UNIVERSITAT POLITÈCNICA DE CATALUNYA

BACHELOR THESIS

Big Data Analytics on Container Orchestrated Systems

Author:
Gerard Casas Saez

Supervisor:
Kenneth M. Anderson

A thesis submitted in fulfillment of the requirements for the degree of Bachelor

Developed as part of

Project EPIC Research Group
University of Colorado Boulder

July 6, 2017

Universitat Politècnica de Catalunya

Abstract

Facultat de Informatica de Barcelona

University of Colorado Boulder

Bachelor

Big Data Analytics on Container Orchestrated Systems

by Gerard Casas Saez

The Thesis Abstract is written here (and usually kept to just this page). The page is kept centered vertically so can expand into the blank space above the title too…

Contents

List of Figures

Listings

1 Introduction

Big Data Analytics is quickly becoming one of the most emergent fields of work. Data is being generated at an increasing rate. Every month, 72 Petabytes are moved around the Internet[2]. This amount is expected to grow into 232 Petabytes/month by 2021[2]. This rapid growth is probably one of the biggest challenges we face in the computer science world. As a consequence, new systems need to be designed to support all of this demand, keeping the response time as well.

Containerization is also becoming a trend nowadays. The ability to run software in separate environments has been a revolution for the tech industry. Containerization is an abstraction for the installation process that allows developers to avoid long installation processes on each system they need to run the software. It's similar to virtual machines but optimized to use system resources instead of simulated resources. Thanks to companies like Docker, containers are becoming a de facto standard for software development and deployment. It's ease of use and consistent behaviour between systems makes it attractive to developers.

However there are a few complexities: network configuration, interaction between containers, difficulties to deploy and release new versions and more. To solve this complexities, container orchestrated system arrived. They are another abstraction layer, this one goes above containers and it's in charge of its management. This systems are in charge of coordinating containers, creating them, destroying them: all the container lifecycle is controlled by the system.

As this systems are becoming more and more popular, software architecture will need to adapt to embrace the new possibilities that this systems provide. Some concepts that previously were close to impossible to do, now are available to any software architect. Applications no longer need to have an static infrastructure, container orchestrated systems can grow and decrease on demand. Systems can be splitted in small microservices, there's no need to merge everything a big monolith application anymore.

In addition, cloud providers are adding support for container orchestrated systems. This makes it even easier to deploy complex infrastructures. Also, you can design systems independently of the providers.

It also helps to find new ways to solve the problem of parallelism and scaling, making it easier to deploy new machines opens the door to new possibilities to have infrastructure on demand or to scale automatically.

In this work, I'll try to dig into the complexities of Container Orchestrated systems and microservices, focusing on a Big Data Analytics infrastructure

2 Background

Before digging into the project itself, we need to understand the problem area.

2.1 Project EPIC

EPIC (Empowering the Public with Information in Crisis) is a project at the University of Colorado Boulder. Its focused in crisis informatics, an area of study that examines how people use different technologies to interact during emergencies and their impact on emergency response.

Project EPIC has a long history on software engineering. Since 2009, this project has done research on large data storage and analysis. When large amounts of data are being collected, software engineers need to focus on structuring this data so it's easy to perform analysis . In addition, this data needs to be easily accessible for analysts.

The research group has with it's own Big Data analytics system running. Developed over the past 8 years by different software engineers researchers, the mentioned system counts with 3 different parts Collect, Analyze and Analytics. EPIC Collect is the in-house Twitter Streaming client that works 24/7 retrieving tweets from the various events that need to be monitored in real time. Since 2009 this software has been collecting tweets with an uptime of 99%. On the storage layer, we have Cassandra. This NoSQL database is focused on writes and allows a really high input. On the other side we have EPIC Analyze, built to help analysts access the data that has been collected on Cassandra. Finally EPIC Analytics is a large machine where analysts execute intensive processes over the collected data.

2.2 Containerization

We understand containerization as operating-system-level virtualization. This technology allows program to be run inside a system, isolated from each other and the host system. It works similarly than traditional virtual machines from the point of view of a program, but makes use of the underlying system resources instead of running a full powered operating system. It provides an abstraction to for programs to run in.

There are many containerization platforms around, however the most used and the one we will use for this project is Docker. This platform was originated as an internal tool in a PaaS company and later on open-sourced. Thanks to many contributions from different companies, this project grew becoming the most used containerization software. It's able to run on almost any system and has many integration. In addition, almost all container orchestrator systems support Docker containers

2.3 Container orchestration technologies

When containerisation technologies raised due to a migration to microservice architectures, big companies needed a way to manage their containers in a more friendly way. Interconnecting containers, managing their deployment, scaling easily, all this tasks were possible with containers, but they were really difficult. This is why container orchestrations systems were born. In order to manage container, the mentioned systems add an abstraction layer on top of the systems, making such tasks easier to perform.

There are a few available container orchestration systems available at the moment. The most popular ones are Kubernetes and Apache Mesos. In this project, we will further explore Kubernetes. The main reason for this is that thanks to its open source community it's easier to find tutorials and courses. In addition, there are a lot of big companies that are backing this project and contributing to it, which provides a security of a long-term support.

Google Cloud seems like the best fit to host Kubernetes as it has a managed cluster option where we don't need to worry to install and wire up the system. In addition, thanks to Google being part of the maintaining team for Kubernetes, there's a great support for Google Cloud infrastructure on Kubernetes.

2.4 Microservices architecture

Microservices is an approach to distributed systems that promote the use of small services with specific responsibilities that collaborate between them, rather than big components with a lot of responsibilities that make interaction more difficult.

In addition, thanks to new technologies like containers and container orchestrated systems, they are quickly becoming a standard on the design of big software systems. This type of architecture has been adopted by many companies as it provides a more maintainable model.

Another of the key advantages of Microservices is that it makes it easier to adopt different technologies for each component. In addition to containerization it decouples differents parts of a system systems. This allows a better optimization of technology depending on the feature that each microservice needs to offer. For example a microservice that needs to store a highly related data can make use of separate graph database, exposing a REST API, allowing for a completly abstraction. At the same time a microservice that needs to store big documents can use a document store database underneath, allowing for a better optimization.

Having small microservices do specific tasks, makes development cycles faster and more independent. Making incremental deployment way easier and less dangerous. They also make it easier to scale systems. We could individually scale parts of the system independently depending on their usage.

Microservices avoid falling in the same problems that previous service-centered architectures had, by centering on their architecture and letting the architect decide the messaging system or deployment type.

In addition microservice architecture allows for a better reliance, as each microservice is independent, if one fails, the rest can remain unaffected. This finds a perfect match with container orchestration systems, as they can abstract health checks on microservice as well deploy again microservices that failed.

