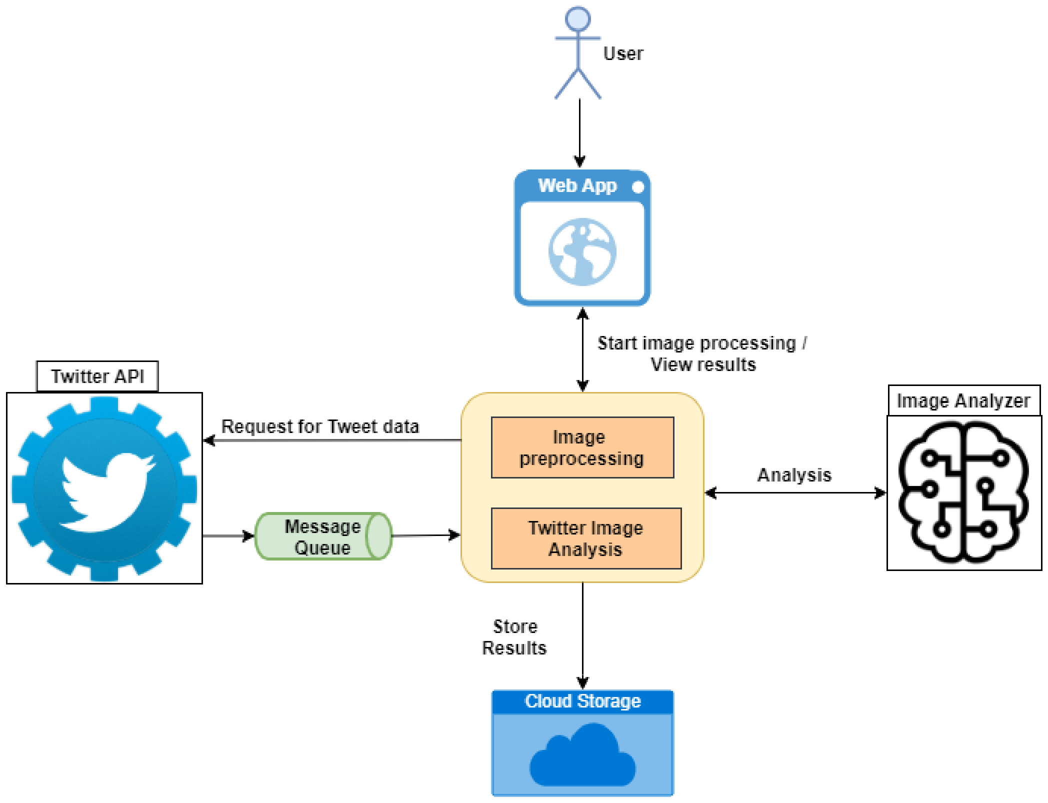


Figure 2: Simplified Architectural Vision

4 Tiered Architecture

In this section, we present our tiered architecture in Figure 3. It becomes apparent that our system comprises three tiers. On top, we have the Presentation Tier, which has no higher-level tiers calling this layer. The Presentation Tier only interacts with tier 2 applications, which include the main business logic. The Business Logic Tier contains our main pipeline for fetching and processing data. The business logic layer in turn is using services provided by the Data and Cloud Tier (tier 3). The services in tier 3 are external to our system, which is why no further dependencies are listed of them as they are external to the scope of our software project. The individual services are not discussed now, as they will be elaborated on in great depth later. Please note that every single service may be deployed on different machines, but we decided to group the business logic in the same tier as this is more intuitive.

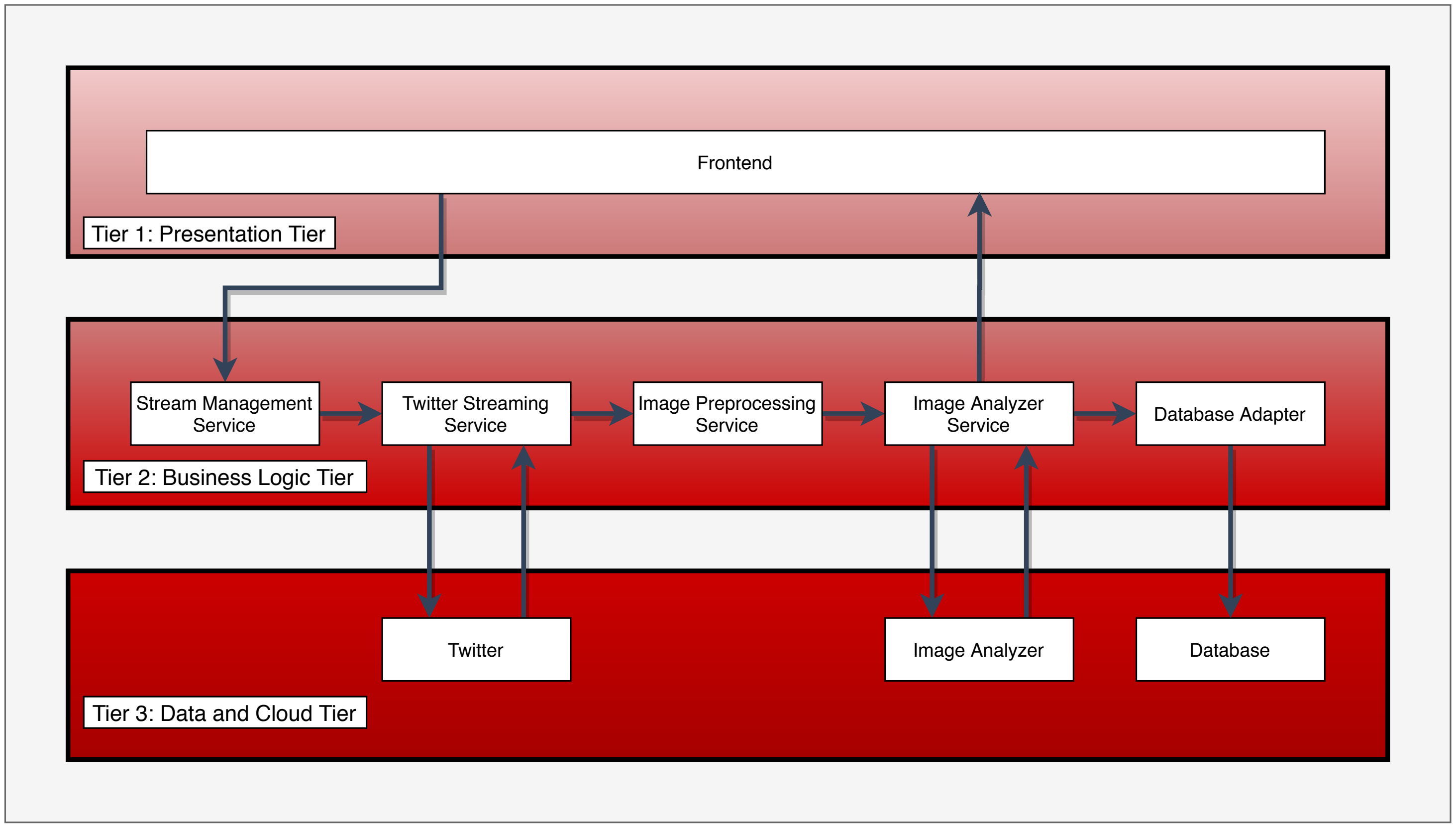

Frontend
Tier 1: Presentation Tier
Stream Management Service
Twitter Streaming Service
Image Preprocessing Service
Image Analyzer Service
Database Adapter
Tier 2: Business Logic Tier
Twitter
Image Analyzer
Database
Tier 3: Data and Cloud Tier

Figure 3: Multitier Architecture

5 System Integration Architecture

This section aims to introduce various issues our software system aims to overcome. We achieve this on a pattern-first design, meaning we first introduce the problems to solve, then patterns are discussed that may solve this problem. Only after having identified the patterns the system should integrate, architectural views are proposed. Using this ordering, the architecture is designed to flow from the requirements, and not the other way around.

5.1 Design Decisions: Integration patterns

The section briefly discusses the different key integration messaging patterns that were chosen to facilitate seamless communication between different components of our system.

