UNIVERSITY OF COLORADO BOULDER

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 Bachelor's degree in Informatics Engineering

in

Facultat d'Informàtica de Barcelona
Universitat Politècnica de Catalunya

July 12, 2017

University of Colorado Boulder

Abstract

Big Data Analytics on Container Orchestrated Systems

by Gerard Casas Saez

Container orchestrated systems are quickly becoming one of the most used approaches for transactional systems. Now they are evolving to allow for more analytical systems to be deployed on top of them. I proposed and implemented a new system based on the Project EPID Big Data Analytics infrastructure in order to compare their reliance and scalability. The results show that scalability is increased and reliance is easier to ensure, lowering the mantainability cost as well.

Contents

List of Figures

Listings

1 Introduction

Big Data Analytics is a research area undergoing intense interest and rapid development. Data is being generated by sensors and computing systems at an ever increasing rate. Indeed, it is estimated that every month, seventy-two petabytes of information are moved around the Internet.[4] This amount is expected to grow to 232 petabytes/month by 2021.[4] This rapid growth is a significant challenge to the designers of software systems and to the field of computer science in general. New software infrastructure will need to be designed to support this demand and to extract useful information from it. In addition, a wide range of techniques and technologies must be mastered to keep analysis time low at this scale.

Containerization is one technique that can be useful in addressing these challenges and tools/technologies that provide containerization services are seeing increasing interest and development as well. The ability to run software in separate environments has been a revolution for the tech industry. Containerization is an abstraction for avoiding long installation processes on each system a software development team needs to run their software. It is similar to virtual machines but optimized to use system resources instead of simulated resources. Put another way, a virtual machine image contains everything needed to simulate a separate operating system. If you have multiple copies of that image, each image contains a complete copy of the operating system and all other components needed to run your software, wasting valuable disk space and consuming extra memory at run-time. With containerization, each container image contains just the software that you want to deploy and its required packages and the containerization run-time system provides a unified operating system for running all containers. Thanks to companies like Docker,[1] containers are becoming a de facto standard for software development and deployment. Its ease of use and consistent behaviour between machines makes it an attractive option for developers.

However, there are a few complexities: network configuration, interaction between containers, system updates, and more. To solve these complexities, container-orchestration systems were developed. These systems provide another abstraction layer on top of containers, taking over the management of an underlying containerization technology. In particular, the container-orchestration system is in charge of coordinating containers, creating them, destroying them, etc. The entire container life cycle is controlled by the orchestration system.

As these orchestration systems become more popular, software architects must create new software architectures that embrace the new possibilities that these systems provide. Some features that previously were difficult to achieve are now available to any software architect. Applications no longer need to have a static infrastructure; container-orchestrated systems can grow and decrease system resources on demand. Systems can be split into small microservices and deployed across a cluster

[1] https://docker.com

of machines; there is no need to merge system components into a big monolithic application anymore.

Furthermore, cloud providers are adding support for container-orchestrated systems, making it even easier to deploy complex infrastructures. Due to the similarities between different cloud offerings, you can design this new style of software system independent of a specific provider. One benefit to container-orchestrated systems is that it provides a means for finding new ways to solve problems associated with making use of parallelism and scaling a software system to handle ever increasing amounts of data. If properly designed, surges in demand can be met by spinning up additional copies of core system services balanced across the cluster of machines you have reserved from your cloud provider.

The goal of my thesis is to explore how this new style of software architecture can be used to deploy software infrastructures that are used to perform big data analytics. In particular, I will be migrating an existing big data infrastructure to make use of the techniques and technologies offered by container-orchestrated systems and compare the new version of the system with its previous incarnation along a number of dimensions.

2 Background

Before digging into my specific project, I provide background information that will help to understand my problem domain. Here, I present information on three different topics: the first provides information on the big data software infrastructure that I use in my work; this infrastructure was developed by Project EPIC [1, 14, 2] to support research into an area known as crisis informatics.[13, 15] Second, I will present more information on container-orchestration technologies and then discuss microservice architectures in more depth.

2.1 Project EPIC

EPIC (Empowering the Public with Information in Crisis) is a project at the University of Colorado Boulder. It conducts research on crisis informatics, an area of study that examines how members of the public make use of social media during times of mass emergency to make sense of a crisis event as well as to coordinate/collaborate around the event.

Project EPIC has a long history of performing software engineering research. Since 2009, Project EPIC has been investigating the software architectures, tools, and techniques needed to produce reliable and scalable data-intensive software systems, an area known as big data software engineering.[2] When large amounts of data are being collected, software engineers need to focus on structuring this data so it is easy to perform analysis and they must work to ensure that this data is easily accessible to analysts. Project EPIC's big data software engineering research has explored these issues in depth in and around the creation of the Project EPIC software infrastructure that consists of two major components, EPIC Collect and EPIC Analyze.

EPIC Collect is a 24/7 data collection infrastructure that connects to the Twitter Streaming API to collect tweets from various crisis events that need to be monitored in real time. Since 2012, this software has been collecting tweets with an uptime of 99% and has collected over two billion tweets across hundreds of crisis events. EPIC Collect's storage layer makes use of Cassandra. This NoSQL database is focused on writes and provides high throughput. EPIC Analyze is a web-based system that makes use of a variety of software frameworks (such as Hadoop, Solr, and Redis) to provide various analysis and annotation services for analysts on the large data sets being collected by EPIC Collect. In addition, Project EPIC maintains one machine—known as EPIC Analytics—with a large amount of physical memory to allow analysts to run intensive processes over the collected data.

The software architecture of EPIC Collect and EPIC Analyze is shown in Figure 2.1. This is a logical architecture that does not show how these systems are deployed. For instance, Cassandra is deployed on four machines that run separately from the machines that host the EPIC Collect software, Postgres, Redis, and the Ruby-on-Rails

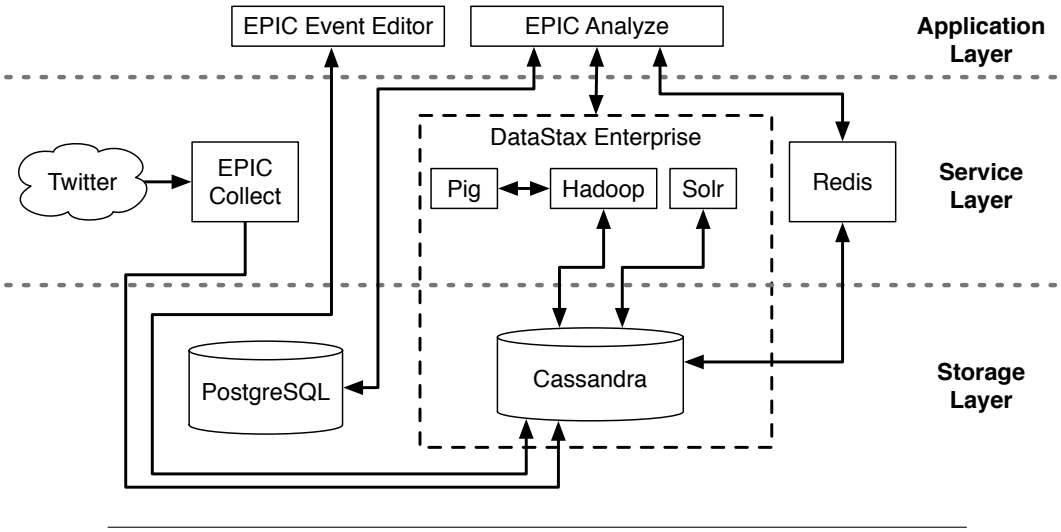


FIGURE 2.1: EPIC Analyze and Collect architecture

code that makes up EPIC Analyze. In all, the existing Project EPIC infrastructure is distributed across seven machines in a single data center maintained at the University of Colorado.

2.2 Containerization

Containerization is a technique that provides operating-system-level virtualization. This technology allows programs to be run inside a "system container"; these containers are isolated from each other, as well as the host machine that is being used to execute them. Containerization works similarly to traditional virtual machines from the point of view of a program, but makes more efficient use of the underlying system resources instead of simulating a full-powered operating system.

There are many containerization platforms available; however, the most widely adopted project is Docker; I will make use of Docker as part of the prototype I have built for testing my thesis work. Docker originated as an internal tool for a PaaS (Platform as a Service) company and was later made available as open source. Thanks to many contributions from different companies, this project has grown to become the most used containerization software. It is able to run on almost any machine and many software packages have been made available as Docker containers, providing a wide range of integration opportunities. Finally, almost all existing container orchestration systems support Docker containers.

2.3 Container orchestration technologies

As containerisation technologies became more widely adopted—spurred by a recent migration to designing systems via microservices—large companies needed a way to manage their containers in a more friendly way. Interconnecting containers, managing their deployment, and scaling them to meet demand, were possible with containerisation technologies, but were difficult to achieve. As a result, container orchestration systems were born. To manage containers, these systems add

an abstraction layer over containerisation technologies, making such tasks easier to perform.

There are a few available container orchestration systems at the moment. The most popular ones are Kubernetes[1] and Apache Mesos[2]. In this project, I will make use of Kubernetes. The main reason for this is that—thanks to the open source community—it is easier to find tutorials and courses for Kubernetes. In addition, there are a lot of big companies backing this project and contributing to it, which provides evidence that this project will be supported well into the future.