In terms of designing microservice architectures there are a couple of approaches that are equally used at the moment: coreography and orchestration[5].

2.4.1 Coreography

Coreography approach centers all interaction on a publish-subscribe philsophy. Events are propagated through the system in an asyncronous basis. When something happens in a microservice, it gets published into a public queue. Other microservices can subscribe to this queue and act consequently when they receive an event.

This approach makes it the system more flexible, removing the responsability of knowing who to send messages to from the microservices. We can add more components easily without having to modify the existing ones.

However, we need extra components in order to make sure that all tasks are performed once an event is published. We can acomplish this by installing and mantaining a monitoring solution.

2.4.2 Orchestration

Orchestration centers interactions in request/response iteractions. Each microservice requests information to any other microservice. For this approach we need of a proper service discover backend. In addition if the service is down, we need to make sure that each microservice retries and is capable of failing gently.

The good thing is that we would be sure that all actions are performed by the end of an action. However if a components takes too long to answer we could break the entire action.

A more modern approach to orchestration has been developed with Service meshes. This components take the responsibility of balancing requests as well as allowing for service discovery to happen, they avoid bad crashes by retrying requests. They work by adding a sidecar to all microservices acting as proxy between the microservice and the outside network. Together with a manager that controls service discovery, load balancing and retrying counters. However these are still new technologies being developed and are still on their early releases. In addition they don't include the choreography extensibility capabilities.

2.5 Messaging systems

Messaging systems organize queues of messages produced by some microservices and notify subscribed systems that are consuming queues when they arrive. Their main objective is to can decouple components and to have a cache if the consumers can't digest all the incoming messages. This allows for more reliable systems as system don't depend on the mutual availability to pass messages as they would if they had to use other messaging system like HTTP.

There are many options used nowadays the most popular being Apache Kafka and RabbitMQ. Kafka, is a decentralized, high availability system that allows us to publish messages organized by topics. It allows for high parallelizability thanks to the partitions on topics, which lets you read and write at a higher rate in a parallel way. Each partition can have one consumer associated to it, which makes it faster to read.

Another good thing about Kafka is its persistence system. Thanks to a great integration with the kernel, it performs and persists data faster than other systems. This is due the fact that gives the responsibility to flush to disk to the kernel, taking advantage to disk page caching and memory caching implemented into current operating systems.

In addition, Kafka it's one of the most used messaging systems in the industry nowadays. Many companies use it on their production systems to decouple their systems and increase code responsability repartition. In addition it has a really strong community behind it, as well as some major technology companies like Linkedin maintaining it. Other messaging options include RabbitMQ or NATS.

Messaging systems are needed for choreography microservice architectures, as they are the communication layer between microservices.

2.6 Big Data storage systems

We will be tracking Twitter with the Streaming API. As the API is limited to provide a 10% of the total tweets generated in twitter every minute, we can estimate how many tweets will go through our system. Last report available of tweets per seconds is from 2013. The rate was 5700 tweets[4] per second on average. Which means that our system should be able to store 570 tweets every second. In addition we need a platform that scales well with our increasingly large datasets. As previously studied in the EPIC project[6], we would need to use a NoSQL database instead of a relational database to support the high throughput of tweets and make the system more scalable.

On the other hand we also need a system that allows analytical and operational queries happening at the same time. We would like to be able to analyze our data in real time without having to stop its storage. Given that some analysis may take a few minutes due to the high amount of items to analyze, we need to have a system that allows a high throughput for both parts. In this case Cassandra is the best option as the number of operations per seconds scales the most compared to other NoSQL alternatives, especially with operational and analytical workloads[3].

To store tweets we base our table structure on the current EPIC Analyze structured data storage in Cassandra as we can see on the CQL in the next page. In addition we also add an index on the event_name attribute so that we can access per event faster. We need to add the index as a lot of queries are performed per event, and so we would benefit from accessing tweets from each event in the fastest way.

```
CREATE TABLE twitter_analytics.tweet (
    id uuid,
    t_id text,
    event_kw text,
    event_name text,
    hashtags list<text>,
    media_url text,
    t_coordinates text,
    t_created_at timestamp,
    t_favorite_count int,
    t_favorited boolean,
    t_geo text,
    t_is_a_retweet boolean,
    t_lang text,
    t_retweet_count int,
    t_retweeted boolean,
    t_text text,
    u_created_at timestamp,
    u_description text,
    u_favourites_count int,
    u_followers_count int,
    u_friends_count int,
    u_geo_enabled boolean,
    u_id text,
    u_lang text,
    u_listed_count int,
    u_location text,
    u_name text,
    u_screen_name text,
    u_statuses_count int,
    u_time_zone text,
    u_url text,
    u_utc_offset int,
    um_id text,
    um_name text,
    um_screen_name text,
    urls list<text>,
    PRIMARY KEY (id, t_id))
```

Listing 2.1 Tweets CQL table script

2.7 Software development in microservices architectures

Thanks to container-orchestrated systems and the popularization of container systems, software development is easier. With microservices, focus moves out of one component software architecture to organizing the whole system. Now, we can focus on packaging alone components instead of having to packaging all of them at once, which makes the development more flexible. This would allow for a more continuous development in the system allowing for a better performance optimization, and an easier way to keep the system updated with the last set of technologies.

3 Development

3.1 Problem Statement

The goal of this project is to create a system with a better scalability and reliability compared to the existing EPIC infrastructure using a container-orchestrated infrastructure while improving our development cycle by switching to a microservices architecture. In addition to improve real time query flexibility and overall performance.

3.2 Approach

Container orchestrated systems help deploy microservices architectures as well as abstract their physical structure. They add a logical abstraction on top of the physical structure allows to work independently from the underneath infrastructure, which makes it easier to move between different cloud infrastructures.

In addition, microservice architectures make systems more flexible as they decouple between functionalities, making it easier and more cost efficient to replace tools and evolve the system. We will take an asynchronous/choreography approach for microservices with message queuing systems. This approach decreases coupling between components increasing the capability to add new components without having to change any configuration.

We will design the system to fit and take advantage of container orchestrated technology. This systems make it easy to create infrastructure on demand. Which in turn would allow for a better optimization of resources. One improvement of the "on demand" feature, is the ability to modify infrastructure on state change. Which in that sense would make possible to remove the state from the designed microservices, make them unaware of the state changes. Thanks to this capability we can design microservices that are completely stateless, improving the performance by removing responsibilities and making their development and maintenance easier.