5.1.1 Integration Style in-between processing pipeline components

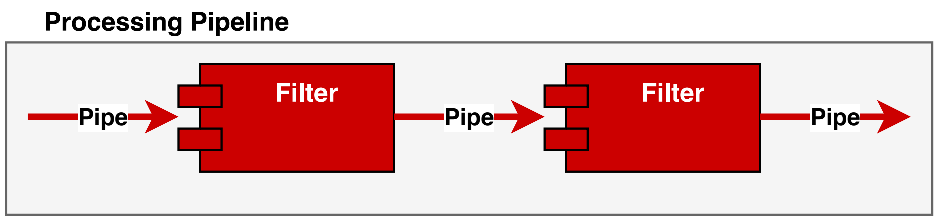


Figure 4: Pipes and Filters

Issue: The output of the Twitter stream requires multiple analysis and transformation steps. How will the different components, performing these steps, be integrated with each other?
Assumptions/Constraints: The output of the twitter stream possibly contains images. Images may be of sufficiently larger size than plain text. The analysis step may take a long time to complete.
Positions

- File Transfer
- Shared Database
- Remote Procedure Invocation

- Messaging
- Point-to-Point Channel
- Publish-Subscribe Channel
- Document Message
- Command Message
- Event Message
- Pipes and Filters

Source: Chapter 2, Chapter 3, Chapter 4 and Chapter 5 of [2].
Decision: File Transfer would have been an ideal solution, if we would only be transferring images from one system to another. This is, however, not the case since entire tweets may be transferred as well. The expectation is also that the output of the analysis step is not just a single image, but that this output will contain metadata as well, such as the objects detected in the image. Therefore, File Transfer does not suit our needs. Shared Database adds unnecessary complexity, especially since we are dealing with byte data (images). Remote Procedure Invocation would be possible, but if there is a large spike in input, the processors in the pipeline may become overloaded. **Messaging** seems to be well-suited to solve this problem. In addition, the components in our pipeline should perform small operations to produce an output from an input. This corresponds to the **Pipes and Filters**-pattern. Furthermore, we only want the same image to be processed by a single processor, and thus, decide to make this a **Point-to-Point Channel**. Additionally, since data is being transferred between applications, the messages being transferred are **Document Messages**.
Illustration: Figure 4

5.1.2 Delivering Analysis Results to the Frontend

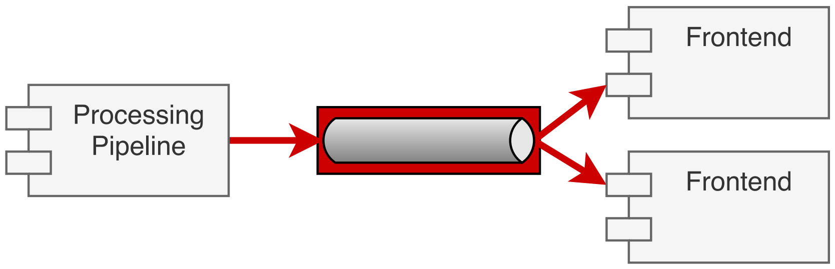


Figure 5: Integration from Pipeline to Frontend

Issue: After an image has finished processing in our pipeline, it should be shown to the user in the frontend. How are the results communicated to the user?
Assumptions/Constraints: There can be more than 1 user subscribed to the same stream.
Positions

- File Transfer
- Remote Procedure Invocation
- Messaging
- Point-to-Point Channel
- Publish-Subscribe Channel

Source: Chapter 2, Chapter 3 and Chapter 4 of [2].
Decision: For the same reason as 5.1.1, file transfer will not suffice. Remote Procedure Invocation would be possible, although this would be difficult. If the pipeline component needs to send it to the frontend, how does it know which of the users to send it to? The same holds true if the frontend uses RPC to the pipeline. Instead, messaging seem to be the ideal case, as multiple parallel computing units can publish their results to the same **Message Channel**. Furthermore, since multiple users can be interested in the same stream, this should be a **Publish-Subscribe Channel**.
Illustration: Figure 5

5.1.3 Pipeline Lifecycle Management

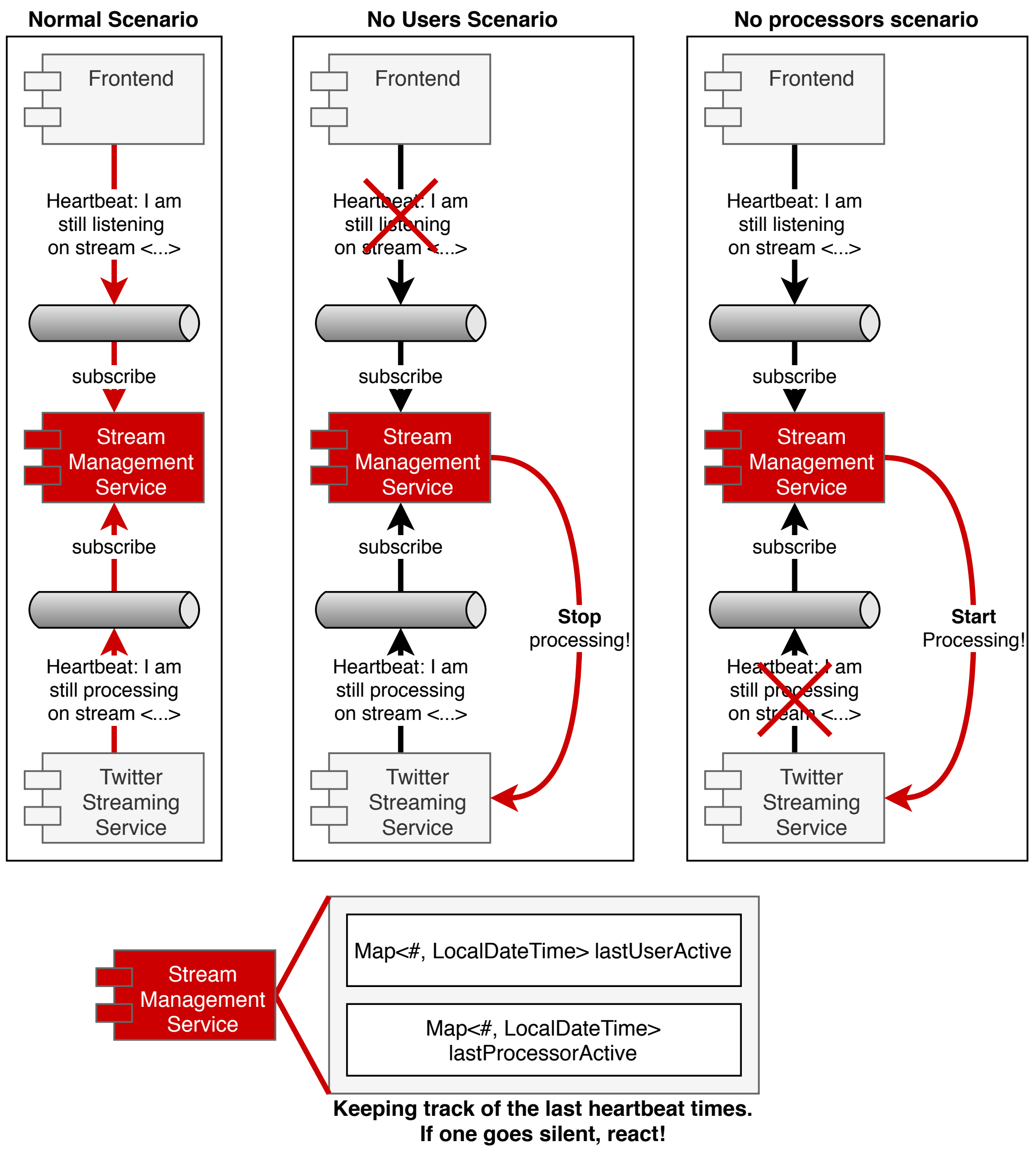

Normal Scenario
No Users Scenario
No processors scenario
Frontend
Heartbeat: I am still listening on stream <...>
subscribe
Stream Management Service
subscribe
Heartbeat: I am still processing on stream <...>
Twitter Streaming Service
Stop processing!
Start Processing!
Map<#, LocalDateTime> lastUserActive
Map<#, LocalDateTime> lastProcessorActive
Keeping track of the last heartbeat times.
If one goes silent, react!

Figure 6: Pipeline Lifecycle Management

Issue: It was explained earlier that streams should be started and/or stopped, based on the connected users. This is challenging to coordinate. When do we start a stream, and when do we stop a stream? However, what happens when users are simply disconnected without closing the stream tidily? What if their UI crashes, their internet connectivity goes offline, or other malfunctions occur? In any case, when there are 0 users left for a given stream, the stream is simply wasting resources and should stop immediately.
Assumptions/Constraints:

- Multiple users can open the same stream simultaneously, then, the stream should only be opened once.
- At any point in time, a stream may have 0 to many users watching the results.
- Users cannot be expected to sign off when they leave the stream/frontend, as network issues may occur, a device may crash, etc.
- When 0 users are watching the output of a stream, the pipeline should be stopped to save resources.
- If one user disconnects from the output of a stream, but others are still connected, the stream should continue to process.
- When more than 0 users are watching the output of a stream, but not a single stream processor is ingesting the input, there is no processing pipeline running and a new one needs to be created.