Google Cloud seems like the best fit to host Kubernetes as it has a managed cluster option that makes it straightforward to install and connect system components. In addition, thanks to Google being part of the maintenance team for Kubernetes, there is a great support for Google Cloud infrastructure within 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 making use of big components with a lot of responsibilities that make interaction more difficult. They are thus more cohesive units of software with minimal dependencies between them. A system designed with loosely-coupled, highly-cohesive software components has always been a highly-desirable goal in software design[3] and microservices help to achieve that goal with distributed software systems...

In addition, thanks to new technologies like containers and container-orchestrated systems, microservices are quickly becoming a standard tool in the design of large software systems. This type of architecture has been adopted by many companies as it makes maintenance of these systems more straightforward.

Another key advantage of the microservices approach is that it makes it easier to adopt different technologies for each component of your software system. In addition to containerization, it decouples different parts of a system. This separation allows a better optimization of technology depending on the feature that each microservice needs to provide. For example, a microservice that needs to store highly-related data can make use of a graph database behind the scenes while exposing a REST API for uniform access. At the same time a separate microservice that needs to store big documents can use a document store underneath, allowing a better fit for its needs while allowing both microservices to be deployed as part of a single software system.

Having small microservices do specific tasks, makes development cycles faster and more independent. It also makes incremental deployment much easier and less dangerous. Finally, microservices make it easier to scale software systems since one can individually scale parts of the system depending on their usage.[5]

Microservices avoid problems associated with previous service-oriented architectures, by focusing solely on the overall software architecture and letting the architect decide how services are to be deployed or what messaging system is used to communicate between them. In addition microservice architecture allows for improved

[1] https://kubernetes.io/
[2] http://mesos.apache.org/

reliability and resilience, as each microservice is independent. If one fails, the rest are unaffected. This feature is a perfect match with container orchestration systems that can easily perform health checks on microservices and redeploy any that have failed.

In terms of designing a microservice architecture, there are two approaches that are equally used at the moment: choreography and orchestration.[12]

2.4.1 Coreography

Choreography centers all interactions around a publish-subscribe philosophy. Events are propagated through the system in an asynchronous manner. When something happens in a microservice, a message gets published on a public queue. Other microservices can subscribe to this queue and respond when they receive that message.

This approach makes a system more flexible, removing the responsibility of knowing how to send messages to other microservices. One can add more components easily without having to modify the existing ones. However, we need extra components to make sure that all tasks are performed once an event is published. We can accomplish this by installing and maintaining a monitoring solution.

2.4.2 Orchestration

Orchestration centers interactions in request/response interactions. Each microservice makes requests directly on other microservices. For this approach to work, one needs a service discovery backend. In addition, if a service is down, we need to make sure that microservices are architected to retry failed attempts and to fail gently if a required microservice is unavailable. The benefit to this approach is that due to the synchronous nature of these interactions, we know that all steps of an activity have been completed when the original call returns. The limitation is that one non-responsive microservice can bring down an entire set of actions.

A more modern approach to orchestration has been developed with service meshes.[11] A service mesh has the responsibility of providing service discovery while also balancing requests across multiple instances of a microservice; they also guard against having an activity fail by retrying requests. A service mesh works by adding a sidecar to all microservices allowing it to act as a proxy between a microservice and the outside network. Unfortunately, service meshes are still new technologies that are rapidly evolving; they often lack features found in other approaches such as the extensibility capabilities found in choreography-based systems. As a result, I do not make use of them in my thesis work.

2.5 Messaging systems

Messaging systems organize queues of messages produced by microservices and notify subscribed components when they arrive. Their main objective is to decouple components and to serve as a cache if consumers cannot digest all of the incoming messages. This allows for more reliable systems as a system does not depend on the

mutual availability of a sender and a receiver to pass messages as they would if they used other messaging system like HTTP.

There are many options available in this space, with the most popular being Apache Kafka and RabbitMQ. Kafka is a decentralized, high-availability messaging system that allows one 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. Each partition can have one consumer associated with it, providing faster reads.

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

In addition, Kafka is one of the most used messaging systems in the industry. Many companies use it in production to decouple their systems. Kafka has a strong community behind it as well, and large organizations (e.g. LinkedIn) take an active role in maintaining it.

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

2.6 Big Data storage systems

For my thesis work, I will be collecting data from Twitter via the Streaming API. This API is limited to provide 1% of the total tweets generated in Twitter every minute. Based on a report from 2013, I know that rate is 5700 tweets[9] per second on average. As a result, I can estimate the total number of tweets per second that I can estimate will flow through my system per second and that is 57 tweets per second on average as a minimum bound. Since that corresponds to $\tilde{4}$.9M tweets/day, I need a data storage technology that scales to handle large datasets. As previously studied in the EPIC project[14], I need to use a NoSQL database instead of a relational database to support the high throughput of tweets and make my system more scalable.

On the other hand I also need a system that allows analytical and operational queries to happen at the same time. I would like to be able to analyze my data in real time without having to stop my data collection process. Given that some analysis may take a few minutes due to the high number of tweets, I need to have a system that allows a high throughput for both parts. In this case, Cassandra is an excellent option as the number of operations it can perform per second scales better compared to other NoSQL alternatives, especially with operational and analytical workloads.