As we have previously seen, the current EPIC infrastructure use cases can be summarized in 4 points:

- Collect tweets in real time
- Classify collected tweets in real time
- Modify classification in real time
- Analyze collected data

In that way, we first need a place to store tweets. To store tweets we use Cassandra as it's the one used by the current EPIC infrastructure. Currently Cassandra is the best option thanks to it's write perfomance, high scalability and great concurrency. In addition, it's backed by DataStax a private company that has written a high amount of documentation in how to optimize Cassandra deployments.

To collect the data in real time, we need a microservice that connects to the public Twitter API and requests tweets. For that we use the Streaming API. We create a specific microservice for this. In order to remember it later, we will call it Twitter tracker. It will act as a gateway pulling data from the public API and passing it to the messaging system to be processed. We just want to get the tracked tweets into the system.

To classify the incoming data, we can take advantage of the orchestration system and create an instance for each category we need to store. We plug each instance into the queue created by Twitter tracker and make that each instance decides if each message is part of their own category or not. If the answer is positive, this microservice will store the tweet as well as normalize it. This way, we have normalized tweets which in turn will make it easier to analyze the data that we have. We will call this microservice Tweet normalizer. For each category we will have an instance running. As we previously stated, this makes the microservice way easier to develop.

Next, we need to modify and create the classification configuration in real time. To do so, we would need a couple of components:

- *Event Manager UI*: Let's analysts manage categories/events adding keywords, toggling tracking and more. This microservice would not manage anything from the system, it would be completely unaware of the surrounding system where it is running and what other microservices exist. Finally, any change performed in the state will be broadcasted through a messaging system so that any other microservice can act upon the change. Messages should contain full state instead of incremental states. This is to ensure that state storage responsibility is not replicated in differents components.
- *Infrastructure Controller*: Subscribes to the messages generated for events and modifies the current infrastructure creating or destroying instances of tweet normalizer or updating *Twitter tracker* in order to match the change. This microservice needs to be aware of the orchestration system where its running. However, thanks to the *Event Manager UI* sending messages with full state, we can make this component totally stateless. In addition, the responsibility of detecting what changes need to be performed in the infrastructure can stay into the container orchestrated system. So this microservice responsibility stays limited to translating changes in state to changes in infrastructure

Finally, Analysts want to access the collected data and perform analysis on top of it. Cassandra is not really prepared to answer specific queries from analysts. Its CQL interface is great for small queries but doesn't allow for high complexity. In addition, it was never intended to be used as an analysis tool. So we choose a different tool that it's optimized to read and process large datasets: Spark. It is a well known big data processing tool, and it has a connector for Cassandra that takes advantage of the cluster data locality to perform tasks. This way we take the read aspect of the cluster away from Cassandra and give to a tool that has been optimized for this task.In order to make it easier for analysts to access and perform analysis, we will add a web Notebook UI to give analysts an easy way to execute queries on Spark with a web UI.

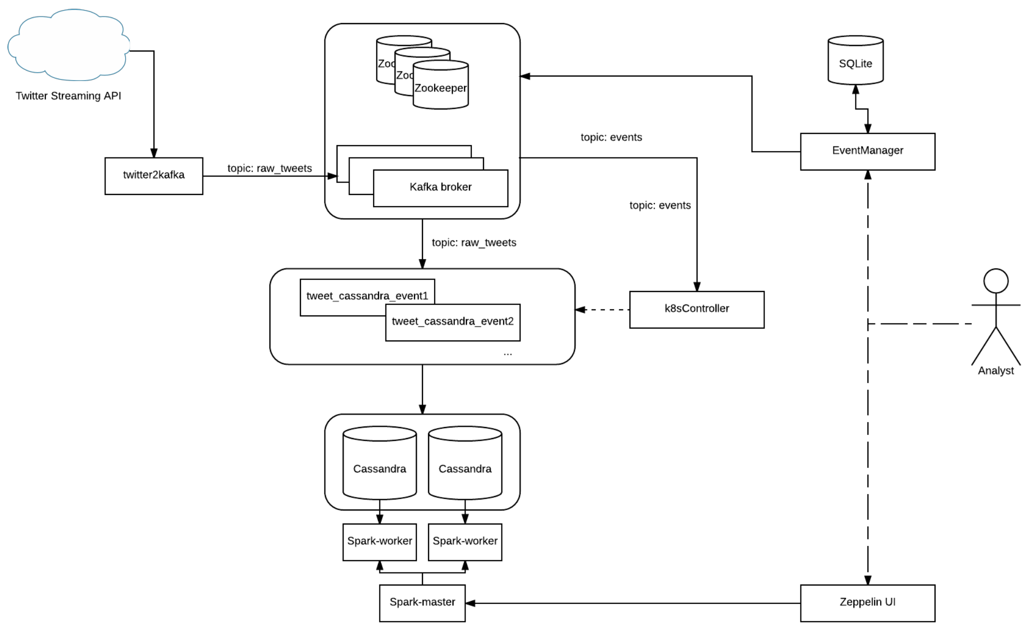
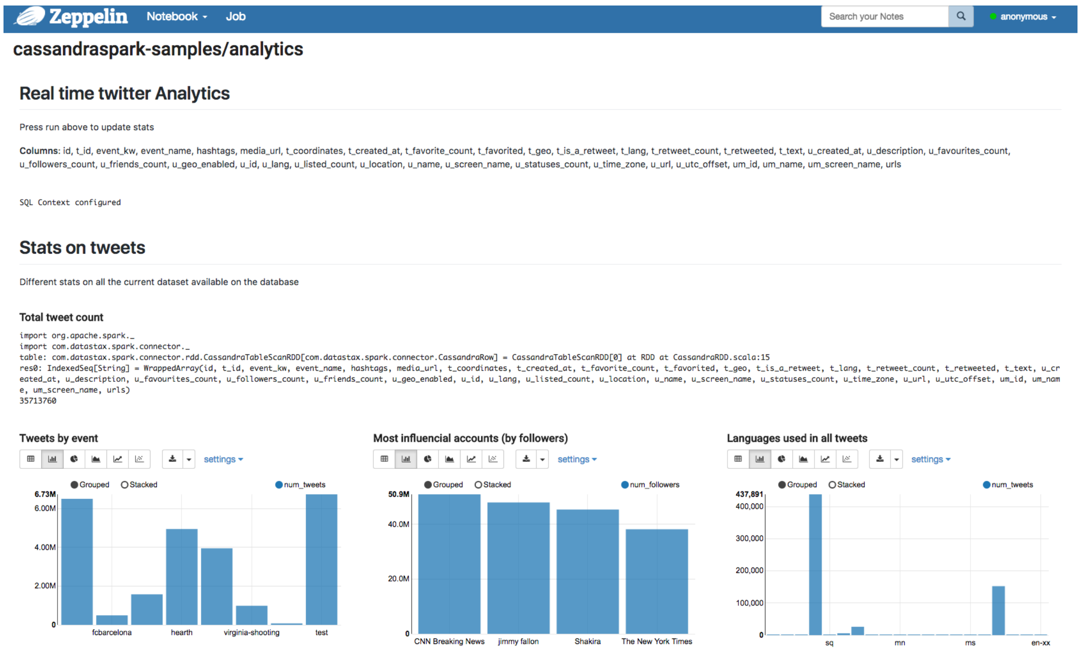