Positions

- Mediator
- Observer
- Point-to-Point Channel
- Publish-Subscribe Channel
- Command Message
- Event Message
- Request-Reply

- Correlation Identifier
- Idempotent Receiver
- Content-based router

Source: Chapter 5 of [1] and Chapter 4, 5, 7, 10 of [2].
Decision: A request-reply could be used to manage the lifecycle: a request is made to ask if someone is processing on a topic (hashtag), and if the reply is negative from all processors, a new service is started. This results in tight coupling between the processors and the users, which we want to avoid.
Alternatively, a mediator can be added in between the users and the processors, to coordinate starting/stopping the services. That way, the user need not be concerned about the underlying processing implementation/services. Still, the mediator would be quite complex. It would have to check what topics users are listening to, by making requests to the frontend which require a stateful open connection. Also, this does not scale well when multiple of these mediators are deployed.
That brings us to our third and final consideration: the **observer**. Again, we are creating an additional service in between the user's frontend and the processors. Only this time, they both don't know this intermediary observer exists. We call this the Stream Management Service. This service will listen to **Event Messages**, including heartbeats of all users: "I am interested in the stream of #...". It will also listen to the heartbeats of all processors: "I am ingesting data of #...". Then, these messages are processed differently using a **Content-based router**: if a heartbeat of a user is received, check if there is a process active in the registry, if not, start the pipeline using a start **Command Message** on a point-to-point channel. If a heartbeat of the process is received, check if users are active on that topic, if not, stop the pipeline using a stop Command Messgae on a point-to-point channel.
One issue that now remains to be solved, is how we make this scalable. By deploying two instances, heartbeats on a point-to-point channel might go to different instances, thus having incomplete registries, misleading the service in which topics are active or not. Changing this to a publish-subscribe channel does not solve the issue, as 2 instances may start processing on a pipeline simultaneously. A solution would be to use **Correlation Identifiers**, to ensure heartbeats on topic X always go to the same observer on a point-to-point channel. That way, a heartbeat of a user and a heartbeat of an ingestion processor are sent to the same observer, although they are

partitioned over our observers. Similarly, correlation identifiers are used for the start and stop command, to ensure no publish-subscribe is required for this, but only a start and stop command to the ingestion processor in charge. Then, if the same ingestion processor *does* receive 2 start commands, it can check in its own memory if the second is a duplicate. That way, the end-points are **Idempotent Receivers**, as long as the ordering of Start- and Stop-commands are preserved.
Illustration: Figure 6

5.1.4 Coupling between code and messaging system

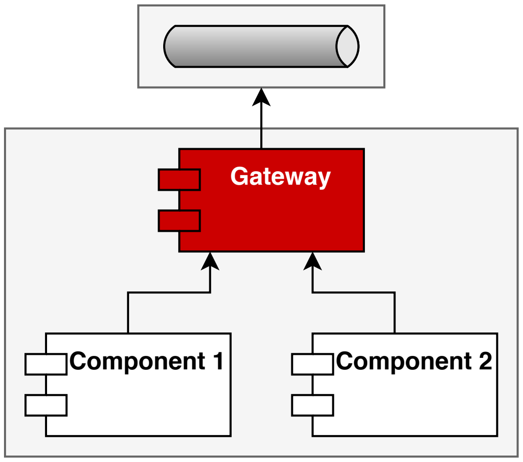


Figure 7: Gateway

Issue: For our applications to interact with the messaging system, there needs to be code. This code should be as decoupled as possible, to allow our core business logic to remain simple.
Assumptions/Constraints: Core business logic should be agnostic to the messaging implementation. Furthermore, the messaging system may be replaced by a different implementation.
Positions

- Gateway

Source: Chapter 10 [2].
Decision: In order to separate the messaging code from the business logic, senders and receivers will be placed in a separate component called the **messaging gateway**.
Illustration: Figure 7

5.1.5 Cross-application data model sharing

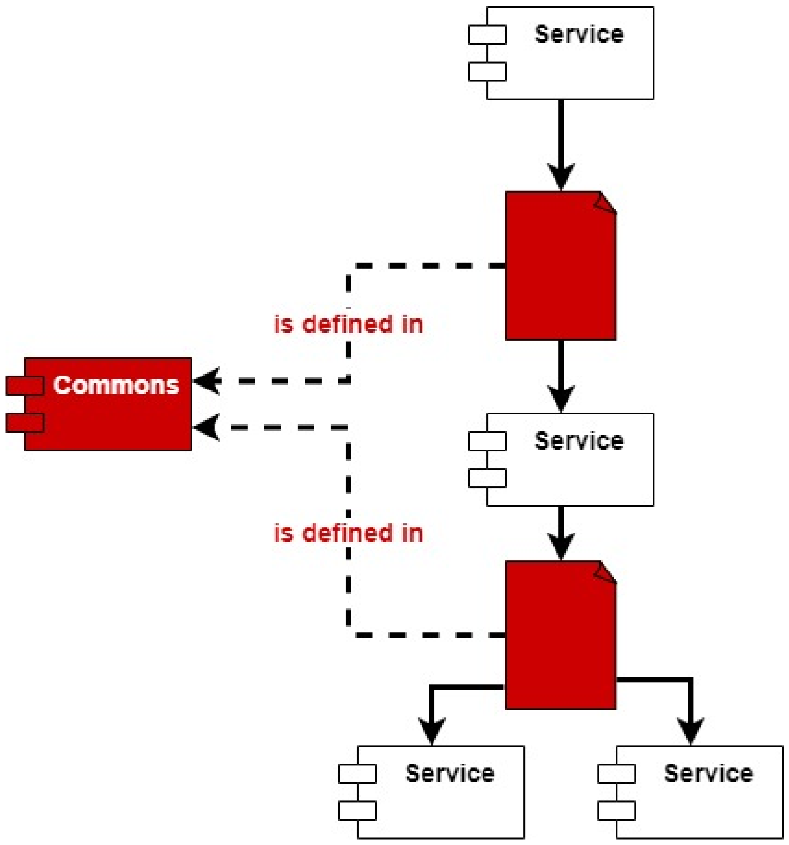


Figure 8: Canonical Data Format

Issue: Communication between applications happens through messaging. However, there needs to be consensus on the message format, otherwise they cannot be read (correctly) by others. A technique is required to facilitate this.
Assumptions/Constraints: Components communicate information. Different data formats are used throughout the application. Applications may need to modify the data format. Applications should be decoupled from each other, including the data format. Applications should allow for independent compilation.
Positions

- Canonical Data Format

Source: Chapter 8 of [2].
Decision: The most naive approach would be to have the consumer re-use the data format specified by the producer. This, however, is bad practice, because compilation of the consumer depends on the producer. Ideally, we want these to be decoupled completely. Then, one might argue, why not have the exact same data format redefined on the Consumer itself? This is easy to do and this would work, but introduces a different issue: changing the data format is difficult, as many different specifications need to be changed across applications. The solution here, is to introduce a canonical data format by creating a new library: the commons package. This package specifies data formats and are re-used for our components.
Illustration: Figure 8

5.1.6 Filtering results and removing noise

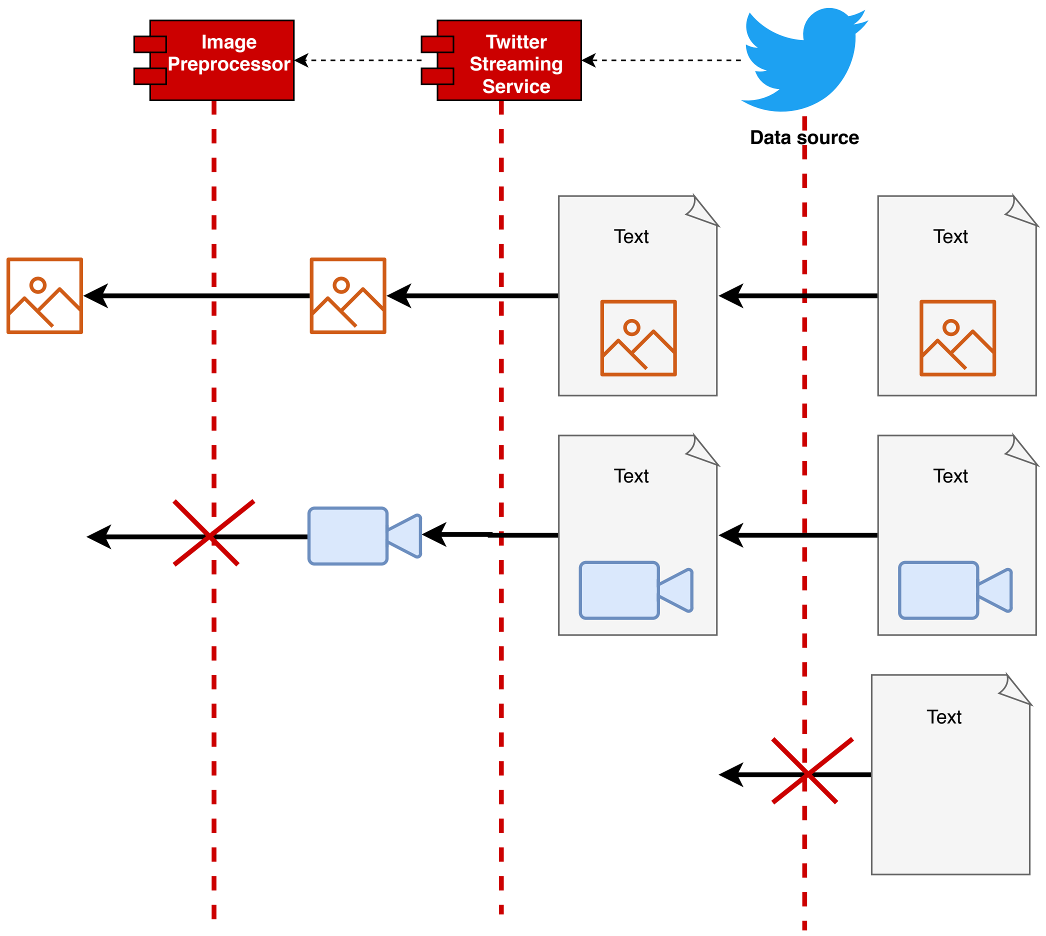


Figure 9: Content Filter

Issue: Data is being fetched from a public source, where anyone can create posts. These posts can have a lot of different formats: plain text, static images, videos, or animated GIFs. Our system will only look at static images, and thus, needs to remove unwanted data from our pipeline.
Assumptions/Constraints: N/A
Positions