To store tweets, I base my table structure on the current EPIC Analyze column family structure in Cassandra as described by the CQL code on the next page. I have added an index on the event_name attribute so that I can access events faster. I need this index as many queries are performed per event.

```
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

3 Problem Statement

The goal of my thesis is to explore the benefits and limitations in using container-orchestration systems to build and deploy software systems that engage in big data analytics. To perform this exploration, I will be redesigning the Project EPIC software infrastructure as a set of microservices—each inside a separate Docker container—that are deployed on a cluster of cloud-based machines via a container-orchestration system. The current infrastructure was manually deployed on a set of physical machines in a local data center and was not developed using microservices or containerization. My hypothesis is that I'll be able to achieve greater scalability and reliability with the new architecture with significantly reduced maintenance costs. I will also explore whether I am able to achieve greater query flexibility and overall performance using this new style of software infrastructure. My specific research questions are:

1. What advantages and/or limitations will the new Project EPIC infrastructure have with respect to its predecessor?
   (a) Is it more reliable? If so, why?
   (b) Is it more scalable? If so, why?
2. Does the new infrastructure have lower maintenance costs than the existing infrastructure?
   (a) Is it easier to deploy?
   (b) Is it easier to upgrade?
   (c) Is it more resilient to failures? If so, how?

4 Approach

In this section, I present the design of my replacement for the Project EPIC infrastructure. The primary features of this infrastructure are:

- Event management (creation/deletion/modification of events with specific keywords)
- Real-time collection of streaming Twitter data
- Real-time classification of incoming tweets (assigning tweets to active events)
- Data Analysis (performing queries on collected tweets using batch processing)

My goal is to recreate these features with less code that is easier to deploy and update and is more flexible, scalable, and reliable than the existing infrastructure.

As mentioned above, my design will rely on a choreography-based microservices approach that is deployed on a cloud-based infrastructure making use of Docker and Kubernetes. My microservices will rely on Kafka to pass messages to one another via message queues (topics) that are created in response to requests to collect on crisis events.

One goal I have in my design is to make as many of my system components to be stateless. That is, these components will be designed to receive an incoming message from a Kafka queue that contains all the state they need to perform their operation. They will, in turn, generate messages that contain all the state that is needed for downstream components to process them. This design approach will pay dividends as it will allow the underlying container orchestration system to easily replace services that have crashed and to instantiate multiple copies of a particular service when there is a surge in demand for its services.

Not all of my components will be microservices. I will rely on Cassandra to persist the tweets that are collected; the reasons for using Cassandra have already been well documented in Project EPIC's prior work.[1, 14, 2] Furthermore, I will be making use of Apache Spark to perform data analytics in my prototype and will provide a web-based "notebook user interface" for the analysts to easily submit their queries and view the results. Examples of tools that provide notebook user interfaces for data analysis are Zeppelin[1] and Project Jupyter[2]. These platforms integrate directly with Apache Spark and provide advanced features for viewing the results of Spark queries as tables and charts. The use of Apache Spark in my infrastructure was driven by the fact that it comes with a specialized "Cassandra connector" that allows it to perform queries on a cluster of Cassandra nodes efficiently. For instance, the Cassandra connector is smart enough to "take the query to the data" and perform a particular query distributed on the data stored in each Cassandra node rather than requiring data to be transferred between nodes before the query is performed. This

[1] https://zeppelin.apache.org/
[2] http://jupyter.org/

feature is known as preserving data locality and allows Apache Spark to perform queries on Cassandra in near real-time.

4.1 Custom Components

The specific components that I will be developing for my infrastructure are:

- **Event Manager UI:** A web-based system that presents a user interface for managing data collection events. Each event has a name and a set of associated keywords. For instance, if a hurricane were to threaten the Eastern Seaboard of the United States, an event would be created using the hurricane's name and the year it occurred. For example, "2012 Hurricane Sandy." The list of keywords then specify items of interest related to that event; these words are typically place names ("new york city"), behavior-related terms ("evacuate", "charging devices", etc.) or event-specific terms ("flooding", "hurricane", etc.) and it is a request to collect every tweet that contains one or more of these terms. Each time there is a change made to the current set of events, the Event Manager UI submits a message with all of the current events and keywords to a Kafka queue to report the change. The Event Manager UI makes use of its own local database (SQLite) to ensure that its state is saved in the event of a crash.
- **Infrastructure Controller:** The infrastructure controller is responsible for issuing commands to Kubernetes to ensure that all instances of the microservices needed to collect the current set of events are up and running. It receives the messages generated by the Event Manager UI and updates the Twitter Tracker and Twitter Normalizers (both discussed next) to match the new state.
- **Twitter Tracker:** The Twitter tracker is a microservice that connects to the Twitter Streaming API, submits a set of keywords provided by the Infrastructure Controller, and then places each tweet that it receives in a Kafka message queue. This will be done with no downtime, by destroying the old instance only when the new instance has established a connection with Twitter and starts receiving tweets. All configuration will be done by environment variables with no storage, making this microservice stateless.
- **Twitter Normalizer:** A Twitter normalizer is a microservice that gets assigned the keywords associated with a single event and is plugged into the Kafka queue that receives tweets from the Twitter Tracker. (This particular queue is configured such that all subscribers see all messages; in this case, each message contains a single tweet that was received from the Twitter Streaming API.) When a Twitter normalizer finds a tweet that contains one of its keyword, it normalizes that tweet to match the schema shown in Listing 2.1 and stores the tweet in Cassandra. Note: since more than one event can specify the same keyword (for instance two hurricanes active at the same time may both contain "hurricane" as one of their keywords), a tweet may be stored multiple times by my prototype. Since most queries are focused on a particular event, this duplication is not an issue. However, if a query is specified across multiple events, then the onus is on the analyst to remove duplicate copies of a tweet before metrics are calculated. One Twitter normalizer is created for each event specified by the state produced by the Event Manager UI. Of course, the Infrastructure Controller has the option of instantiating multiple instances of

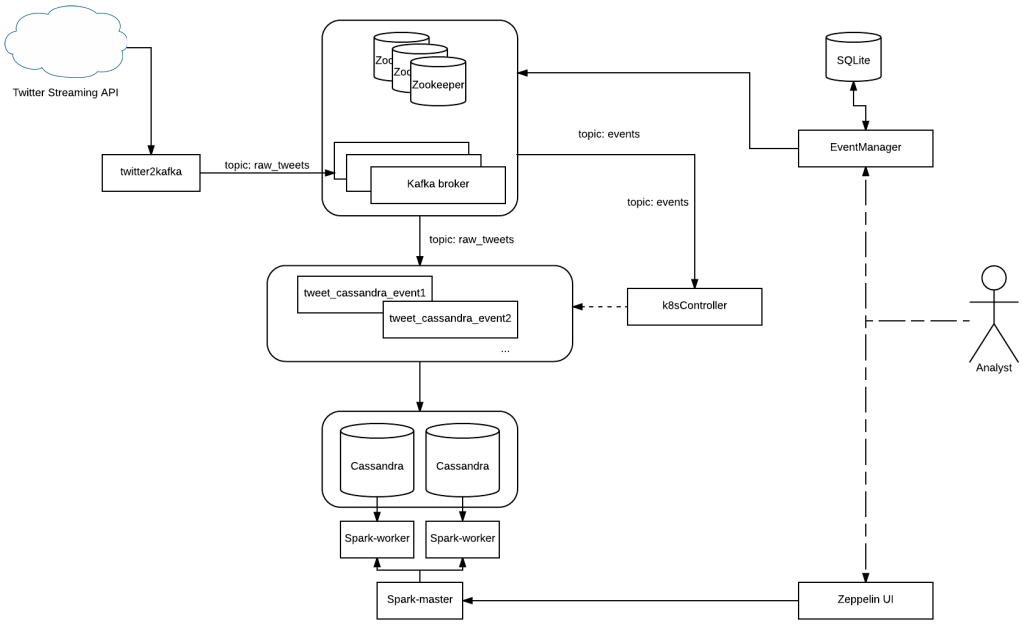


FIGURE 4.1: Proposed system architecture

a Twitter normalizer if, for instance, it has determined that an event has been assigned a lot of high-frequency keywords but that particular functionality has not yet been implemented in my prototype.

Given these descriptions, the software architecture of my prototype is shown in Figure 4.1. In the next section, I will present more information about how my prototype is actually implemented.

5 Implementation

In this section, I provide implementation-related details of my software prototype. The prototype is fully implemented and running on a set of nodes hosted by Google Cloud. We created four nodes in total each with one virtual CPU and 4 gigabytes of RAM. On those nodes that host Cassandra, we reserve 1 gigabyte of RAM for Cassandra's exclusive use and allow the remaining memory to be used by other components deployed on those nodes (such as Apache Spark). The nodes that host Cassandra are deployed using a feature from Kubernetes called stateful sets. This data structure associates persistent volumes with particular system components such that data is preserved across restarts and crashes. They are designed to be used with distributed databases like Cassandra making it easy to add and remove nodes that host Cassandra at run-time. Thus, if we add a new node to our Cassandra cluster, Kubernetes ensures that a new persistent volume is created and attached to that node and then ensures that each time that node is activated the same persistent volume is made available to the database running on that node.

As mentioned above, we also designed our Kubernetes configuration to deploy an instance of an Apache Spark worker on each Cassandra node and then also specified one additional node to serve as the Apache Spark master node. A container with Zeppelin was also deployed configured to send queries to the Spark master node via Zeppelin's cassandra-spark library.

I now present details on how each of the custom components discussed in Chapter 4 were implemented. In general, microservices were implemented first in python for ease of prototyping and then switched to a different implementation language if performance problems were detected. Furthermore, all microservices were placed in individual Docker containers which were then deployed via Kubernetes as dictated by the Infrastructure Manager. Further details on each component are now presented:

- **Event Manager UI:** The event manager is implemented as a stateful django web application. It supports CRUD operations on events. Any change of state is pushed out as a message on a Kafka queue (and acted upon by the Infrastructure Controller). It stores its data in SQLite as a file on a Google Cloud persistent disk. This set-up ensures that it can find its state across restarts.
- **Infrastructure Controller:** This controller is written in python and makes use of python libraries that allow it to interact with Kafka and Kubernetes. When it receives a message from the Event Manager UI, it issues commands to Kubernetes to declare the new state of the world. If an event is no longer active, its associated Tweet Normalizer will be shut down. If a new event is specified, a new instance of the Tweet Normalizer is configured and deployed.

- **Twitter Tracker:** The Twitter Tracker was first implemented in Python but I discovered that python's run-time engine was not fast enough to handle consistently high streaming volumes over long periods of time. As a result, I reimplemented this microservice in Go for better reliability and performance. As discussed above, this service submits keywords to Twitter's Streaming API and then stores each tweet that it receives in a Kafka topic. The infrastructure controller is the one in charge of deploying and updating the instance. A set of all the tracked keywords is passed as environment variable on start by Kubernetes To update the keywords Kubernetes perform a rolling update by creating a new instance and destroying the old one once the new one has correctly started the stream.
- **Twitter Normalizer:** The Twitter Normalizer is the one component in my infrastructure that can be instantiated multiple times and needs to monitor a different set of keywords in each instance. To facilitate this, I had the Infrastructure Controller direct Kubernetes to pass the keywords needed by each instance of the Twitter Normalizer as environment variables. Kubernetes could then deploy an instance of the Twitter Normalizer Docker container onto a node, configure its environment variables to match the keywords of the given event, and launch the microservice. The Twitter Normalizer was implemented in Python but specific C-based libraries were used to implement tasks that it executes over and over, e.g. loading and parsing JSON objects. This technique enabled the Twitter Normalizer to process the incoming stream of tweets with acceptable performance.

5.1 Deploying the System