FIGURE 3.1: Proposed system architecture



cassandraspark-samples/analytics

Real time twitter Analytics

Press run above to update stats

Columns: id, t_id, event_kw, event_name, hashtags, media_url, t_coordinates, t_created_at, t_favorite_count, t_favorited, t_geo, t_is_a_retweet, t_lang, t_retweet_count, t_retweeted, t_text, u_created_at, u_description, u_favourites_count, u_followers_count, u_friends_count, u_geo_enabled, u_id, u_lang, u_listed_count, u_location, u_name, u_screen_name, u_statuses_count, u_time_zone, u_url, u_utc_offset, um_id, um_name, um_screen_name, urls

```
SQL Context configured
```

Stats on tweets

Different stats on all the current dataset available on the database

Total tweet count

```
import org.apache.spark._
import com.datastax.spark.connector._
table: com.datastax.spark.connector.rdd.CassandraTableScanRDD[com.datastax.spark.connector.CassandraRow] = CassandraTableScanRDD[0] at RDD at CassandraRDD.scala:15
res0: IndexedSeq[String] = WrappedArray(id, t_id, event_kw, event_name, hashtags, media_url, t_coordinates, t_created_at, t_favorite_count, t_favorited, t_geo, t_is_a_retweet, t_lang, t_retweet_count, t_retweeted, t_text, u_cr
eated_at, u_description, u_favourites_count, u_followers_count, u_friends_count, u_geo_enabled, u_id, u_lang, u_listed_count, u_location, u_name, u_screen_name, u_statuses_count, u_time_zone, u_url, u_utc_offset, um_id, um_nam
e, um_screen_name, urls)
35713760
```

FIGURE 3.2: Zeppelin server with some visualization from the dataset

3.3 Implementation

To prove that the specified architecture is feasible, I developed a working prototype of the system.

We started with 2 nodes of Cassandra to store the tweets with Google Cloud disks attached to keep data persistent. In order to deploy this, we use stateful sets, Kubernetes objects designed to support stateful components and allow scalability. Persistent volumes can be attached to any node to keep data between restarts keeping a naming consistency. They can be used for stateful scalable apps like distributed databases. They provide an easy way to add or removed nodes from the set, allowing for a controlled scalability. In our case we could scale Cassandra dynamically to increase storage space and/or performance by making the cluster more parallel.

In addition, we add a spark workers to each instance of Cassandra to increase and ensure data locality when running Spark jobs. This way, spark works with local data from their correspondent Cassandra node. All of this workers are coordinated by a spark master node that is in charge of planning and distributing the work between each worker.

To interact with the Spark cluster, we deploy a Zeppelin notebook container. We tweaked this container to interact efficiently with our Spark and Cassandra stack by adding a default configuration to use the cassandra-spark library. Notebooks makes the system more accessible by abstracting how to execute scripts in Spark and allowing to present results in an easy format of a website.

We also need some custom microservices for the designed solution. At the beginning all microservices were implemented in Python, basically because it's easy and fast to prototype, and it also let's use iterate faster. The goal was to keep all microservices short and simple in order to make their maintenance easy, enabling developers to maintain and deploy changes faster. Also, keeping them small makes it easy if any microservice needs to be rewritten in another language because python is not fast enough. Apart from implementing them, we also needed to containerize them to make deployment in the orchestration server easier.

For the collection pipe, we divided the process to support better scalability and avoid high incoming tweets to collapse the system. Between parts we use Kafka a high reliable messaging system. Thanks to this decoupling we could also plug other backend to analyze tweets in real time.

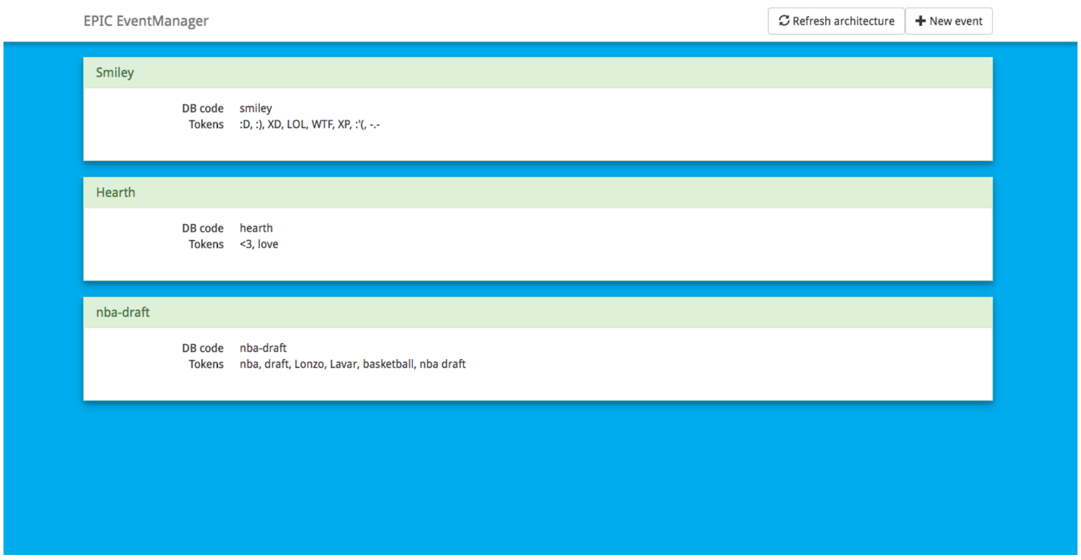
Finally, to make the system work we need to add a few custom components. The following components were implemented specially to work on the described environment.