- Content Filter

- Pipes and Filters

Source: Chapter 3 and Chapter 8 of [2].
Decision: By re-using the **pipes and filters** pattern discussed before, some of our applications can act as a **content filter**. First, our data source (*Twitter*) already filters which data we retrieve. This is achieved using a rule, to specify we only want tweets containing images and our hashtag. However, this also includes GIF-images. Next, the *twitter-streaming-service* maps the tweet information to a media file, and then our image preprocessor filters the media files to discard GIF-images. This is a **content filter**.
Illustration: Figure 9

5.1.7 Object Detection in images

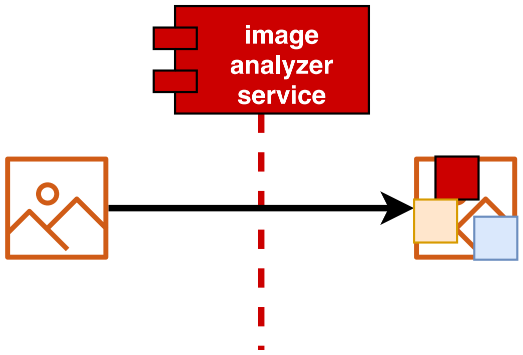


Figure 10: Content Enricher

Issue: An input image is to be analyzed for objects detected in the image. Downstream applications should receive the output: not just the image itself, but also the detection of objects.
Assumptions/Constraints: Downstream applications need to have two pieces of information: the image, as well as the detected objects.

Positions

- Message Sequence
- Correlation identifier
- Aggregator
- Composed Message Processor
- Content Enricher

Source: Chapter 5, 7 and 8 of [2].
Decision: The first option is to send the output using 2 different messages: one for the analysis, and one for the image itself. This would require a message sequence or correlation identifier to keep them together, and an aggregator to combine them. However, every single downstream application will use the image and analysis data combined, so there is no need to send them separately. Therefore, a composed message processor would have been a valid option. Since the output of the image and the analysis is always

only used for visualization purposes, we decided to enrich the original image message by drawing boxes of the identified objects on top of the original image. The image is being enriched, so that the DB adapter and frontend do not need to do the same operation again (twice). Therefore, our image analyzer service acts as a **content enricher**.
Illustration: Figure 10

5.2 Process View

In this subsection, the Process View of our system has been elaborated. This view deals with the system's dynamic aspects, explains the system processes and how they communicate, and focuses on the system's run time behavior. Here we have created the Process View based on the guidelines and notions of the 4+1 View [3]. The process view is described by several levels of abstraction, where each level is addressing different concerns. A process is a grouping of tasks that form an executable unit. Processes represent the level at which the process architecture can be tactically managed (i.e., started, recovered, reconfigured, and shut down). A task corresponds to a thread that is the building block of a process. Tasks can either be synchronous or asynchronous.
Figure 11 represents the Process View for our system. The below list outlines the processes ordered by the run sequence:

- **View Process:** UI process that sends heartbeats to initiate the streaming process and receive annotated images.
- **ActiveMQ Artemis:** The primary JMS compliant messaging queue service that acts as the backbone for our system. All messages between services are shared via this service.
- **Stream Management Service:** Receive heartbeats from the View Process with the hashtag topic name and heartbeats from the Twitter Streaming service to Start/Stop the streaming process.
- **Twitter Streaming Service:** This service fetches image URLs from tweets using the Twitter API 2 and sends heartbeats to the Stream Management Service. The retrieved image's message is then sent to Preprocessing Service.
- **Preprocessing Service:** Filter the incoming messages based on the image formats that can be used by the Image Analysis Service.
- **Image Analysis Service:** Perform image annotation and send the message to Database Service and View Process.
- **Database Service:** Store the incoming JSON in the database.

The figure also shows the message flow coded with different colors based on the different Processes used. Here, all the message flow connection is via the point to point protocol except for the connection between the Image Analysis Message Queue Runner and View Process.

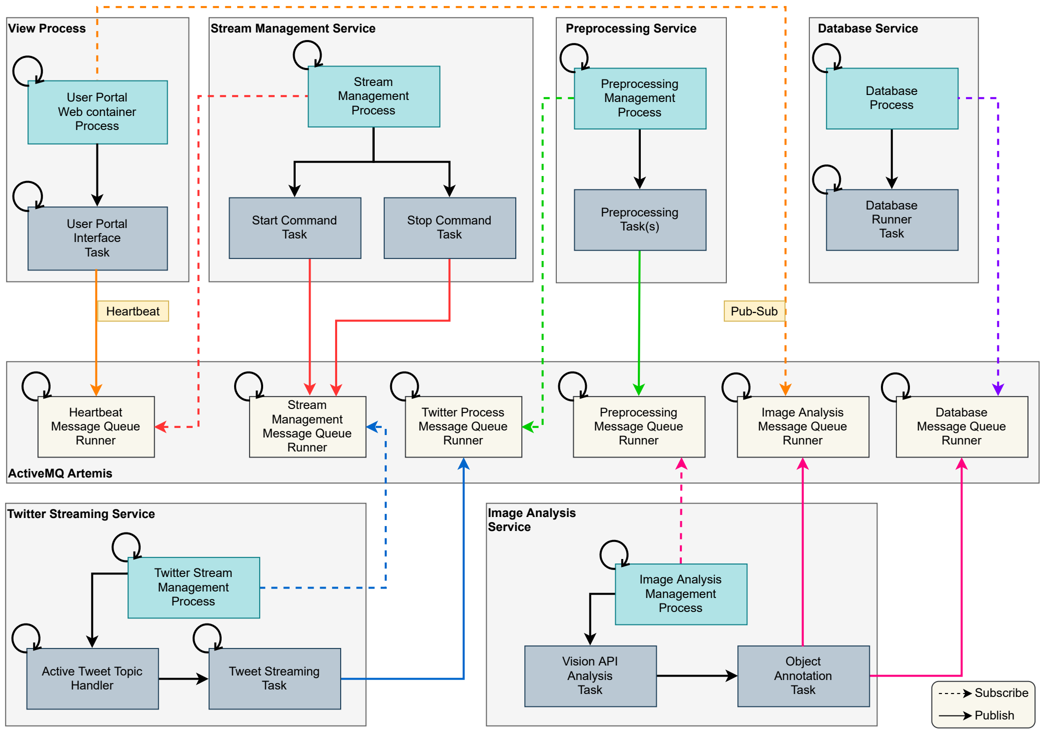
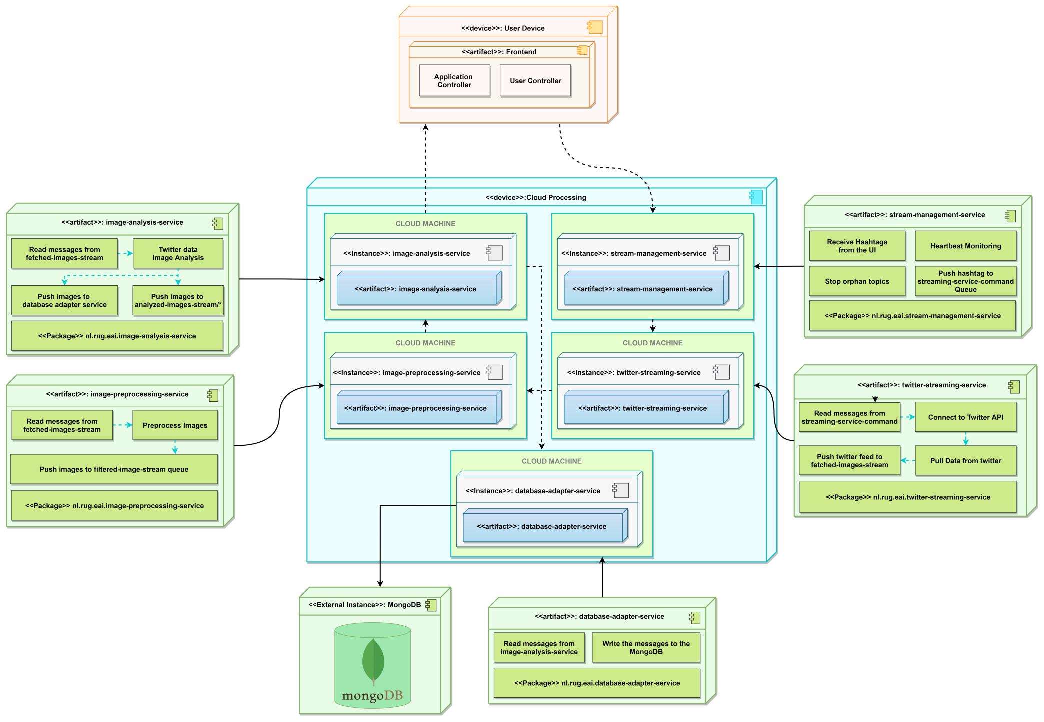