Deploying my prototype is straightforward given the use of Google Cloud and Kubernetes. As mentioned above, I created a four-node cluster with each node allocated 1 virtual CPU and 4 gigabytes of RAM. Kafka and Cassandra/Spark are deployed first. In my prototype, I created two Kafka brokers that work together to manage the two primary topics needed by my design (the queue between the event manager and the infrastructure controller and the queue between the Twitter tracker and all instances of the Twitter normalizer) and instances of Cassandra/Spark on three of our four nodes. We then deploy the containers for our two front-end components: Zeppelin and the event manager. Finally, we deploy an instance of the infrastructure controller. All other components will be deployed by the infrastructure controller (including the Twitter tracker) when it receives a message from the event manager. This approach makes sense since there is no need to have the Twitter tracker and the Twitter normalizers running if there are no events to collect.

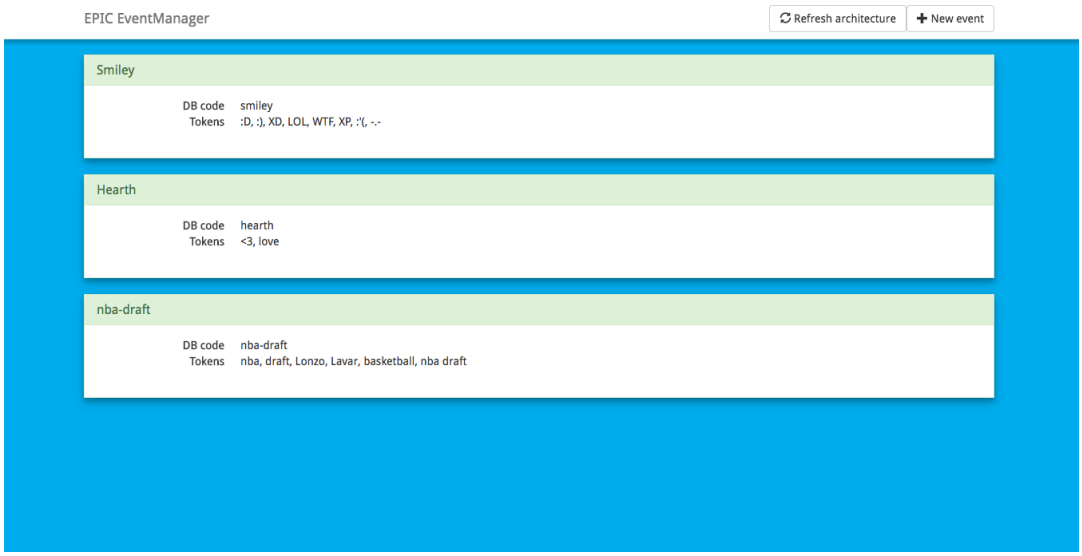


FIGURE 5.1: Event manager UI

5.2 Front-End Components

Figure 5.1 shows the user interface of the event manager. Each event is shown in a separate box with information about the event's keywords. There are controls for creating new events and a separate control for sending a message to the infrastructure controller with the most recent state of the world. Events can be edited/deleted by controls that appear when its box is selected. Figure 5.2 shows the user interface provided by Zeppelin. Queries can be submitted via a textbox and results can be displayed in tables or via bar graphs (as shown).

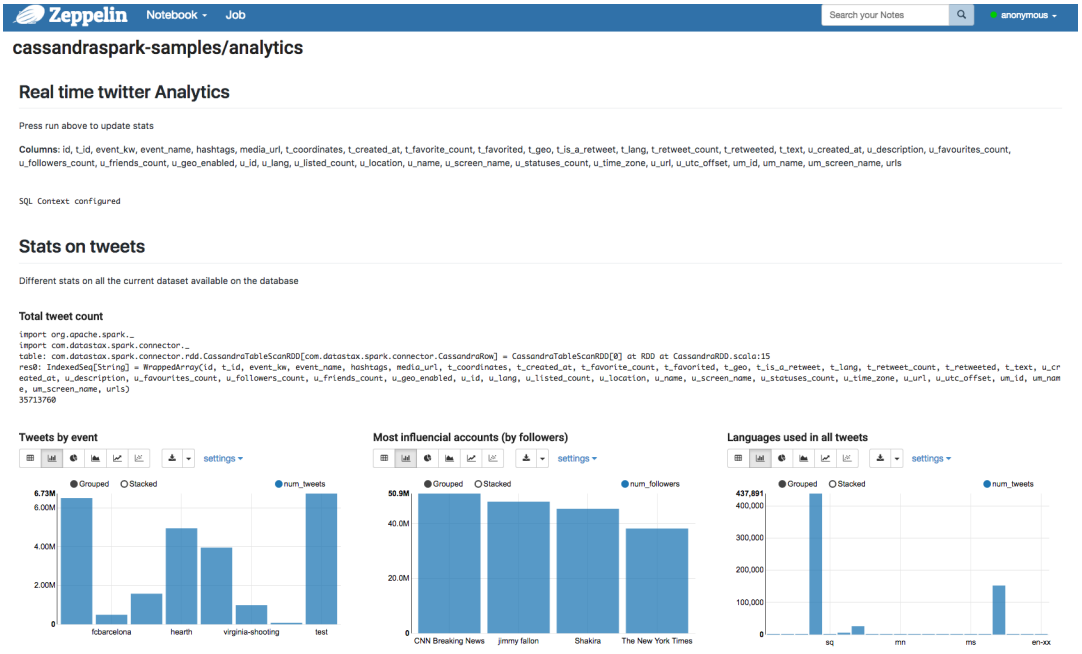


FIGURE 5.2: Zeppelin server with some visualization from the dataset

6 Evaluation

In order to evaluate my work, I compare the architecture and implementation of my prototype with the existing Project EPIC infrastructure along the dimensions of reliability, scalability, performance, and real time delivery.

6.1 Reliability

In order to evaluate my work, I compare the architecture and implementation of my prototype with the existing Project EPIC infrastructure along the dimensions of reliability, scalability, performance, and real time delivery.

6.1.1 Current infrastructure

Reliability of the existing EPIC infrastructure needs to be studied separately across EPIC Collect and EPIC Analyze. EPIC Collect is designed to be highly available. It is implemented as a multi-threaded Java application; some threads are used to read the incoming tweet stream, some are used to classify tweets, and some are used to monitor the other threads. If the monitoring threads detect that one of the readers is no longer producing tweets for the classifiers, it can issue a command that causes the current connection to the Twitter Streaming API to be dropped and all of these threads to be deactivated. This action, in turn, causes another thread to detect that the Twitter connection is down and it then restarts the connection which causes readers, classifiers, and monitors to once again be instantiated. This approach can ensure reliable performance for many days; however, sometimes errors occur that cause all threads to lock up. To handle this situation, EPIC Collect makes use of a cron job that wakes up once per minute to examine the current length of the EPIC Collect log file. Each reader and classifier will send information to the log file and when data collection is proceeding smoothly, the log file is always increasing in size. As a result, if the cron job wakes up and discovers that the log file has not increased in size over the past minute, it assumes that the collection software has locked up. It will invoke a command to kill the previously running process and it then invokes the collection system, notes the size of the log file, and goes back to sleep. With these techniques, EPIC Collect has achieved 99% uptime since the summer of 2012. The only problem that these techniques cannot account for is if the data center loses its network connection. When that happens, the cron job will be stuck in a cycle of terminating and restarting the software until the network connection is restored. Fortunately, complete loss of the data center's connection to the Internet is a very rare event, happening only once in a four year period.

EPIC Analyze does not have the same level of reliability.In order for the web application to function, it requires that Redis and Solr be up and running. If these systems

are not available, then the web application is non-functional. Compounding this situation is the fact that EPIC Analyze is deployed manually by its developers; there is no automated way to deploy it and there is no monitoring system detecting for system failure. As a result, there is also no automated recovery procedure. All aspects of the system deployment for EPIC Analyze require manual intervention by developers.

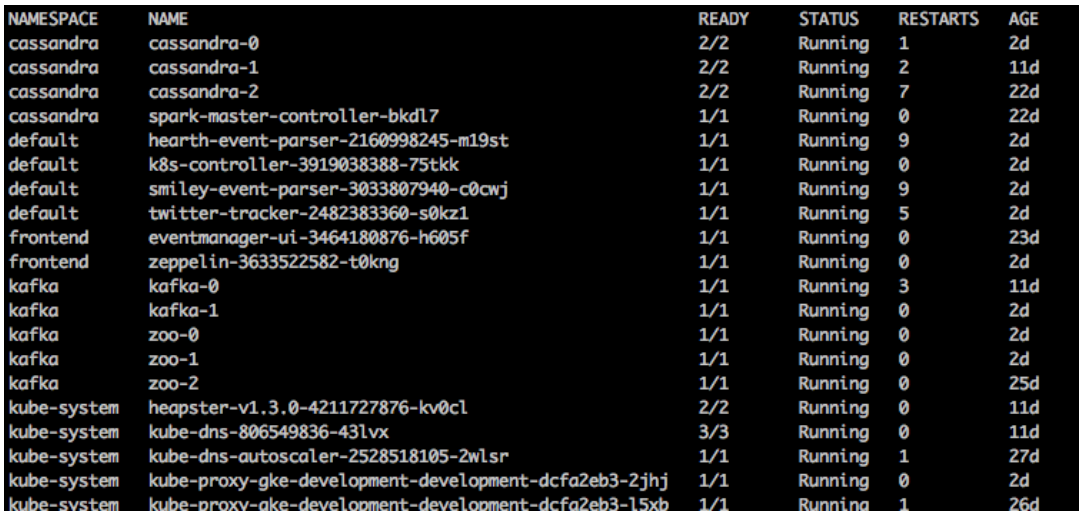
6.1.2 System Prototype

In the case of my system prototype, overall reliability is high, due to the use of a container orchestration system. Kubernetes provides two components to increase the reliability of our system. The first Kubernetes-provided component is the controller manager that runs on the master node of our cluster. This component is responsible for keeping track of all containers running on our cluster. It also follows the requests made by the infrastructure controller to ensure that the right number of replicas are created for the components that need them. For instance, in my prototype, I specify that there should 2 replicas of a Kafka broker available at all times. If any of those replicas go down, Kubernetes will detect that and launch a new one. This functionality extends to all of our containers; if any container stops running, the controller manager detects it and schedules a new instance of that container to run on an available node. This check is performed when the infrastructure controller makes new requests or when an existing node informs the controller manager that one of its containers went down.

The second Kubernetes-provided component is the scheduler. This component makes sure that new containers are scheduled on the best node possible. Kubernetes allows an engineer to configure the amount of memory and cpu permitted by a container; this information allows the scheduler to find the best fit for each deployment request.

With these two components, Kubernetes automates the deployment of containers on a cluster of machines and handles any failures automatically. Its services are significantly more advanced than the existing reliability measures put into place by the Project EPIC developers, who were more interested (at the time) in system functionality and not in automated failure recovery mechanisms beyond what was done to ensure reliable data collection.

Kubernetes provides one additional reliability-related feature and that is related to upgrading containers to provide new versions of the microservice within, it's called rolling update. When performing an upgrade, the controller manager first deploys the new version of the component and ensures that it is up and running. It then removes the container containing the previous version of the component. I make use of this functionality with the Twitter Tracker component. When a new change to data collection is announced by the Event Manager UI component, the infrastructure controller arranges to have a new instance of the Twitter Tracker component deployed with the new state. It starts to collect tweets using the newly updated keyword list while the previous instance is still collecting data on the prior set of keywords. This approach ensures that no tweets are missed when the transition occurs. Twitter applications can only have one standing connection. When the new instance establishes a connection, Twitter will close the old connection ensuring that there won't be any duplicated tweet stored in the transition.



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 6.1: Number of automated restarts by Kubernetes of the system prototype over a period of eleven days.

Figure 6.1 displays a typical report for the number of times the containers in our system prototype were restarted automatically by Kubernetes over a period of eleven days with no interaction from the user. The first thing to note is that all components remained active for the entire time period; that is data collection proceeded uninterrupted during those eleven days. However, due to system demands, Kubernetes may have found itself needing to, for instance, delete a container on an overloaded node and move it to a node that had more resources available. Given that my code is now fairly stable, if we increased the amount of memory on each node and added a few more nodes to our cluster, the total number of restarts would go down. But, given the limited resources I had during development, the more important issue is that despite limited resources, the system continued to run 24/7 with no interventions required by the developer. Note: that in Figure 6.1, some instances are listed as only existing for two days; this discrepancy is due to the fact that Kubernetes will restart a container's count if it needs to do a hard restart of the container. This occurs when Kubernetes issues a request for the container to shut down and it stays active, ignoring the request. This might occur due to the contained microservice crashing inside and thus unable to exit gracefully. As a result, Kubernetes is forced to kill the container without a graceful shutdown.

6.2 Scalability

The second dimension I am using for my evaluation is scalability. I am interested in how well both infrastructure deal with the ability to scale to large amounts of data. I want to avoid wasting system resources unnecessarily while having the capacity to scale and I want to understand how scaling impacts overall system performance.

6.2.1 Current infrastructure

Scalability with the existing infrastructure is not a straightforward process. With respect to throughput capacity, EPIC Collect would need a developer to manually

launch a new instance with new Twitter credentials and set-up an a second cron job to monitor the second instance's log files. That work is feasible but not straightforward and would have to be performed anew if more capacity was needed and a third instance was required. This situation is one reason why EPIC Collect does not perform data normalization; it simply classifies tweets with respect to the current set of events and then it stores them in Cassandra. If data normalization was added to the existing EPIC Collect then it would struggle to handle spikes in incoming traffic as it would not be able to automatically scale its capacity to handle demand.

With respect to storage capacity, EPIC Collect is in a better situation since it stores tweets to a four-node Cassandra cluster with terabytes of disk space available. If additional capacity is needed, a developer just needs to configure a new node and add it to the existing cluster. As with the discussion above concerning throughput capacity, while this works, it is hardly an automated approach to scaling the infrastructure on demand. With respect to EPIC Analyze, the only component that is a target for scaling, apart from storage, is the frontend module that is currently written in Ruby on Rails. To do that, multiple instances of the application would need to be launched and a load balancer put in front of those instances. To make this work, the existing web application would need to be refactored such that session state can be made consistent across all of the instances. That is a straightforward engineering task but not simply by any means. Furthermore, maintaining a load balancer is a complex process when done manually and, as discussed above, all maintenance on EPIC Analyze has to be performed manually.

As a result, I must conclude that the existing Project EPIC infrastructure is not easy to scale.

6.2.2 System Prototype

Scaling my system prototype is much easier given the functionality gained from Kubernetes. When deploying a microservice via a container, I can specify how many replicas of that service I would like to deploy alongside it. The controller manager will try to schedule the requested number of replicas as long as there are enough system resources available across the cluster. Our ability to deploy multiple replicas is helped by the fact that most of our services were designed to be stateless. All of the state that they need to perform their task is contained in the messages that it receives. As a result, it doesn't matter which replica handles a given input message. Replicas can also be automatically triggered based on CPU usage. If a container hits 90% CPU utilization because it has experienced a spike in the number of input messages, Kubernetes can automatically spin up new replicas until the utilization goes down due to the fact that the new replicas can help the existing component handle the spike in messages.

While these services are largely automatic, a developer can always interact with Kubernetes directly to manually deploy additional components to help with scalability. The developer can issue these commands using the kubectl command line tool or via Kubernetes's web interface.

Due to the features of container orchestration systems, my system prototype is much easier to scale than the existing Project EPIC infrastructure.