- *Tweet normalizer*: Fast classifier and tweet data normalizer that receives tweets from the raw_tweets pipe and stores them on Cassandra. This microservice is designed to only keep track of one event. The tracked event information is sent to the microservice on start through environment variables defined with Kubernetes. To increase this microservice performance, some C based libraries replaced some regular Python libraries for intensive task as json encoding and loading.
- *Twitter tracker*: Twitter Streaming API client. Connects to the public Streaming API to fetch tweets and send them to the raw_tweets pipe. Written initially on Python and re-written in Go afterwards to increase throughput and avoid some python errors that appeared after a couple of days running.
- *Event Manager UI*: Stateful django web application for analysts to interact with. Enables CRUD actions on events. Any change on state is later broadcasted to the rest of the system through Kafka. It keeps track of changes made on events to facilitate collaboration between analysts. SQLite as a file is used for storing data and a Google Cloud disk is used to allow saving state between container restarts.
- *Infrastructure controller*: In this case this is a Kubernetes controller, written in python using a python client library. It subscribes to the Kafka topic and receives events generated by the event manager UI. It applies changes to the infrastructure incrementally to meet the changes performed in the UI by the analysts. It's completely stateless, all it does is act upon what is received.

3.3.1 Deployment

To deploy the system, we create a 4 node cluster on Google Container engine. Each node having 1 virtual CPU and 4Gb or RAM. This is the major factor limiting our deployment. We also limit each Cassandra node to 1 GB of RAM for the Cassandra part. We won't limit Spark, as we want it to use as much resources as we have available.

To start we need to deploy the basic components driving the infrastructure. Those are Kafka and Cassandra. In our case we are running Kafka with 2 brokers and Cassandra with 3 nodes each one running Spark as a sidecar. After this we need to deploy the frontend components. One the analytics side we deploy Zeppelin, and on the collecting side we deploy the EventManager. The final part is the Kubernetes



Smiley

DB code	smiley
Tokens	:D, :), XD, LOL, WTF, XP, :'(, -.-

Hearth

DB code	hearth
Tokens	<3, love

nba-draft

DB code	nba-draft
Tokens	nba, draft, Lonzo, Lavar, basketball, nba draft

FIGURE 3.3: Event manager UI

controller. Once deployed the rest of the infrastructure will be deployed when the state requires it.

Tracking will start once we add an event in the EventManager UI. After the creation action, the UI will send a message through Kafka saying that this event was created. Kubernetes Controller will collect this message and update the twitter tracker and will create a new microservice to normalize and classify tweets for the provided event.

4 Evaluation

In order to evaluate this project we are going to compare the proposed architecture with the existing EPIC architecture in reliability and scalability as well as performance and real time delivery as proposed in the problem statement. We will also evaluate manteinability and future software development.

4.1 Reliability

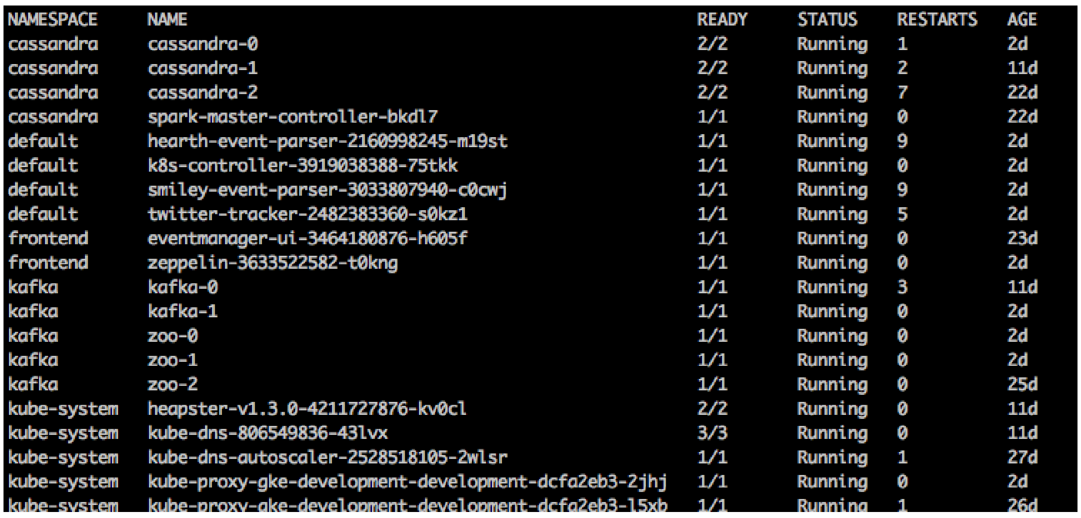
To analyze reliability between systems we will analyze how well each system recovers if a components fails. We are interested to see how much time does it take to have the system recover if something failed. In addition we want to check what mechanisms do we have in both cases to check the health of each component. This would be useful to restart parts that have stopped working.

4.1.1 Current infrastructure

Reliability on the existing EPIC infrastructure needs to be studied separately between EPIC Collect and EPIC Analyze. Collect is designed to be highly available. Using different techniques like cron jobs that check the state and other. EPIC Collect is a really reliable system. It has been available 99% of the time since 2012, making it really reliable for analysts to trust that their tweets are being collected. In the worst case scenario, EPIC Collect takes X seconds to recover. This is due the fact that the cron job that checks that EPIC Collect is running is scheduled every 5 seconds.

EPIC Analyze doesn't have the same availability. The system has suffered multiples upgrades and depends on different pieces that can fail. Analyze depends on a separated Cassandra with Solr as well as a Redis cluster. If any of those two were to fail, the system would go down, being completely unusable. There's no automatic recovery procedure.

In conclusion, the current EPIC infrastructure depends of what secure layers developers have added to their programs, which makes it hard to develop and costly to keep it reliable. In addition, the deployment of the system is manual, which could involve human errors on deployment that could cause future instability of the system. This could be caused because components overlap each other RAM usage causing a full node of the cluster to go down.



```
NAMESPACE    NAME                                                 READY   STATUS    RESTARTS   AGE
cassandra    cassandra-0                                          2/2     Running   1          2d
cassandra    cassandra-1                                          2/2     Running   2          11d
cassandra    cassandra-2                                          2/2     Running   7          22d
cassandra    spark-master-controller-bkdl7                        1/1     Running   0          22d
default      hearth-event-parser-2160998245-m19st                 1/1     Running   9          2d
default      k8s-controller-3919038388-75tkk                      1/1     Running   0          2d
default      smiley-event-parser-3033807940-c0cwj                 1/1     Running   9          2d
default      twitter-tracker-2482383360-s0kz1                     1/1     Running   5          2d
frontend     eventmanager-ui-3464180876-h605f                     1/1     Running   0          23d
frontend     zeppelin-3633522582-t0kng                            1/1     Running   0          2d
kafka        kafka-0                                              1/1     Running   3          11d
kafka        kafka-1                                              1/1     Running   0          2d
kafka        zoo-0                                                1/1     Running   0          2d
kafka        zoo-1                                                1/1     Running   0          2d
kafka        zoo-2                                                1/1     Running   0          25d
kube-system  heapster-v1.3.0-4211727876-kv0cl                     2/2     Running   0          11d
kube-system  kube-dns-806549836-43lvx                             3/3     Running   0          11d
kube-system  kube-dns-autoscaler-2528518105-2wlsr                 1/1     Running   1          27d
kube-system  kube-proxy-gke-development-development-dcfa2eb3-2jhj 1/1     Running   0          2d
kube-system  kube-proxy-gke-development-development-dcfa2eb3-l5xb 1/1     Running   1          26d
```