View Process
User Portal Web container Process
User Portal Interface Task
Stream Management Service
Stream Management Process
Start Command Task
Stop Command Task
Preprocessing Service
Preprocessing Management Process
Preprocessing Task(s)
Database Service
Database Process
Database Runner Task
Heartbeat
Pub-Sub
Heartbeat Message Queue Runner
Stream Management Message Queue Runner
Twitter Process Message Queue Runner
Preprocessing Message Queue Runner
Image Analysis Message Queue Runner
Database Message Queue Runner
ActiveMQ Artemis
Twitter Streaming Service
Twitter Stream Management Process
Active Tweet Topic Handler
Tweet Streaming Task
Image Analysis Service
Image Analysis Management Process
Vision API Analysis Task
Object Annotation Task
Subscribe
Publish

Figure 11: Process View

5.3 Physical view

The topology of software components on the physical layer and the physical connections between these components are elaborated in physical view. The view is an depiction of the system in the system engineer's perspective that addresses the non-function requirements of the system [3]. The figure 12 depicts the physical view of our system. The artifacts are the docker images and/or source code that are used by the worker machine on the cloud to spin up an instance that performs the relevant task and instance should always be housed within a device. The internal connections of a process are also elaborated within the artifacts. The instances or containers in this context are also connected which is displayed using the dashed arrows.



<<device>>: User Device
<<artifact>>: Frontend
Application Controller
User Controller
<<device>>:Cloud Processing
CLOUD MACHINE
<<Instance>>: image-analysis-service
<<artifact>>: image-analysis-service
CLOUD MACHINE
<<Instance>>: stream-management-service
<<artifact>>: stream-management-service
CLOUD MACHINE
<<Instance>>: image-preprocessing-service
<<artifact>>: image-preprocessing-service
CLOUD MACHINE
<<Instance>>: twitter-streaming-service
<<artifact>>: twitter-streaming-service
CLOUD MACHINE
<<Instance>>: database-adapter-service
<<artifact>>: database-adapter-service
<<artifact>>: image-analysis-service
Read messages from fetched-images-stream
Twitter data Image Analysis
Push images to database adapter service
Push images to analyzed-images-stream/*
<<Package>> nl.rug.eai.image-analysis-service
<<artifact>>: image-preprocessing-service
Read messages from fetched-images-stream
Preprocess Images
Push images to filtered-image-stream queue
<<Package>> nl.rug.eai.image-preprocessing-service
<<artifact>>: stream-management-service
Receive Hashtags from the UI
Heartbeat Monitoring
Stop orphan topics
Push hashtag to streaming-service-command Queue
<<Package>> nl.rug.eai.stream-management-service
<<artifact>>: twitter-streaming-service
Read messages from streaming-service-command
Connect to Twitter API
Push twitter feed to fetched-images-stream
Pull Data from twitter
<<Package>> nl.rug.eai.twitter-streaming-service
<<External Instance>>: MongoDB
mongoDB
<<artifact>>: database-adapter-service
Read messages from image-analysis-service
Write the messages to the MongoDB
<<Package>> nl.rug.eai.database-adapter-service

Figure 12: Physical View

6 Architectural Components and Technologies Elaborated

6.1 Architectural Diagram

In this Section we present our architectural diagram in Figure 13, which comprises of all the components that compose our application and that will be discussed further. It is composed of the Services that are explained with great detail in Section 6.2 and of the Technologies mentioned in Section 6.3.

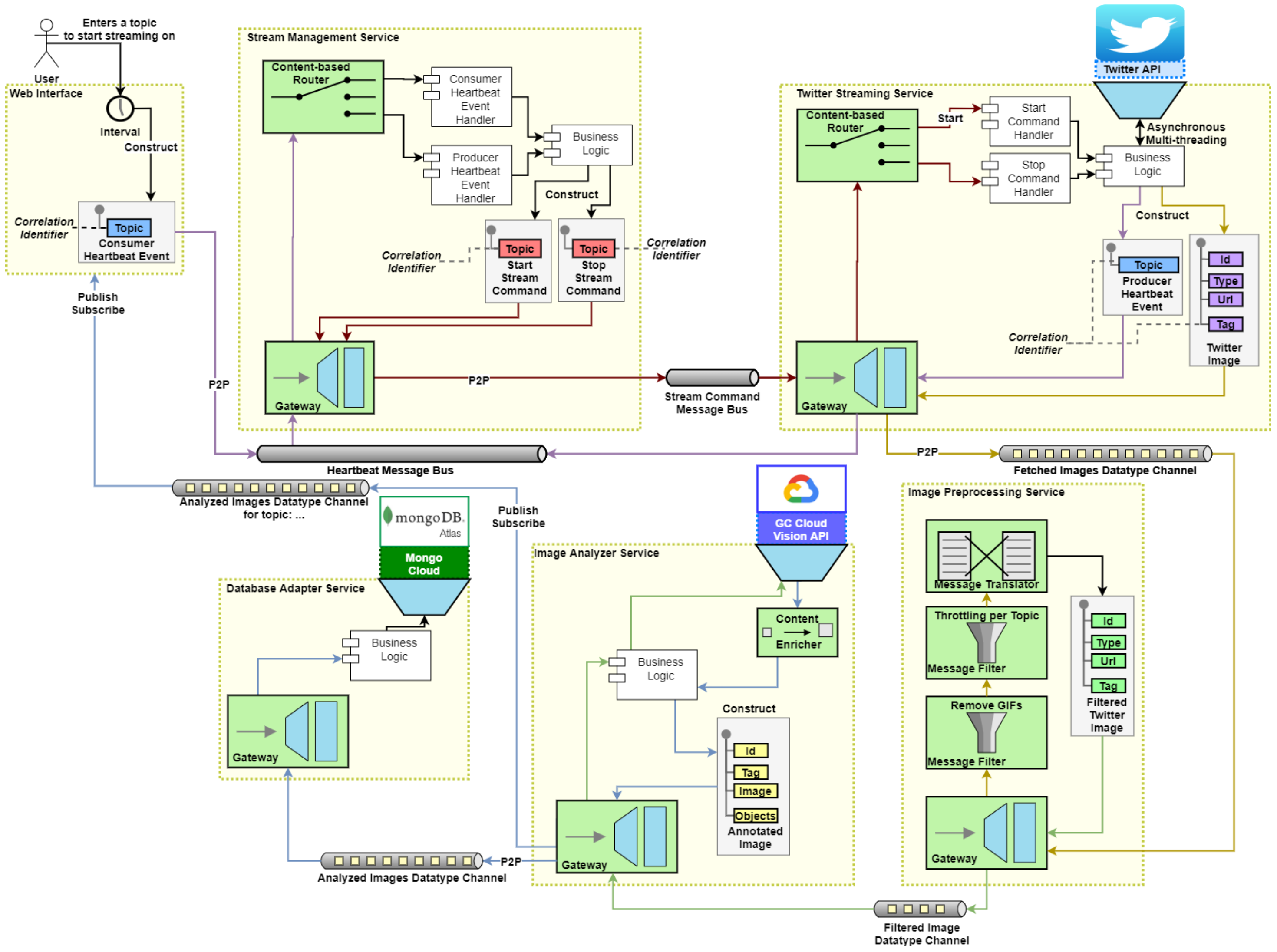


Figure 13: Architectural Diagram

6.2 Services

In this Section we will cover the Micro Services that compose our application, by highlighting their responsibilities and to which other components they connect to.

6.2.1 Frontend

Responsibilities: The frontend is responsible for receiving user input regarding which hashtag they would like our system to process. The frontend will transmit the hashtag and receive the annotated images via the Message Queues. Apache ActiveMQ Artemis supports Stomp over Web Sockets. Modern web browsers support Web Sockets that can send and receive Stomp messages from Apache ActiveMQ Artemis. So, the team decided to have the frontend to exchange these messages on a JMS topic using the STOMP protocol.
Connects to: Stream Management Service through an ActiveMQ instance

6.2.2 Stream Management Service

Responsibilities: The Stream Management Service is responsible for receiving the hashtag from the Message Queue from different topics, and starting a command for each request coming from the frontend if there are heartbeats that reach the UI. Otherwise it will create a Stop Command to save resources. This means that each different hashtag will be assigned to a specific topic.
Connects to: Twitter Streaming Service through an ActiveMQ instance

6.2.3 Twitter Streaming Service

Responsibilities: The Twitter Streaming Service is responsible for contacting the Twitter API, fetching the results for tweets of a specific hashtag that contain images, and obtaining the url of the image, which will them be fed to the Image Preprocessing Service through a Message Queue.
Connects to: Image Preprocessing Service through an ActiveMQ instance, Twitter through a REST API