```
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 6.1 WordCount Spark script

6.3 Performance

I am unable to provide a comparison between the two infrastructures with respect to performance. Both systems have similar performance with respect to data collection but detailed tests and comparisons are not possible since EPIC Collect is a production system that other members of the Project EPIC team depend on for their research. As such, in this section, I provide insight into the performance my system prototype achieves on my small four-node cluster. As a reminder, each node in the cluster has access to one virtual CPU and four gigabytes of RAM. My test involves executing a Spark-based job on the cluster to count the number of words contained in all collected tweets. Each time I ran the query, the total number of tweets collected was different. I selected this particular test since word count is a highly parallelizable operation. It also demonstrates the integration of Spark into my system prototype.

The code entered into Zeppelin is shown in Listing 6.1. The first line establishes a connection to Cassandra and creates a Spark Context object by which queries can be invoked. The second line expresses a series of transformations that Spark will apply to count all of the words of all tweets contained in Cassandra. It starts by selecting the text of the tweet, splitting the text into words, mapping each word into a pair (word, 1) and then reducing all such pairs by adding up the integers for each matching key. Thus all pairs like (cat, 1) would eventually turn into a single pair (cat, 1000) where 1000 represents the number of times that cat appears in the collected tweets. Finally, the pairs are inverted, e.g. (1000, cat), and then sorted in descending order.

The power of Spark is that all of these transformations are applied to every tweet in parallel and, furthermore, as much of the transformations are applied locally on every node before any data is sent to the master node for the final combination of pairs across nodes. Spark provides a mechanism to reveal how it will execute a query known as the debug string.

Each indentation in the debug string is a map stage, and each + is a shuffle phase. As we can see in Listing 6.2, Spark delays shuffling data until it is time to execute the reduceByKey step. This makes sense since it can generate word pairs on each node without having to transfer data across nodes. However, once it needs to count the total number of words, it has to send data across nodes to a master node to create the final counts. It then performs one more shuffle when it sorts the final key-value pairs after swapping them from this format (cat, 1000) to this format (1000, cat).