FIGURE 4.1: Status instances after 11 days

4.1.2 Proposed infrastructure

In this case, reliability is a responsibility from the container orchestrated system. In our case, Kubernetes includes a controller manager on the master node. This component of the cluster keeps track of the status of each one of the containers running. The controller manager makes sure that a certain amount of replicas are up and running. If any container stops running at some point, the controller manager will make sure to schedule a new container and run it on an available node. This check is done on demand when there are new deployments requested or whenever a node notifies that a container is down.

In addition to the controller manager, we also have the scheduler. This component makes sure that new container are scheduled in the best node possible. Kubernetes allows you to configure the amount of memory and cpu usage permitted by container, allowing the scheduler to find the best fit for each deployment request.

Thanks to this deployment automation, node failures can be avoided easily as well as resources get a more optimized usage than if doing this process manually. Finally something else that Kubernetes allows to do natively is to rollout new versions of a service with no downtime. To do so the controller manager deploys the new version before removing the previous. This is really useful for example in the Twitter tracker module, as this way all tweets get tracked even when the keywords change. In this case, the controller manager deploys a new version with the new keywords updated. Once it starts to run, it kills the previous version, which means that there's always at least one version running and tracking all tweets. This ensures that we can get all tweets.

To test it, we left the system tracking for 11 days with no interaction from the user. As we can observe in Figure 4.1, each instance has restarted many times. This could be caused when the system reached peaks of load. With the limited amount of resources availables, Kubernetes has to kill some containers to avoid the whole system collapsing. Note that some instances say that have been existing for less than 2 days, this is due the fact that Kubernetes will restart the count if it needs to do a hard restart.

4.2 Scalability

We want a system that adapts to our needs, so we want a high scalability for it. We want to avoid wasting resources unnecessarily as well, so we want to be able to fit our available infrastructure. To analyze this we will see how easy is to scale each system and how scaling affects performance in either infrastructure.

4.2.1 Current infrastructure

The current infrastructure is not easy to scale. If we need to scale EPIC Collect, we would need to run manually a new instance, add cron jobs as well as add new Twitter credentials for it. This is one of the reason why EPIC Collect doesn't do data normalization, if we added data normalization to the current EPIC Collect then on traffic spikes the system could be overloaded and would not be able to scale during high demand tweet incoming.

Regarding storage, scalability is high thanks to Cassandra. Adding new nodes to the cluster it's a pretty straight forward process. However regarding it will be difficult to add more Cassandra nodes than machines we have available, due to the fact that we will need to add more open ports to the outside of the machine. This is a small step back, as it's not really complex.

Regarding EPIC Analyze, the only thing we could scale, apart from storage, would be the frontend module. However that would mean adding a standalone load balancer in front of the service, which makes it a complex process if done manually and adds more complexity to maintain it. Also, we would need to add an external component to keep session information consistent across all the replicas, even though we could reuse the current Redis cluster to do so. The application would still need a refactor either way.

In conclusion, it's not really easy to scale the current infrastructure.

4.2.2 Proposed infrastructure

On the other side, the proposed architecture has a better and easy way to scale thanks to Kubernetes.

When deploying a service we can specify how many replicas we want to deploy. The controller manager will try to schedule as many replicas as it has been specified if there are enough resources. In addition, we tried to approach the design of the different components in a stateless way so that deploying new replicas would not require any additional change in the components. To achieve this, we moved the state to the infrastructure. The components are unaware of the state of the system, which makes it easy to add replicas. When added they do their task independently.

We could also easily automate scalability for each component depending on their CPU usage. If a certain container is in 90% of their CPU usage, we can make that the system increments the number of replicas. This way the underlying resources can be administered in regards of the current needs of the system.

In terms of time to scale, it's a matter to send the changes to Kubernetes. This can be done with a single command thanks to the command line tool kubectl or using

```
val table = sc.cassandraTable("twitter_analytics", "tweet")
val words = table.select("t_text").flatMap(l => l.getString("t_text").
    split(" "))
        .map(word => (word.toLowerCase,1)).reduceByKey(_ + _).map(_.swap)
            .sortByKey(false,1)
```

Listing 4.1 WordCount Spark script

the web interface for Kubernetes. Scaling is an integrated tool into container orchestrated systems, specially Kubernetes.

4.3 Performance

In order to analyze performance between both systems we need to take into account resources availables. All the analysis has been performed in the previously deployed 4 node cluster, each one with 4Gb or RAM and 1 virtual CPU. We will measure time to execute a word count script for a certain number of collected tweets. We chose word count as it's a highly a parallelizable program.

Regarding performance we need to take into account different aspects. The current EPIC architecture performs really well when asking for some very specific queries. However, if we want more general queries or complex ones we would have to download the dataset to work with it with local tools, which makes analysis way slower.

The goal for the architecture should be to give tools for analysts to develop their work without having to wonder how it was implemented. For this reason I focused on delivering a query flexible system that let's query data in different ways.

In order to analyze the performance in an overall way, we will take an example of dataset collected in the current infrastructure and one in a WordCount exercise. I selected WordCount as it's a simple to describe data analysis and it has a high degree of parallelization. This way we can see how each system takes advantages of the distributed system.

In the new infrastructure we can use Spark running on top of Cassandra. This work should be optimized thanks to having a spark worker on each cassandra node, minimizing all network traffic by using data locality.

To check how the query is performed we will look into the debug string from Spark. This is string is used to check how Spark will execute a query into its cluster. Each indentation is a map stage, and each + is a shuffle phase. As we can see in Listing 4.2, Spark is not doing any shuffles until the reduce stage, meaning that it's optimizing the query to take advantage of data locality by executing maps on the same node that the data is loaded to.