6.2.4 Image Preprocessing Service

Responsibilities: The Image Preprocessing Service is responsible for dealing with image formats that require ”cleaning” before heading to the Image Analysis Service. Odd image formats (e.g. GIF) need to be dealt with since the Image Analysis Service only process one-framed images. Additionally, our system has a quota of about 1000 free image analyses on Google Cloud. Therefore, we are throttling at the image preprocessing, to only allow 1 image to go through per specified amount of seconds (15 by default). If more are fetched from twitter, they are being discarded, until the time since last passthrough is larger than 15, and a new one is let through.
Connects to: Image Analysis Service through an ActiveMQ instance

6.2.5 Image Analysis Service

Responsibilities: The Image Analysis Service is responsible for communicating with the Image Analysis Provider (Cloud Vision) to process the Twitter images. The process starts by sending the image URL to Cloud Vision and then receiving the identified objects. Next, this service annotates the images using the discovered objects and converts them to base64 string. Finally, Image Analysis Service will then pass the JSON that contains the annotated items and base64 string to two services via the ActiveMQ, the Database Adapter Service (to be stored) and the Frontend (to be displayed to the user).
Connects to: Database Adapter Service through an ActiveMQ instance, Frontend through STOMP, Cloud Vision through a REST API

6.2.6 Database Adapter Service

Responsibilities: The Database Adapter Service will be responsible for receiving the messages from the ActiveMQ instance that contains the messages flowing from the Image Analysis Service and then sending them to the Database instance to store them.
Connects to: Database

6.3 Technologies Stack

The following are the technologies used to realize our system and arguments to support our decision.

Technology	**Reasoning**
Twitter API	The API integration to fetch the twitter feed onto our system UC-4.2.
Cloud Vision API	A google service that performs object detection on the images UC-4.2 and FR-8
MongoDB	NoSql document based database that is compatible with our system's requirements FR-9.
Java Spring Boot	Native support with ActiveMQ platform, extensive documentation on messaging systems, implicitly promotes conciseness, clean and efficient coding practices.
ActiveMQ	An asynchronous messaging system that provides out of the box JMS complaint platform.
Docker and -compose	Seamless transition to cloud system, portability, implicitly encourages self sufficient modules.

6.3.1 Twitter API

By making use of Twitter's Developer API, we are able to stream tweets that contain certain parameters (we will focus on the hashtag parameter). This way whenever a user inserts a desired hashtag into our frontend we will utilize Twitter's API to listen to that specific hashtag and provide the user with information regarding all the incoming tweets that contain images and make use of that hashtag.

6.3.2 Cloud Vision API

Cloud Vision is a Google-Provided service in the form of an API that allows us to easily integrate vision detection feature, including image labeling, face and landmark detection, optical character recognition (OCR), and tagging of explicit content. We will scan all the images that we receive from Twitter using this API, and we will show the results to the user in the Frontend. The result of Cloud Vision is a JSON that contains all the information regarding a specific image.

6.3.3 Database - MongoDB

The database will be utilized as a logging system/persistence store, so that we can access all the images that were scanned, by storing the JSON outputs provided by Cloud Vision.
For our data storage we have decided to select **MongoDB** to persist the JSON's that were originated by the Cloud Vision API, so that we can keep a log of what images have been analyzed.
Since we are storing JSON's, and MongoDB is composed of a table styled database that stores records in JSON format, it seemed the most logical decision to use it.
Alternatives: PostgresSQL or Cassandra

6.3.4 Programming Language - Java Spring Boot

For our programming language, we have selected Java, by making use of Spring Boot, which is an open source Java-based framework used to create micro services.
The reason why we chose this framework comes from the previous experience some group members already had with using it, and also due to its ability to incorporate already implemented annotations and with different components which simplifies the program comprehension and also allows for an easier integration with other technologies. For example Spring Boot Integration already contains the implementations for most Enterprise Application Integration patterns, it supports JMS and offers inter-operable solutions with components like ActiveMQ which are critical for our application.
For our Frontend, we have decided to utilize the ReactJS framework for JavaScript, since it was the language the group felt more comfortable using and it provides a great UI to the users.
Alternatives: AngularJS for Frontend

6.3.5 Messaging Systems - Active MQ

For the Messaging Systems we have decided to utilize **Apache ActiveMQ™**. The reason for it was mostly the compliance with Java Message Service (JMS), since this is one of the requirements for this project. It is an open source, multi-protocol, Java-based messaging server. It is easily integrated with our micro-services architecture and since we have a frontend to display

the results, we can utilize STOMP over websockets to exchange messages to the frontend.
We have decided to go with ActiveMQ Artemis opposed to ActiveMQ "Classic" since it was developed for a more high-performance environment, which fits the goal of our application.
Alternatives: RabbitMQ using a JMS adapter

6.3.6 Containerization - Docker and -compose

For an easier deployment and in order to run our application in any system, we have opted to containerize our application using **Docker**, as it is currently the most inter-operable solution and has been used by the group in the past.
Alternatives: Virtual Box

7 Proposed Implementation and Relationship to Patterns

This section will primarily elaborate how our system tackled the use-cases that were depicted in Section 2 and some of our key design decisions that were specific to our system. Our design decisions are influenced by the KISS principle[2] and is motivated by a spring boot article on messaging systems[3].

7.1 Sender and Receiver patterns

This section will elaborate how our system handles sending and receiving messages across the modules using JMS complaint messaging queues.

7.1.1 Senders

Our system utilizes the spring boot's JmsTemplate Helper class that simplifies synchronous JMS access. As per the functional requirements FR-2 and FR-3 our system required 2 flavours of sender functions one to start the relevant processed and the other to stop the process gracefully. The sender endpoints generally implement the point-to-point messaging pattern with an exception of publisher-subscriber setup between the Image analysis service to the Front-end.

```java
@Component
public class StartCommandSender {

    @Autowired
    private JmsTemplate jmsTemplate;

    public void send(StartCommand startCommand) {
        jmsTemplate.convertAndSend(
           "streaming-service-commands",
           startCommand,
                msg -> {
                    msg.setStringProperty(
                       "JMSXGroupID",
```

[2]https://en.wikipedia.org/wiki/KISS_principle
[3]https://spring.io/guides/gs/messaging-jms/

```
                        "streamTopic=" +
                        startCommand.getTopic());
                    return msg;
                });
    }

}
```

Code snippet 1: StartCommand sender class.

```
public void convertAndSend(String destinationName,
                           final Object message,
        final MessagePostProcessor postProcessor)
```

Code snippet 2: Convert and send function.

The Code snippet 1 illustrates a simple StartCommand **point-to-point** messaging system implementation using the JmsTemplate helper class provided by the spring boot framework. The **converAndSend** function is the one responsible for converting and forwarding our java objects as messages to the queue specified. The syntax of the function is as as shown in the Code snippet 2 . The first parameter depicts to which queue the message should be sent to, followed by the message in the second parameter and a post-process operation as an anonymous function in the third parameter that is applied to the message implicitly.

```
jmsTemplate.convertAndSend(
   "streaming-service-commands", stopCommand,
        msg -> {
            msg.setStringProperty("JMSXGroupID",
               "streamTopic=" +
               stopCommand.getTopic());
            return msg;
        });
```

Code snippet 3: StopCommand sender JmsTemplate.

The JmsTemplate for a StopCommand is a shown in the Code snippet 3, it has a similar implementation as the StartCommand sender that also reuses the same messaging queue for transmitting the messages. The only difference being the **message object** that is being transmitted across the queue is different. A content based router is setup at the receiving end to handle these two types of commands.

The Start and Stop Commands are limited to Stream Management Service and Twitter Streaming Service, and the intermediate modules simply apply their transformations on the message(s) and forward the results to the next queue. Section 5.1.3 elaborates the decision behind implementing the start and stop commands for these two modules.

```java
public void send(MessageObject messageObject) {
    try {
        jmsTemplate.convertAndSend(
           "<DESTINATION-MESSAGE-QUEUE>",
           messageObject);
    } catch (Exception e) {
        log.error(e.toString());
    }

}
```

Code snippet 4: Message sender pattern.

The intermediate modules Image preprocessing service and Image analyzer service also utilize the JmsTemplate's to forward their messages. Code snippet 4 shows a simply **send** function that pushes the messages to a JMS queue. Fault tolerance has been handled in the form of try-catch blocks that will log any unforeseen invalid state(s) of the system for bug-fixing and analysis.

```java
public void send(AnnotatedImage annotatedImage) {
    log.info("Sending data on queue: " +
       annotatedImage);
    try {
        // Publish the message to the channel for
           P2P transmission to database adapter.
        jmsTemplate.convertAndSend(
           "analyzed-images-stream",
           annotatedImage);

7          // Publish the message with publisher-subscriber pattern.
8          jmsTemplate.setPubSubDomain(true);
9          jmsTemplate.convertAndSend("analyzed-images-stream/" + annotatedImage.getTag().replaceAll(" ","_"), annotatedImage);
10         jmsTemplate.setPubSubDomain(false);
11     } catch (Exception e) {
12         log.error(e.toString());
13     }
14
15 }
```

Code snippet 5: Publisher subscriber message sender pattern.