As we can see in Figure 6.2, the word count performance is not linear but, indeed, performs better with larger numbers of tweets to analyze (see the increase in tweets processed per second in Table 6.1). The likely reason for this performance curve is

```
(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 6.2 Debug string rdd

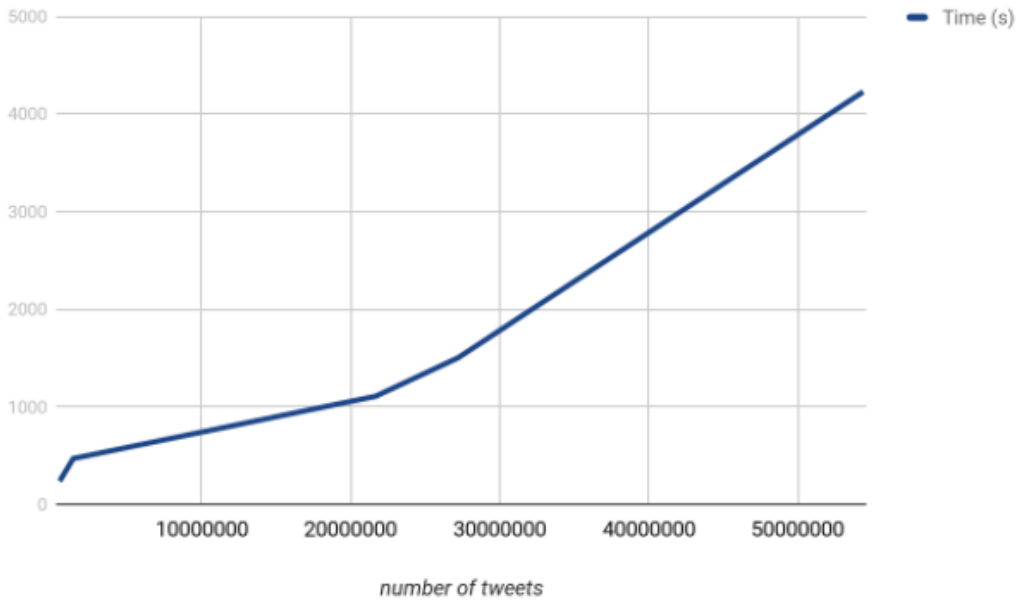


FIGURE 6.2: Plot on wordcount time for dataset size

that there is a certain amount of overhead that is incurred each time to stage the job and perform the shuffle steps at the end, counting up the pairs and sorting them. However, as the number of tweets increases, each individual node can do more work uninterrupted and can execute as quickly as possible without the need for coordination messages. As a result, the performance increases and keeps the overall time sublinear at least for datasets in the millions of tweets.

6.4 Software development and maintenance: Ken start here

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.

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

TABLE 6.1: Number of tweets in dataset vs seconds took to perform word count

6.4.1 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.

6.4.2 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.

7 Results

After analyzing and comparing both system we are going to extract and do a definitive comparison to answer the proposed questions in chapter 3.

On the advantadges/limitation respect our current system, we have see some new capabilities for the system that were too complex to implement in the previous infrastructure of Project EPIC. One of them is the system availability tracking. Moving it 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 makes the system more reliable at the same time that keep development simplicity. However the field that seems to have a major improvement is scalability. Thanks to the stateless microservice approach, scaling is easier than before. In addition, the container orchestration systems provides an abstraction layer to scale up and down different components individually saving us time for configuration. Resource intensive workload can be more efficient by scaling up the resource intensive instances, increasing the throughput if the system needs it.

Regarding mantainability costs, we have several improvements. First, using microservices, separates responsabilities into different components that are smaller in size. This size difference makes it easier for developers to embark into each microservice mantainance without having to understand the whole system at once. This in turn means that we don't need experts for the whole system, each microservice can be mantainer independently. On the other side, Kubernetes offers a platform to deploy the microservices that allows a better separation between deployment infrastrustructure and application development. 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 for specific tooling to be developed. Kubernetes is like a system platform, where you can create your own tooling. Thanks to the community, some tools are already built, and can be added with ease into Kubernetes clusters. As an example for reliability, we can add monitoring tools into Kubernetes clusters in a really easy way. The API exposes resource usage allowing you to configure alerts if needed, and the community has built packages of configuration files that help to deploy monitoring with ease.

About deploying the infrastructure, we can create configuration files that can be used in the container orchestrated system. Making it really easy and fast to deploy into any Kubernetes cluster. This makes it easier to migrate between different cloud providers allowing to focus on saving infrastructure costs. Another aspect that lower the mantainence cost is the Kubernetes scheduler. Thanks to its dynamic instance assignation, it allows for a better resource usage making sure we use our available resources the best as possible. In addition, as each microservice is kept small, there's a chance to do a faster development and deployment system.

We can also upgrade the system easily. Each microservice can be independently upgraded using the rolling update feature from Kubernetes. In this part we also

get benefit from the microservice architecture, as we can update each microservice independently allowing for a faster upgrade time compared to upgrading the whole system at once. On the other side, the system can also be expanded easily thanks to the coreography approach. Making it also good to upgrade between technologies or on major releases updates.

8 Related Work

There has been a lot of research on big data analytics. Open source tools like Hadoop or Spark have made available scalable, distributed computation to the general public. In addition thanks to the progress in Big data storage with systems like Cassandra, hosting internal storage has never been easier. However there's not a lot of work regarding container-orchestration for big data analysis systems. There has been a lot work in both fields, but there's not a lot of work about combining them.

The work presented by [6] is probably the closest approach to the system I present here. The main differences are in the Actor system and the orchestration platform. Instead of using Mesos for Orchestration and Akka for actors, I preferred to use Kubernetes with container microservices instead of actors. The stack described also uses Spark Streaming to deploy a Lambda architecture which could be done in the current infrastructure extending the capabilities of the system. The main reason for avoiding Spark Streaming has been that it's micro batching focus could involve to lose some tweets during the normalization process. On the other side, the reason to avoid Akka for actors was that it had similar features than Kafka and Microservices in Kubernetes which makes it a component that can be avoiding by using other tools. In that way, I considered that adding Akka would make the system more complex and therefore more difficult to maintain.

There has also been previous work trying to approach Big Data Analytics to a higher scalability like this paper from 2014 [8]. This system proposal is agnostic of how to deploy it. Leaving the developer to decide how to deploy it. As container and container orchestrated systems were not popular at the moment, there's no mention of them. It tries to propose a scalable Big Data Analytics system using the different Hadoop components. As we have seen previously, Spark is preferred nowadays thanks to its increased performance.

In [7] we see a similar approach using Hadoop and Pig in the batch layer instead of Cassandra and Spark. They also incorpore a streaming layer for real-time analysis by doing a lambda infrastructure. In [10] we can see Zeppelin used as a dashboard interface similarly to our work. However none of this works mention any container orchestrated approach.

9 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.

10 Conclusions

Container orchestrated technologies make it easier to develop Big Data Analytics systems. Their abstraction layer allows for a separation of responsabilities between developers and system operators, allowing them to work without overlapping their work. This opens a gate for innovation on the Big Data Analytics field, as improvements can be developed separately for infrastructure and software. It also opens a gate to new approaches to system state, extracting the state into the ruling container orchestrated system.

In our case, using container orchestration to recreate Project EPIC infrastructure has proved to be easier to scale. It also, has kept reliance from the previous system, abstracting it and making it part of the orchestration system. It has proved to be an easier to mantain infrastructure, as components are smaller and it's easier to make a more continous development and deployment cycle. In addition, container systems like Docker allows for developers to focus more on the application logic instead of worrying about deployment infrastructure. Finally, container orchestrated systems allows for a better reliance by managing the deployment cycle and ensuring a certain amount of deployed instance at any point.

In conclusion, we have proved that container orchestration systems can become a great option when developing Big Data Analytics infrastructures that require a flexible scaling and high reliability. However there are still some limitations. Will we see this approach become a standard the facto? Or will container orchestrated systems only succeed in transactional infrastructures?

A Microservices code

Attached here is a version of the code developed for some of the custom microservices discussed in chapter 4. The event manager UI is not included due to the extensivity of the codebase.

A.1 Twitter Tracker

A.1.1 twitter_tracker.go

```
package main

// OAuth1
import (
  "github.com/dghubble/oauth1"
  "os"
  "bufio"
  "strings"
  "net/url"
  "gopkg.in/Shopify/sarama.v1"
  "log"
)

// Line separator function, detects new lines
func scanLines(data []byte, atEOF bool) (advance int, token []byte, err
     error) {
  if atEOF && len(data) == 0 {
    return 0, nil, nil
  }
  if i := strings.Index(string(data), "\r\n"); i >= 0 {
    // We have a full '\r\n' terminated line.
    return i + 2, data[0:i], nil
  }
  // If we're at EOF, we have a final, non-terminated line. Return it.
  if atEOF {
    return len(data), dropCR(data), nil
  }
  // Request more data.
  return 0, nil, nil
}

func dropCR(data []byte) []byte {
  if len(data) > 0 && data[len(data)-1] == '\n' {
    return data[0: len(data)-1]
  }
  return data
}
```

```
func main() {
  // Get environment variables
  var access_token = os.Getenv("ACCESS_TOKEN")
  var token_secret = os.Getenv("ACCESS_TOKEN_SECRET")
  var consumer_key = os.Getenv("CONSUMER_KEY")
  var consumer_secret = os.Getenv("CONSUMER_SECRET")
  var tokens = os.Getenv("TOKENS")
  var kafka_servers = strings.Split(os.Getenv("KAFKA_SERVERS"),",")

  // Prepare OAuth1 client
  conf := oauth1.NewConfig(consumer_key, consumer_secret)
  token := oauth1.NewToken(access_token, token_secret)
  client := conf.Client(oauth1.NoContext, token)
  v := url.Values{}
  v.Set("track", tokens)

  // Get url for stream
  stream_url := "https://stream.twitter.com/1.1/statuses/filter.json?"
      + v.Encode()

  // Connect to URL with POST
  resp, err := client.Post(stream_url, "application/json", nil)

  if err != nil {
    log.Fatalf("Error while connecting to twitter: %s", err)
    panic(err)
    return
  }

  if resp.StatusCode != 200 {
    log.Fatalf("Error while connecting to twitter, status code returned
        : %d", resp.StatusCode)
    panic(err)
    return
  }

  // Create buffer scanner for request and split by custom function
  scanner := bufio.NewScanner(resp.Body)
  scanner.Split(scanLines)

  // Start asyncronous kafka producer
  producer, err := sarama.NewAsyncProducer(kafka_servers, nil)

  // Close producer before exiting program
  defer func() {
    if err := producer.Close(); err != nil {
      log.Fatalln(err)
    }
  }()

  if err != nil {
    log.Fatalf("Error while bootstraping Kafka producer: %s", err)
    panic(err)
    return
  }

  // Start separate thread to track Kafka produced errors
  go func() {
    for err := range producer.Errors() {
      log.Fatalf("Kafka error: %s", err)
      // Exit application if any error from Kafka
      // Force Kubernetes to recover
```

```
        os.Exit(2)
      }
    }()

    // Main loop to scan request. Only breaks if error from Kafka.
    for scanner.Scan() {
      tweet := scanner.Bytes()
      if len(token) == 0 {
        // empty keep-alive
        continue
      }

      // Send tweet to producer
      producer.Input() <- &sarama.ProducerMessage{Topic: "raw_tweets",
          Key: nil, Value: sarama.StringEncoder(tweet)}
      log.Printf("Tweet received")
    }
    log.Printf("Closing")

  }
```

A.2 Twitter Normalizer

A.2.1 model.py

```
# -*- coding: utf-8 -*-
from __future__ import unicode_literals
import logging
import os
import socket
import uuid
from datetime import datetime
import ujson
from cassandra.cluster import Cluster
from cassandra.cqlengine import columns, connection
from cassandra.cqlengine.management import sync_table
from cassandra.cqlengine.models import Model

CASSANDRA_IPS = list(
    map(socket.gethostbyname, os.environ.get('CASSANDRA_NODES', '
        127.0.0.1').replace(' ', '').split(',')))
KEYSPACE = 'twitter_analytics'