As we can see in Figure 4.2 the Word Count script performance is not linear, but it performs better with a bigger amount of tweets to analyze. This could be due the fact that the bigger value is being accessed as a total, therefore avoiding accessing indexes. This would make it quicker. We would need more results in order to analyze if this is true or not. However due to a timeout on reaad access to disk this can't

```
(1) ShuffledRDD[112] at sortByKey at <console>:31 []
+-(176) MapPartitionsRDD[111] at map at <console>:31 []
    |   ShuffledRDD[110] at reduceByKey at <console>:31 []
    +-(176) MapPartitionsRDD[109] at map at <console>:31 []
        |   MapPartitionsRDD[108] at flatMap at <console>:31 []
        |   CassandraTableScanRDD[107] at RDD at CassandraRDD.scala:15
            []
```

Listing 4.2 Debug string rdd

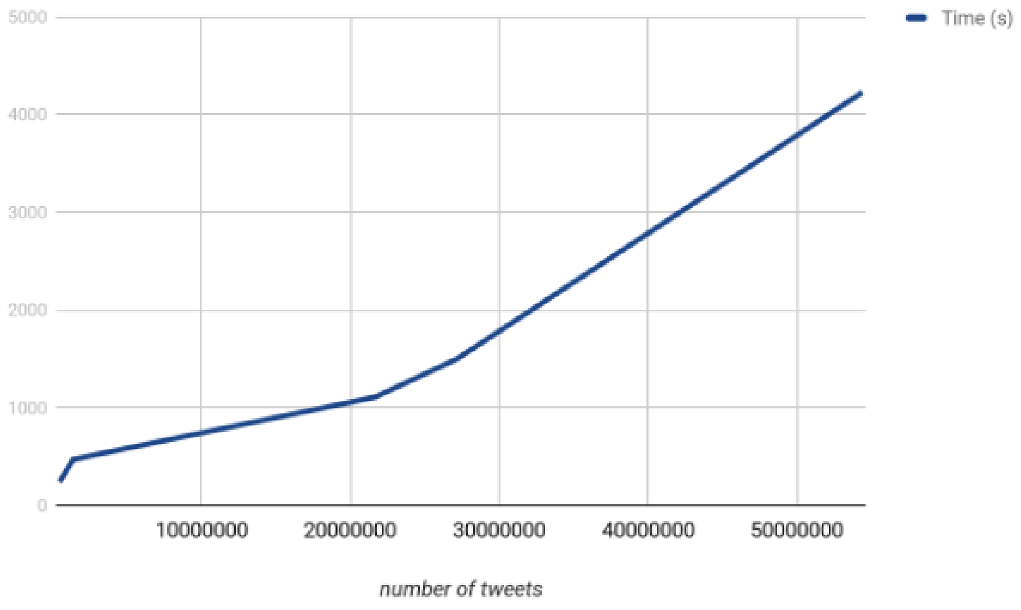


FIGURE 4.2: Plot on wordcount time for dataset size

be performed. This is caused due to having really big indexes and big partitions on Cassandra.

To solve this tradeoff, we could move the index to primary key, adding an incrementing partition key to balance distribution between nodes similarly as random key. This is inspired to the column family distribution proposed in the work by Ahmet Arif, but using structured rows for each tweet instead of multiple JSON objects on each column. However due to the work being more focused on the container infrastructure part and concept development, this could not be performed during this thesis work.

On the other side, the current EPIC infrastructure is mainly oriented to collect tweets. Analysis are performed in dedicated machines that have a high number of cores and high RAM memory. In EPIC, mainly most of this analysis are performed in EPIC Analytics. However we need to take into account that due to the structure of the current system, batch analytics are performed after data has been extracted from Cassandra and therefore we can't fully compare both system in terms of time

Number of tweets	Time (s)
490199	238
1400884	469
21680851	1107
27199614	1500
54395957	4228

TABLE 4.1: Number of tweets in dataset vs seconds took to perform word count

to obtain a result since tweet collection. The current EPIC infrastructure is prepared to answer queries extracting data and running batch analysis.

To solve this tradeoff, we could move the index to primary key, adding an incrementing partition key to balance distribution between nodes similarly as random key. This is inspired to the column family distribution proposed in the work by Ahmet Arif[1], but using structured rows for each tweet instead of multiple JSON objects on each column. However due to the work being more focused on the container infrastructure part and concept development, this could not be performed during this thesis work.

On the other side, the current EPIC infrastructure is mainly oriented to collect tweets. Analysis are performed in dedicated machines that have a high number of cores and high RAM memory. In EPIC, mainly most of this analysis are performed in EPIC Analytics. However we need to take into account that due to the structure of the current system, batch analytics are performed after data has been extracted from Cassandra and therefore we can't fully compare both system in terms of time to obtain a result since tweet collection. The current EPIC infrastructure is prepared to answer queries extracting data and running batch analysis.

4.4 Real time analytics

Real-time analysis has been approached before in the Project EPIC[1]. However the previously mentioned approaches have a limited number of queries that can be answered in real time. Adding new queries would mean having to refactor the code and deploy again. The data used would also start to be collected once it's deployed. Even though this approach gives result in a really low time, it lacks flexibility when it comes to custom queries. If analysts want custom queries, they would have to download the datasets and run their own custom analysis. This approach can be difficult, we need to predict what are the queries that we will need to answer ahead of time and sometimes this can be close to impossible.

For the new system, we took a more flexible approach. In order to increase flexibility, we sacrifice response time. Queries can be run on Spark while data is being collected, which makes their results near-real-time. We have a delay between when we start the query and when it ends, meaning that new data may have arrived during the query execution may have not been taken into account.

This approach may seem as a lost in performance. However if analysts need custom analysis, this approach grants them a faster option. In addition, we allow for the analyst to run the query in an already existing system, which allows for the analysts to avoid having to configure their own system. Finally, thanks to the system decoupled architecture, it is possible in the future to add some real-time analysis by adding new consumers in the raw_tweets queue that perform the mentioned static analysis, allowing for the existing feature in the current system to be re-deployed back in the new system.

4.5 Software development and maintenance

One of our goals was to develop a system that was easier to maintain. That was one of the reasons to adapt microservices was to avoid having components get outdated by making it easier for developers to onboard on the code and maintain it. In addition we also want to compare how easy would be to replace a component and code it from the ground up.

4.5.1 Current infrastructure

Currently the code is divided in 2 big monoliths. This can be a disadvantage on terms that developers must dig into the code of at least one of them in order to perform maintenance. This can be an small overhead as both of the projects are quite big.

EPIC Analyze has 5086 lines of code. On boarding this part can be difficult. Specially since it uses a pretty closed framework as Ruby on Rails is. This can be a difficulty in terms of having to write new code, anyone who wants to write something need to learn the usage of Ruby on Rails. This framework is big and has a quite fast learning curve. However, it can be difficult as it implements practices that sometimes are not obvious for a developer coming from other frameworks. In addition to the code, we have a slow deployment process. In order to update the deployed version of EPIC Analyze, we need to restart the server manually. This makes it difficult to deploy, which in result turns into a really slow development process. Development for Analyze happens on a git repository, this makes collaboration easier between different developers.