A key implementation decision was taken to allow end-users searching for the same hashtag access to the same active queues. In essence, each hashtag will spawn an unique queue that is accessible with user's who query the same hashtag. To facilitate this setup the messaging strategy between the Image analysis service to the front-end had to follow the Publisher-Subscriber messaging strategy. The Code snippet 5 depicts the sender function of the Image analysis service. Here, at line 5, a copy of the output is forwarded to the analyzed-image-stream, that will be received by the database adapter service to write an entry onto the MongoDB and on line 8-10 the publisher subscriber message forwarding is performed and the jmsTemplate is reverted back to P2P pattern to facilitate the next message publishing cycle.

7.1.2 Receivers

Our system utilizes the spring boot's @JmsListener(destination) annotations to receive the messages from a JMS queue. The receiver patterns of Stream management service and Twitter streaming service utilizes a Content Based Router pattern to handle the Start and Stop-command messages. The Image preprocess service, Image analysis service, and the Database adapters adopt a point-to-point receiver pattern. The front-end implements a subscriber pattern.

```
@Component
@Slf4j
```

```java
public class ContentBasedRouterReceiver {

    @Autowired
    StartCommandHandler startCommandHandler;

    @Autowired
    StopCommandHandler stopCommandHandler;

    @JmsListener(destination = "<DESTINATION-MESSAGE-QUEUE>")
    public void receiveGenericMessage(Message message) {
        try {
            // Parsing logic
        } catch (JMSException e) {
            e.printStackTrace();
            throw new UnsupportedOperationException("Unable to parse message to object: " + message.toString());
        }
        forwardToHandler(object);
    }

    private void forwardToHandler(Object object) {
        if(object instanceof StartCommand)
            startCommandHandler.handle((StartCommand) object);

        else if(object instanceof StopCommand)
            stopCommandHandler.handle((StopCommand) object);

        else
            throw new UnsupportedOperationException( "No handler found for " + object.getClass().toString());
31         }
32 
33 }
```

Code snippet 6: Content Based Router.

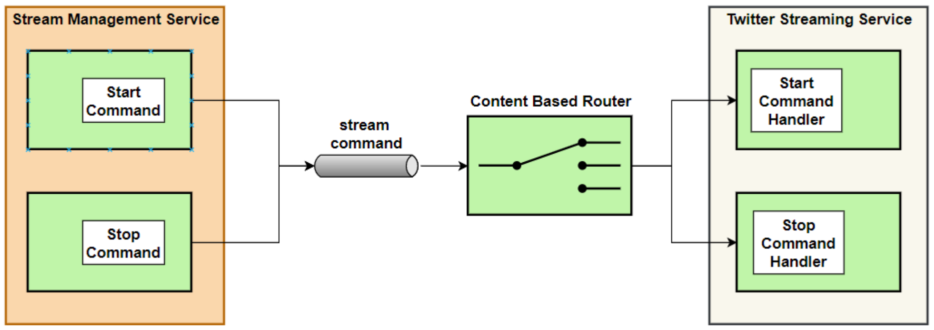


Figure 14: Content based router design.

Code snippet 6 should the template of the Content based router used in our system. Line 11 is used to configure the messaging queue the receiver function is listening to and the business logic can be categorised into two sections with sufficient fault handling techniques. The message is first parsed and later forwarded to the appropriate handlers (line 24 and 27). The handlers are logical functions that perform a transformation on the message.
Illustration: Figure 14

```
@Component
@Slf4j
public class Receiver {

    @Autowired
    MyMsgHandler myMsgHandler;

    @JmsListener(destination = "<DESTINATION-MESSAGE-QUEUE>")
    public void receiveGenericMessage(MyMessage myMessage) {
        myMsgHandler.handle(myMessage);
    }

}
```

Code snippet 7: Receiver pattern.

In instance where there is a single transformation path for the message(s), the content based filter is replaced with a message receiver as shown in the Code snippet 7. The logic to parse and transform the message is invoked from within the handler.

The front-end adopts a publisher-subscriber receiver pattern, further discussion regarding this is done in Section 7.5.

7.2 Heartbeats

In a streaming system where the queue topics are generated dynamically based on the end-user's interactions. It is essential to track the activity over these topics and release any unused resources. Heartbeats introduce this feature to our application by monitoring the topics spawned by Streaming management services. The conceptual design of this system is explained in the Section 5.1.3.

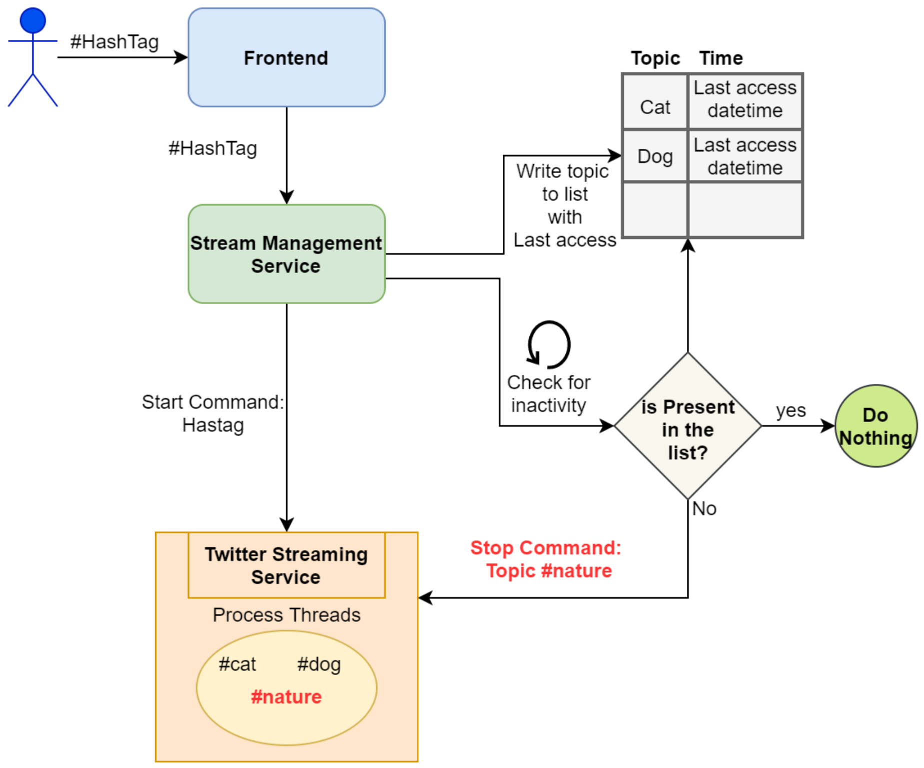


Figure 15: Heartbeats on Twitter streaming service.

Figure 15 shows the flowchart of how heartbeats are implemented on Stream management service. The end-user's input hashtag will spawn a new topic on the messaging channel. If the topic is new an entry is made to the list, and subsequent access to the same hashtag will update the **last access datetime** of that topic. The management service will repeatedly check the list for threads that have been stagnant (inactive) for an extended period of time. A Stop command is initiated to the twitter streaming service in an event where such threads are found by the system.

7.3 Fetching Twitter images

Twitter streaming service module is responsible for fetching the twitter feed using the Twitter API2 restful services. The stream management service and twitter stream service collaborate together to start and stop streaming of the hashtags through *streaming-service-commands* message channel. The module utilizes the RestTemplate helper class provided by spring boot to construct the Http request url, header and request body to perform the API call. The module is also responsible for mapping the response of the API call(s) to an appropriate object. Fault handling techniques are implemented to recover from unforeseen errors. In a rare event where there is an exception from the twitter server the system is equipped to detected and recover from this fatal error by terminating the topic that lead to this anomalous state of the system.

```
 1 private List<TweetMedia> getTweetMedia(String
      topic) {
 2     LocalDateTime previousUpdate =
          streamController.getLastProcessTime(topic);
 3     // Update the last process time in the topic
          list
 4     streamController.updateLastProcessTime(topic);
 5     try {
 6         // Fetch the new images from twitter server
              and map them to the appropriate object
              type.
 7         return
              twitterStreamingUtil.fetchNewImages(topic,
              previousUpdate);
 8     } catch (Throwable e) {
 9         // Fault handling
10         log.error("Something went wrong while
              fetching data from Twitter API.");
11         // Stop the anomalous topic
12         streamController.stop(topic);
13         return new ArrayList<>();
14     } }
```

Code snippet 8: Fetch images from twitter.

Code snippet 8 shows the orchestrator of the twitter fetch operation. The twitter module also maintains a list of topics and their last update time to track open connections that have not been updated for an extended period of time. The images are fetched and mapped to a corresponding object type on line 7 and fault tolerance is achieved with the help of try catch methods strategy.

7.4 Preprocessing and Image analysis

Cloud Vision API is an Google cloud service that provides insights from your images, in our system we are leveraging it object detection capabilities. The service provides its services to limited formats of images, whereas twitter is an source for very diverse set of multimedia documents. Our system implements a preprocessing step to exclude any multimedia image formats that is incompatible with the Cloud Vision API. We have achieved this by implementing a reverse filter that only allows image formats that are compatible with the system.