# Tweet model definition as Python Class
class Tweet(Model):
    id = columns.UUID(primary_key=True, default=uuid.uuid4)
    event_name = columns.Text(index=True)
    t_id = columns.Text(primary_key=True, clustering_order="DESC")
    event_kw = columns.Text()
    t_created_at = columns.DateTime()
    t_text = columns.Text()
    t_retweet_count = columns.Integer()
    t_favorite_count = columns.Integer()
    t_geo = columns.Text()
    t_coordinates = columns.Text()
    t_favorited = columns.Boolean()
    t_retweeted = columns.Boolean()
    t_is_a_retweet = columns.Boolean()
    t_lang = columns.Text()
    u_id = columns.Text()
    u_name = columns.Text()
    u_screen_name = columns.Text()
    u_location = columns.Text()
    u_url = columns.Text()
    u_lang = columns.Text()
    u_description = columns.Text()
    u_time_zone = columns.Text()
    u_geo_enabled = columns.Boolean()
    media_url = columns.Text()
    um_screen_name = columns.Text()
    um_name = columns.Text()
    um_id = columns.Text()
    u_followers_count = columns.Integer()
    u_friends_count = columns.Integer()
    u_listed_count = columns.Integer()
    u_favourites_count = columns.Integer()
    u_utc_offset = columns.Integer()
    u_statuses_count = columns.Integer()
    u_created_at = columns.DateTime()
    hashtags = columns.List(value_type=columns.Text)
```

```
    urls = columns.List(value_type=columns.Text)


logging.info('Connecting to cassandra...')

cluster = Cluster(CASSANDRA_IPS)
with cluster.connect() as session:
    logging.info('Creating keyspace...')
    # Keyspace creation
    session.execute("""
            CREATE KEYSPACE IF NOT EXISTS %s
            WITH replication = { 'class': 'SimpleStrategy', '
                replication_factor': '1' }
            """ % KEYSPACE)

    # Establish connection with Cassandra cluster
    connection.setup(CASSANDRA_IPS, KEYSPACE, protocol_version=3)

    #Table creation
    logging.info('Creating table...')
    sync_table(Tweet)


def create_dict(event_key, event_kw, tweet):
    return {
        'id':str(uuid.uuid1()),
        't_id': tweet['id_str'],
        'event_kw': ','.join(event_kw),
        'event_name': event_key,
        't_created_at': datetime.strptime(tweet['created_at'], '%a %b %
            d %H:%M:%S +0000 %Y').isoformat(),
        't_text': tweet['text'],
        't_retweet_count': tweet['retweet_count'],
        't_favorite_count': tweet['favorite_count'],
        't_geo': str(tweet['geo']),
        't_coordinates': str(tweet['coordinates']),
        't_favorited': tweet['favorited'],
        't_retweeted': tweet['retweeted'],
        't_is_a_retweet': 'retweeted_status' in tweet,
        't_lang': tweet['lang'],
        'u_id': tweet['user']['id_str'],
        'u_name': tweet['user']['name'],
        'u_screen_name': tweet['user']['screen_name'],
        'u_location': tweet['user']['location'],
        'u_url': tweet['user']['url'],
        'u_lang': tweet['user']['lang'],
        'u_description': tweet['user']['description'],
        'u_time_zone': tweet['user']['time_zone'],
        'u_geo_enabled': bool(tweet['user']['geo_enabled']),
        'u_followers_count': tweet['user']['followers_count'],
        'u_friends_count': tweet['user']['friends_count'],
        'u_favourites_count': tweet['user']['favourites_count'],
        'u_statuses_count': tweet['user']['statuses_count'],
        'u_created_at': datetime.strptime(tweet['user']['created_at'],
            '%a %b %d %H:%M:%S +0000 %Y').isoformat(),
        'hashtags': list(map(lambda h: h['text'], tweet['entities']['
            hashtags'])),
        'urls': list(map(lambda url: url['url'], tweet['entities']['
            urls'])),
        # Concat names with a space separation
        'um_screen_name': ' '.join(map(lambda um: str(um['screen_name'
            ]), tweet['entities']['user_mentions'])),
```

```
            'um_name': ' '.join(map(lambda um: str(um['name']), tweet['
                entities']['user_mentions'])),
            'um_id': ' '.join(map(lambda um: str(um['id_str']), tweet['
                entities']['user_mentions'])),
            'media_url': ' '.join(map(lambda m: str(m['media_url_https']),
                tweet['entities']['media']))
            if 'media' in tweet['entities'] else None,

        }


session = connection.session
prep_query = session.prepare("INSERT INTO %s.tweet JSON ?" % KEYSPACE)


def save_tweet(tweet, event_key, event_kw):
    session.execute_async(prep_query, [ujson.dumps(create_dict(
        event_key, event_kw, tweet)), ])
```

A.2.2 tweetparser.py

```
import logging
import os
import sys
import ujson
from confluent_kafka import Consumer, KafkaError, KafkaException

EVENT_KEY = os.environ.get('EVENT_KEY', '')
assert EVENT_KEY, 'Event key must be specified as environment variable'

TOKENS = list(filter(None, os.environ.get('TOKENS', '').split(',')))
assert TOKENS, 'Tokens can\'t be empty'

KAFKA_SERVER = os.environ.get('KAFKA_SERVERS', 'localhost:9092')

def main(save):
    conf = {'bootstrap.servers': KAFKA_SERVER,
            'group.id': EVENT_KEY,
            'session.timeout.ms': 6000,
            'default.topic.config': {'auto.offset.reset': 'smallest'}
            }
    # Create Kafka consumer with specified configuration
    c = Consumer(**conf)

    # Custom function for assignment printing
    def print_assignment(consumer, partitions):
        logging.info('Assignment: %s' % partitions)

    # Subscribe to topics
    c.subscribe(['raw_tweets', ], on_assign=print_assignment)

    msg_count = 0
    while True:
        msg = c.poll()
        if msg is None:
            continue
        if msg.error():
            # Error or event
            if msg.error().code() == KafkaError._PARTITION_EOF:
                # End of partition event
                logging.info('%% %s [%d] reached end at offset %d' % (
                    msg.topic(), msg.partition(), msg.offset()))
            elif msg.error():
                # Error
                raise KafkaException(msg.error())
        else:
            tweet = None
            try:
                tweet = ujson.loads(msg.value())
            except TypeError:
                logging.error("Message not json: %s" % msg.value())
                continue
            except ValueError:
                logging.error("Message not json: %s" % msg.value())
                continue

            # Twitter internal messages are discarded here
            if 'text' not in tweet:
                logging.info('Internal message: %s' % tweet)
```

```
                continue
            msg_count += 1

            # Check if tweet should be stored
            if any(token in tweet['text'] for token in TOKENS):
                logging.info('Tweet accepted: %s:%d:%d: key=%s tweet_id
                    =%s' %
                             (msg.topic(), msg.partition(), msg.offset
                                 (),
                              str(msg.key()), tweet['id']))
                save(tweet, EVENT_KEY, TOKENS)

        # Readiness write on first message (used by Kubernetes)
        if msg_count == 1:
            open('/tmp/healthy', 'a').close()


if __name__ == "__main__":
    logging.basicConfig(
        format='%(asctime)s.%(msecs)s%(levelname)s:%(message)s',
        level=logging.INFO
    )
    import model

    logging.info('Event: %s' % EVENT_KEY)
    logging.info('Tracking keywords: %s' % ','.join(TOKENS))
    logging.info('Kafka servers: %s' % KAFKA_SERVER)
    logging.info('Connecting to Cassandra...')
    logging.info('Start stream track')
    if not TOKENS:
        logging.error('Tokens can\'t be empty')
    main(model.save_tweet)
```

A.3 Infrastructure Controller

There's also YAML templates to generate the deployments that are sent to Kubernetes. I don't include them here. To check some sample YAML file see Appendix B

A.3.1 start.py

```
import json
import logging
import os

from kafka import KafkaConsumer

import k8scontroller

bootstrap_servers = os.environ.get('KAFKA_SERVERS', 'localhost:9092').
    split(',')
topic = os.environ.get('KAFKA_TOPIC', 'events')

TEST_TYPE = 'test'
EVENT_TYPE = 'event'
QUERIES_TYPE = 'queries'

UPDATE_ACTION = 'update'
REFRESH_ACTION = 'refresh'
IGNORE_ACTION = 'ignore'


def main():
    consumer = KafkaConsumer(topic, group_id='k8scontroller-eventparser
        ', bootstrap_servers=bootstrap_servers,
                             value_deserializer=lambda m: json.loads(m.
                                 decode('utf-8')))

    for message in consumer:
        value = message.value
        type_ = value['type']
        action = value['action']
        if (action == UPDATE_ACTION or action == REFRESH_ACTION) and
            type_ == EVENT_TYPE:
            data = value['data']
            try:
                if data['tracking'] and data['tokens'].replace(' ', '')
                    .replace(',', ''):
                    k8scontroller.apply_eventparser(data['code'], data[
                        'tokens'])
                    logging.info('Created event partser for event: %s'
                        % data['code'])
            except KeyError:
                logging.info('Message received was not formatted
                    correctly. Message:\n %s' % data)
        elif (action == UPDATE_ACTION or action == REFRESH_ACTION) and
            type_ == QUERIES_TYPE:
            tokens = value['data']
            try:
                k8scontroller.update_queries(tokens)
                logging.info('Updated twitter streaming with queries: %
                    s' % tokens)
```

```
            except KeyError:
                logging.info('Message received was not formatted
                    correctly. Message:\n %s' % value)


if __name__ == "__main__":
    logging.basicConfig(
        format='%(asctime)s.%(msecs)s:%(name)s:%(thread)d:%(levelname)s
            :%(process)d:%(message)s',
        level=logging.INFO
    )
    logging.info('Checking Kubernetes connection...')
    logging.info('Kubernetes current pods ips: %s' % k8scontroller.
        get_pod_ips())

    logging.info('Kafka servers: %s' % ','.join(bootstrap_servers))
    logging.info('Start tracking changes')
    main()
```

A.3.2 k8scontroller.py