EPIC Collect is difficult to analyze as it's not on a git repository. Written in Java it's built thinking on efficiency first. This can be a bit of a difficulty in terms of collaborating and maintaining the code. In addition deploying new versions involves a lot of effort as it doesn't only involve deploying the code, but also maintain the Tomcat server. We also need to take into account all the processes that are keeping EPIC Collect alive. Due to the fact that this scripts are embedded onto the development of EPIC Collect, we need to redeploy them manually.

4.5.2 Proposed infrastructure

The proposed infrastructure is divided into multiple different pieces. On one side we have the components code, divided between Python and Go code. On the other side we have the declaratives documents on YAML that specify how to deploy the infrastructure into Kubernetes.

- *Twitter tracker (Go)*: 108 lines
- *Kubernetes controller (Python)*: 145 lines
- *Event manager UI (Python/Django)*: 2090 lines
- *Tweet normalizer (Python)*: 209 lines

Code is more equilibrated distributed between different parts. The good thing is that we are not tied into an specific and unique framework. Making code small makes it easier for other developers to understand the purpose of each component. In addition thanks to separating responsibilities into microservices we can make collaboration easier by letting different developers work on different parts without crossing each other.

Something that also helps is that we can use different technologies in each component. This means that we can focus on building on the best available option for what we need to do. It also allows for a better pivoting of a component into a language that the maintainer prefers. If it can be containerized then it doesn't matter what language or technology stack we use. It will be easily deployed into Kubernetes.

Finally we have the YAML files that store how the Kubernetes system looks like. All the system is declared in 3345 lines. There are many tools being released that will help make this code smaller. However right now the best option is to manually write and review each component in Kubernetes YAML files.

5 Results

After analyzing both systems and comparing their capabilities, we need to check how close have we got to the first stated goal. A part of our goal was to improve reliability and scalability. As seen in the previous section we can probably assume that we have accomplished it.

Moving the system reliability check to an external tooling allows us to ignore how it will be deployed while developing and focus more on the code and performance of each component. This is a feature that's making Kubernetes really great for companies. It helps separate the deployment logic with all the reliability and configuration outside, while making it easier for developers to deploy their code. Developers don't need a system administrator or a system expert to deploy their code, they just need to understand how containerization works. The rest can be done easily. It also allows us for appropriate tooling to be developed. Kubernetes is like a system platform, where you can create your own tooling. Thanks to the community, some reliability is already built-in, and adding capabilities to our deployment system becomes way easier. As an example for reliability, adding monitoring tools into Kubernetes clusters is really easy. The API exposes resource usage allowing you to configure alerts if needed.

Regarding scalability, thanks to the container orchestration and the stateless approach we took, it's easy to scale service up or down depending on our needs. This is an incredible improvement specially for Big Data Analytics. Resource intensive workload can be scaled on demand whenever we know there will be a high demand. The same for the collection system. We can increase the throughput if the system needs it.

On the other side, moving to a microservices-centered architecture will hopefully allow for a better development cycle making it easier to make the system evolve independently for each component. This allows for a better maintenance as well. Finally something else that this approach will allow us is extensibility. Thanks to the choreography approach, we now can add components to the system without having to change any other component.

Finally regarding performance, we got closer to real-time analysis by moving the query resolving responsibility to Spark, allowing for a high flexibility on the query thanks to Spark SQL and other toolings.

6 Future Work

This project was focused on proving the advantages that moving a Big data Analytics system to a container orchestration system could bring. Therefore the project itself would be more defined as a proof of concept than a production ready product. In that direction then is where future work could be focused on. The system can be improved to make it more production ready. Some feature that can be improved in this sense is a better study on each component resource usage. This could be used to make sure that each component is deployed with the corresponding resource usage.

Another feature that would need to be added into this before deploying is a centralized system for authentication and authorization. The proposed system doesn't have any security on purpose, we wanted to prove how the system would perform, so there was no need to make security a priority. In that way, developing and adapting the frontend components to use a centralized authorization protocol like JWT within a centralized microservice. This is not an easy part, especially since some of the system parts like Spark are using shared resources to work. Administering resource usage between users is a really important work.

Cassandra tables would need to be optimized for the system usage as well. For this project we used a really naive approach to tweet storage. It works pretty great for what we needed to do. However, if we want to replace the current system, this would need to be improved. A way to make this better is by upgrading event_name to partition key as described previously. In addition we would need to add logic in the partition assignment for the tweet normalizer, as Cassandra limits the amount of row a partition can contain and we need a way to make it easier for data to be distributed between all nodes in the best way. We could do this by assigning a random partition number when the tweet normalizer starts and choose a new value every 100.000 stored tweets. This would need to be studied more carefully in the future.

Then on the other side, thanks to the system extensibility, there are a few directions that the system could be improved. For example, we could add a real-time query resolver for a very specific query by plugging it into the raw_tweets queue and make it analyze the data. Or we could add other systems for specific queries like Elastic search for word search or text analysis. In addition we could extend the system to include better collaboration tools like a notification system plugged into the event_updates queue. The possibilities are quite a lot, the best part is that this doesn't need to upgrade other components of the system at any time.

Bibliography

[1] Ahmet Aydin and Ken Anderson. "Batch to Real-Time: Incremental Data Collection & Analytics Platform". In: *HICSS*. 2017.

[2] Cisco. *Cisco Visual Networking Index: Forecast and Cisco Visual Networking Cisco Visual Networking Index: Forecast and Methodology, 2016–2021*. Tech. rep. Cisco, 2017. URL: http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/complete-white-paper-c11-481360.pdf.

[3] EndPoint. *Benchmarking Top NoSQL Databases: Apache Cassandra, Couchbase, HBase, and MongoDB*. Tech. rep. EndPoint, 2015.

[4] Raffi Krikorian. *New Tweets per Second Record, and How!* URL: https://blog.twitter.com/engineering/en_us/a/2013/new-tweets-per-second-record-and-how.html.

[5] Sam Newman. *Building Microservices*. 1st. O'Reilly Media, Inc., 2015. ISBN: 1491950358, 9781491950357.

[6] Aaron Schram and Kenneth M. Anderson. "MySQL to NoSQL: Data Modeling Challenges in Supporting Scalability". In: *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*. SPLASH '12. Tucson, Arizona, USA: ACM, 2012, pp. 191–202. ISBN: 978-1-4503-1563-0. DOI: 10.1145/2384716.2384773. URL: http://doi.acm.org/10.1145/2384716.2384773.