The image analysis receives the image url's from the image preprocessing service through the *filtered_image_stream* message channel in a point-to-point messaging pattern. The decision to adopt this strategy was influenced by the degree of interactions between these two services. A point-to-point setup would ensure that each message is delivered once and to one specific endpoint, also consistency in delivery was not our primary key driver since any lost images would not have any adverse effect on our systems functionality. The sender template used by the preprocessing service is as shown in Code snippet 4 and the receiver as shown in Code snippet 7.

The analyzed images are postprocessed where bounding boxes are constructed around the objects that were detected and the images are later encoded with base64 encoding for transmission via the **analyzed-images-stream** queue to the front-end for visualization in a publisher-subscriber messaging pattern. The decision to chose this pattern was influenced by the requirement that multiple users will search for the same hashtag and our system ensures that all the end-users receive a copy of the analyzed images. The image analyzer constructs dynamic channels based on the topic name for the front-end to differentiate the messages. Code snippet 5 line 6 shows how this is achieved by appending the topic name at the end of a static queue name to create a unique dynamic queue. A copy of the results is also sent to the analyzed-image-stream queue in a point-to-point connection that will be received by the database adaptor to write the results onto the MongoDb.

7.5 Front-end

Code snippet 9 shows the function that is responsible for subscribing to the messages published by the image analyzer onto the dynamic messaging queues. This is achieved by utilizing the hashtag stored in the user's instance and concatenating it with the static queue name. The code snippet is a generic subscriber function that accepts a queue name (here the dynamic queue) and processes the messages from the respective queue. Appropriate checks are also setup as shown in the snippet to avoid any unexpected state of the system.

```
// Get the dynamic queue (hashtag) from user's
   session.
subscribeFromQueue(queue) {
this.client = new Client();
this.client.configure({
  brokerURL: BROKER_URL,
  connectHeaders: {
    login: JMS_USERNAME,
    passcode: JMS_PASSWORD,
  },
  onConnect: () => {
  // Generic subscribe method to receive messages
     from queue.
    this.client.subscribe(queue, message => {
      if (message.body) {
      const text =
         JSON.parse(message.body).annotatedImage;
      const identifiedObject =
         JSON.parse(message.body).identifiedObject;
      const tag = JSON.parse(message.body).tag;
      const {
        createTweet,
        activeUser: { id: userId },
      } = this.props;
      createTweet({ userId, text, identifiedObject,
         tag });
    } else {
      console.log("Empty message");
    }
    });
  },
  // Helps during debugging, remove in production
  debug: (str) => {
    console.log(new Date(), str);
  }
});
this.client.activate();
}
```

Code snippet 9: Subscriber function on Front-end.

7.6 Database

The database adapter adopts a simple point-to-point receiver only pattern (sink). The messages written by the image analyzer on the static *analyzed-images-stream* message channel is processed and written to the MongoDB NoSql database for persistent storage.

8 Management and Fault Handling

In this Section, we explain how some fault handling and management techniques have been used/implemented in our system:

8.1 Start/Stop of Process

As discussed in Section 5.1.3, pipeline lifecycle management is one of the crucial design decisions for our system that handles the start and stop of the Twitter streaming process. When a user starts the process, the entered hashtag is attached as a payload and sent to the "Stream Management Service" with a fixed interval. This is referred to as a heartbeat in our system. Once the Twitter streaming service fetches the images from Twitter, it also sends heartbeats back to the "Stream Management Service", indicating that it is processing the desired hashtag. Here, if the front end or the Twitter streaming service crashes, our system gracefully calls the stop command and can potentially prevent situations of endless streaming of a hashtag.

8.2 Caching and Database

Performing Object detection using Cloud Vision API is charged $2.25 per 1000 images post the first 1000 images annotated, and hence dropping a processed image is undesired. Moreover, Twitter service streams live data, and therefore a fetched picture might not be available again. So, in case the client breaks down, the annotated will be persisted in the MongoDB. We have additionally implemented a logic in the frontend to cache the annotated image results. This implemented process prevents loss of data in case the browser is refreshed or if it crashes.

8.3 Frontend Auto-reconnect feature

Maintaining the connection between the client and the "Image Analyzer Service" while subscribing/listening to a Queue is vital for the frontend to receive the annotated images. The StompJs implementation used in our frontend supports the auto-reconnect feature if the WebSocket connection between the service drops and guaranteeing that no messages are lost.

8.4 Transactional Client(Not implemented)

There can be data loss when messages are transferred from one service to another because of crashes or interruptions. With Transactional Clients, fault handling is considered as the queue messages are only removed after they have been processed successfully. Even if the technique causes performance degradation, our system will not be considerably impacted as we internally perform rate limiting due to the Cloud Vision API limitation.

9 Conclusion

We have developed our "Viral Images!" system which is capable of integrating with various other systems, and to have the communication flow across the pipeline using various Enterprise Application Integration patterns. Our system offers a user the capability to listen to a specific hashtag from Twitter and see in the same dashboard the images from that specific hashtag that were created in the meantime, with the addition of annotations which are added with the aid of Google Cloud's Cloud Vision. We envisioned and achieved a scalable and fault-tolerant system, that automatically stops processing to save resources, that is utilizes both Point-to-Point and Publish-Subscribe types of messaging and that stores the images that are analyzed in a database in the cloud-based.
We believe that our system shows the capabilities of integrating several existing application to one common goal. We successfully obtain user input from the frontend, use it to obtain images from Twitter, filter and throttle the obtained content, enrich the images with annotations from Cloud Vision, which are sent to the user who requested the images via it's dashboard. Additionally, the same results are also persisted to a database for us to log the user requests.
What we extract from this project is that most Enterprise Application Integration patterns are remarkably easy to use but boost the code comprehension and allows the developers to incorporate many systems which are complex while allowing a standardized communication protocol between all parties. Crucial to the implementation is to write clean code and always think of a good separation of concerns, otherwise the code will quickly become unmanageable. We are of the opinion that our resulting system is fundamentally easy to grasp by looking at the code. Except for the Message Oriented Middleware, our system is mostly stateless. As a result, components may crash and come back online, without too much information being lost. In the worst case, an image is lost during one of the processing phases. Due to the use of correlation identifiers, our system is inherently scalable: all of our services are horizontally scalable, and allow multiple instances to be deployed. For instance, N users are able to subscribe to the same topic, while the only 1 processor works to serve all these users (instead of N processors). Fetching information from twitter is balanced over all twitter instances. If the number of twitter topics to fetch images from is greater than the number of instances, multiple threads are used to fetch from Twitter. That way, even a

single twitter-streaming-service can serve many different topics/streams. The correlation identifiers allow throttling even if multiple instances are deployed. Future work for our system could be to improve management and fault handling. Service Discovery can be used to dynamically find instances that are online, instead of statically connecting the services with each other using the docker identifiers. This would be similar as to how Web Services are discovered. We can monitor instances that are offline, and if so, remove them from the pool of instances to connect to. As for messaging, when a message may be corrupt or of an incorrect format, an invalid message channel should be used more proactively to capture and handle these messages. Another improvement is to use transactional clients, such that a message is only removed from a channel, once it has been successfully processed. That way, a message would not be lost when a service crashes during processing. Despite being interesting improvements, they are outside the scope and time limit of this project. All in all, we are very satisfied with the resulting architecture, documentation and implementation of our system.

A Role Distribution and Responsibilities

Task	**Week(s)**	**Name(s)**
Write introduction	48, 49	Mark
Write functional requirements	49	Mark
Create report structure	48	Mark
Design initial architectural vision	48	Anil
Write Use case Section 2.1	49	Anil
Write simplified architectural vision	49	Mark
Create Initial Multitier Architectural Diagram	49	Mark
Architectural Decision Sections:5.1.1, 5.1.2, 5.1.3	49	Mark
Setting up project files	49	Mark
Implementation of heartbeats	50	Mark
Implementation of Twitter streaming service	50	Swastik, Anil, Filipe, Mark
Refactoring of Twitter Streaming Service	50	Mark
Add more design decisions:5.1.4, 5.1.5, 5.1.6, 5.1.7	50	Mark
Implement Frontend + Stomp connection	50	Anil
Implement Image Analyzer Service	51	Anil
Refine Architectural Diagram	51	Filipe
Write Architectural Components	51	Filipe
Write Technology Stack	51	Filipe
Implement pre-processing service	51	Anil, Filipe, Swastik, Mark
Refactor application stack for interconnectivity	52	Mark
Implement Database Adapter Service	52	Filipe
Write Conclusion	1	Filipe, Mark
Create Presentation Slides	1	Anil, Filipe, Mark
Write Management and Fault handling Section 8	1	Anil
Write the Physical view Section 5.3	1	Swastik
Added Section 7	1	Swastik
Section 5.2 Process view	1	Anil, Swastik
Proof-Read the report	2	Anil, Filipe, Swastik, Mark

References

[1] Erich Gamma. *Design patterns: elements of reusable object-oriented software.* Pearson Education India, 1995.

[2] Gregor Hohpe and Bobby Woolf. *Enterprise integration patterns: Designing, building, and deploying messaging solutions.* Addison-Wesley Professional, 2004.

[3] Philippe B Kruchten. The 4+ 1 view model of architecture. *IEEE software*, 12(6):42–50, 1995.