```
import os

import logging
import yaml
from kubernetes import client, config
from kubernetes.client.rest import ApiException

KAFKA_SERVERS = os.environ.get('KAFKA_SERVERS', 'localhost:9092')
CASSANDRA_SERVERS = os.environ.get('CASSANDRA_SERVERS', 'localhost')

ACCESS_TOKEN = os.environ.get("ACCESS_TOKEN", "ENTER YOUR ACCESS TOKEN"
    )
ACCESS_TOKEN_SECRET = os.environ.get("ACCESS_TOKEN_SECRET", "ENTER YOUR
     ACCESS TOKEN SECRET")
CONSUMER_KEY = os.environ.get("CONSUMER_KEY", "ENTER YOUR API KEY")
CONSUMER_SECRET = os.environ.get("CONSUMER_SECRET", "ENTER YOUR API
    SECRET")
TWEET_CASSANDRA_VERSION = os.environ.get("TWEET_CASSANDRA_VERSION", "
    1.2.1")
TWITTER_STREAMING_VERSION = os.environ.get("TWITTER_STREAMING_VERSION",
     "1.1.0")


def load_config():
    try:
        config.load_kube_config()
    except:
        config.load_incluster_config()


def get_pod_ips():
    # Configs can be set in Configuration class directly or using
        helper utility

    load_config()

    v1 = client.CoreV1Api()
    logging.info("Listing pods with their IPs:")
    ret = v1.list_pod_for_all_namespaces(watch=False)
    return list(map(lambda x: x.status.pod_ip, ret.items))


def apply_eventparser(event_code, keywords):
    load_config()

    with open("k8sdeployments/tweet_cassandra.yaml") as f:
        name = '%s-event-parser' % event_code
        dep = yaml.load(
            f.read()
                .replace('{{code}}', event_code)
                .replace('{{keywords}}', keywords)
                .replace('{{name}}', name)
                .replace('{{version}}', TWEET_CASSANDRA_VERSION)
                .replace('{{kafka-servers}}', KAFKA_SERVERS)
                .replace('{{cassandra-servers}}', CASSANDRA_SERVERS)
```

```
        )

        k8s_beta = client.ExtensionsV1beta1Api()
        try:
            resp = k8s_beta.create_namespaced_deployment(
                body=dep, namespace="default")
        except ApiException as e:
            resp = k8s_beta.patch_namespaced_deployment(name,
                                                        body=dep,
                                                            namespace="
                                                            default")

        return resp


def update_queries(queries):
    load_config()
    with open("k8sdeployments/twitter_tracker.yaml") as f:
        name = 'twitter-tracker'
        dep = yaml.load(
            f.read()
                .replace('{{a_token}}', ACCESS_TOKEN)
                .replace('{{a_token_secret}}', ACCESS_TOKEN_SECRET)
                .replace('{{c_key}}', CONSUMER_KEY)
                .replace('{{c_secret}}', CONSUMER_SECRET)
                .replace('{{queries}}', ','.join(queries))
                .replace('{{version}}', TWITTER_STREAMING_VERSION)
                .replace('{{kafka-servers}}', KAFKA_SERVERS)
        )

        k8s_beta = client.ExtensionsV1beta1Api()
        try:
            resp = k8s_beta.create_namespaced_deployment(
                body=dep, namespace="default")
        except ApiException as e:
            resp = k8s_beta.patch_namespaced_deployment(name,
                                                        body=dep,
                                                            namespace="
                                                            default")

        return resp
```

B Deploying microservices in Kubernetes

As we stated in chapter 6, all the system has been built to be deployed using Kubernets YAML configuration files. However the process of deploying a microservice is not stated. Here we describe how to deploy Event Manager UI from scratch as an example.

B.1 Docker image

The first thing that needs to be done in to create a docker image. This can be done by defining a Dockerfile for the project. Once that is done, we can create the image by running `docker build -t projectepic/eventmanager-ui .` in the Dockerfile folder.

```
FROM python:3.6-alpine

RUN mkdir /code
WORKDIR /code

# Install dependencies from requirement.txt
ADD requirements.txt /code/
RUN pip install -r requirements.txt

#Add apps code to image
ADD manage.py /code/
RUN mkdir /code/events
ADD events /code/events
RUN mkdir /code/eventmanager
ADD eventmanager /code/eventmanager
RUN mkdir /code/db

# Collect static resources in image
RUN python manage.py collectstatic --noinput

# Expose port 80 from inside the container
EXPOSE 80

# Add start script
ADD start.sh /code/

# Add external mountable volume on the DB folder
VOLUME ["/code/db",]

# Define starting point
ENTRYPOINT /code/start.sh
```

The base image is Python Alpine, which is a low resource image for Python. Thanks to this our new image is not as big as if we used the regular Python image. In addition we include a volume to save state betweet restarts and a port exposed to access the inner defined server. After being created we need to add a tag to the image so that we can decide what version to use on deployment.

Once the image is created and tagged, we need to push it to an Image Registry service. The choice for this project has been DockerHub, but any other can be used as long as it's accessible by the Kubernetes cluster.

B.2 Kubernetes deployment YAML file

This is a YAML configuration file for the Event Manager UI. The important part is the specification section (*spec*). There you set the number of replicas you want to be deployed and specify the template for each pod that will be deployed as part of the deployment. For the pod template you need to specify the image, the resources it will use and the environment variables. We also include a readiness probe, which will be executed on start to check wether or not the application in the container has been start and it's running smoothly. Finally we declare a volume to be mounted on the pod in the db path as defined in the Dockerfile previously. We also declare how to claim the volume by specifying a volume claim. This will ensure that the same volume is mounted between restarts making sure our data is kept.

To deploy we can use either the Kubectl command line properly configured or the web interface by uploading the configuration file. Both ways have the same effect.

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: eventmanager-ui
  namespace: frontend
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: eventmanager-ui
    spec:
      terminationGracePeriodSeconds: 10
      containers:
      - name: eventmanager
        image: projectepic/eventmanager-ui:1.1.8
        ports:
        - containerPort: 80
        resources:
          limits:
            cpu: 100m
            memory: 50Mi
          requests:
            cpu: 100m
            memory: 50Mi
        env:
          - name: KAFKA_SERVERS
            value: kafka-0.broker.kafka.svc.cluster.local:9092,kafka
                -1.broker.kafka.svc.cluster.local:9092
        readinessProbe:
          httpGet:
            path: /
            port: 80
        volumeMounts:
        - name: datadir
          mountPath: /code/db
      volumes:
      - name: datadir
        persistentVolumeClaim:
          claimName: eventmanager-db
```

Bibliography

[1] Kenneth M. Anderson and Aaron Schram. "Design and Implementation of a Data Analytics Infrastructure in Support of Crisis Informatics Research (NIER Track)". In: *Proceedings of the 33rd International Conference on Software Engineering*. ICSE '11. Waikiki, HI, USA: ACM, 2011, pp. 844–847. ISBN: 978-1-4503-0445-0. DOI: 10.1145/1985793.1985920. URL: http://doi.acm.org/10.1145/1985793.1985920.

[2] Kenneth M. Anderson et al. "Design Challenges/Solutions for Environments Supporting the Analysis of Social Media Data in Crisis Informatics Research". In: *Proceedings of the 2015 48th Hawaii International Conference on System Sciences*. HICSS '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 163–172. ISBN: 978-1-4799-7367-5. DOI: 10.1109/HICSS.2015.29. URL: http://dx.doi.org/10.1109/HICSS.2015.29.

[3] Grady Booch. *Object-Oriented Analysis and Design with Applications (3rd Edition)*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 2004. ISBN: 020189551X.

[4] 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.

[5] N. Dragoni et al. "Microservices: How To Make Your Application Scale". In: *ArXiv e-prints* (Feb. 2017). arXiv: 1702.07149 [cs.SE].

[6] Raul Estrada and Isaac Ruiz. *Big Data SMACK: A Guide to Apache Spark, Mesos, Akka, Cassandra, and Kafka*. Apress, 2016.

[7] Zirije Hasani, Margita Kon-Popovska, and Goran Velinov. "Lambda architecture for real time big data analytic". In: *ICT Innovations* (2014).

[8] Han Hu et al. "Toward scalable systems for big data analytics: A technology tutorial". In: *IEEE access* 2 (2014), pp. 652–687.

[9] 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.

[10] A MadhaviLatha and G Vijaya Kumar. "Streaming Data Analysis using Apache Cassandra and Zeppelin". In: ().

[11] William Morgan. *What's a service mesh? And why do I need one? | Buoyant | The Service Mesh Company*. 2017. URL: https://buoyant.io/2017/04/25/whats-a-service-mesh-and-why-do-i-need-one/.

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

[13] Leysia Palen et al. "Crisis in a Networked World". In: *Soc. Sci. Comput. Rev.* 27.4 (Nov. 2009), pp. 467–480. ISSN: 0894-4393. DOI: 10.1177/0894439309332302. URL: http://dx.doi.org/10.1177/0894439309332302.

[14] 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.

[15] Sarah Vieweg et al. "Microblogging During Two Natural Hazards Events: What Twitter May Contribute to Situational Awareness". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '10. Atlanta, Georgia, USA: ACM, 2010, pp. 1079–1088. ISBN: 978-1-60558-929-9. DOI: 10.1145/1753326.1753486. URL: http://doi.acm.org/10.1145/1753326.1753486.