POLITECNICO DI TORINO

III Facoltà di DAUIN

Corso di Laurea in Computer Engineering

Tesi di Laurea Magistrale

Predictive approaches to cloud computing costs reduction

Relatore:
prof. Paolo Garza

Mario Guerriero

Anno accademico 2018-2019

Alla mia famiglia, che mi ha sempre sostenuto e incoraggiato in qualsiasi cosa avessi voluto fare.

Summary

Cloud Computing has become more and more affordable, it has seen an important growth in terms of adoption among all the IT companies in the last few years, overtaking the previously used on-premise solutions. Indeed, Cloud Computing is generally cheaper and easier to set-up with respect to on-premise solutions, requiring no maintenance and coming with more service stability, generally higher up-time and dedicated support from experienced companies like Google or Amazon in case of problems.

The easiness of Cloud Computing, however, may lead developers to misuse computational resources mainly because it is very easy to add more memory or more processing units to any application. Moreover, developers never know the precise amount of resources their applications will need, thus exceeding with resources allocation just to be ready to handle all possible worst cases.

In this work we will address the misuse of cluster resources from two perspectives, focusing our attention on batch jobs. Firstly, we will address the problem of scheduling several jobs into the same clusters, thus implementing resources sharing among jobs. Then, we will focus on automatically resizing cluster's resources (expressed in terms of computational machines) in order to always provide the best fit between cluster's resources and user's jobs. Moreover, we will prove that better scheduling decisions can be taken by exploiting the prior knowledge we have on our jobs in order to build Machine Learning models.

In conclusion, we will show how we managed to obtain a 65% saving in costs of cloud computing in our team by adopting our system as an intermediate layer between the developers and the Cloud Computing platform.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In recent years the usage of cloud computing resources has seen an increase in popularity among companies, which constantly require to face new and more computationally expensive challenges. This phenomenon is also due to the effort of big companies e.g. Google or Amazon, which focus part of their business in offering on-demand computational power to whoever may request it. This represents quite an opportunity for small or medium sized companies or other organizations, e.g. universities, which can now access high computational power without having to set-up any physical infrastructure.

In addition to being easy to use and to require no set-up effort, the cloud computing services offered by third-party companies are also much cheaper for obvious reasons. Indeed, it has been proved that, for a small organization like a university, the operational costs can be cut of 50-60% [24] when using cloud computing on-demand services rather than building an in-home infrastructure. Such an amount of savings definitely attract a lot of companies which are able to start working with a huge amount of computational resources without having to set-up anything nor requiring a specialist for it, thus saving both money and time.

Modern cloud computing services offer a wide range of features, from storage to databases to high availability clusters. Most of those services rely on virtual machines allocated on physical nodes which offer a high degree of freedom. In particular, when running an application a developer can choose the hardware type on which his code will run, being able to define the amount of memory, the amount of disk space and the number of processing cores per virtual machine. If the developer wants to implement additional reliability mechanisms to its application, he can also choose to run it in a distributed fashion, by replicating its execution on several different machines.

The amount of resources is generally quantified in terms of machines, or nodes, which are actually virtual machines executed on the provider's physical hardware. Those virtual machines, of course, are allocated on-demand and are usually billed

on a time basis. This is the case, for example, of Google Cloud Platform, which bills its users for each second of usage of its virtual machines[1]. Because of this billing system, companies usually rely on ephemeral clusters for executing their code. This means that they keep the clusters they use allocated for as long as they are really necessary, deallocating them when there are no more operations to execute. Of course, carefully handling clusters creation and deletion is crucial for effective cost management.

In order for their services to be more flexible, cloud computing providers usually have several different types of machine, in order to satisfy any possible computational need. Of course, the general rule is that the more the resources, the higher the price.

Since costs of cloud computing could still be significant, especially for small organization and companies, cloud computing vendors recently started to sell their services at highly discounted rates for situations in which high availability is not a strict requirement. Indeed, both Google and Amazon, sells spare time computational resources, which they respectively call Preemptible Virtual Machines[2] and Spot Instances[3]. The major drawback of those discounted instances (which can let the user save up to 80-90% of the cloud computing costs), is that they could be shut down at any time, with short notification.

Spare time computation machines should not be used if constant availability is mandatory for certain tasks. However, if an application has proper failure recovery mechanisms and the user can tolerate a small downtime, those type of machines could still be considered for daily tasks. In particular, discounted machines, such as Preemptible Virtual Machines by Google, could be very useful for batch jobs, such as those relying on Apache Spark[4], since those applications execution flow is generally handled by cluster resources schedulers, which internally implement failure recovery mechanisms, lightening the developer from implementing additional features into their programs.

Cloud computing on-demand services, however, do not have only positive characteristics. In fact, they also present some limitations which are not easy to overcome.

One of the most important is certainly the privacy and security concerns which may derive from letting a third-party entity to hold all the data for our organization. A solution to this problem is certainly using encryption on the user side, who should encrypt all his data before sending them through the network so that the cloud provider is able to store and operate on them but cannot read their content [22]. However, this solution introduces overhead to the cloud service user.

---

[1] https://cloud.google.com/pricing/principles, accessed on 10^{th} October 2018

[2] https://cloud.google.com/compute/docs/machine-types, accessed on 10^{th} October 2018

[3] https://aws.amazon.com/ec2/spot/, accessed on 10^{th} October 2018

[4] https://spark.apache.org/, accessed on $10th$ October 2018

Another problem with cloud computing services may be the absence of properly defined service level agreements (SLA) [22]. With the increasing quality of cloud computing providers, however, this problem has become less and less relevant. Indeed, Google Cloud Platform (GCP), for example, is able to guarantee a 99.978% availability and no scheduled downtime[5] which is acceptable in most of the cases. Of course, if the SLA offered by the provider should not be considered sufficient for the client company, the only solution would be to implement an on-premise infrastructure, trying to do better then the provider itself.

In general, the common characteristic that stands out from all the cloud computing limitations is the fact that we lose direct control on our resources and this, combined with the freedom of resources choice, can be sometimes dangerous in terms of costs. Indeed, since a developer in the company does no longer have a clear physical view of the resources he is using, he may be tempted to abuse the simplicity of adding more computational power to its application, thus resulting in improper resources' usage which waste most of the benefits of using cloud computing services.

Preventing the described misbehavior from happening is the goal of the project which was designed and developer while writing this thesis work: OBI, which stands for Objectively Better Infrastructure.

OBI's main goal is to abstract the resources' definition from the application submission process in a cluster environment. In particular, with OBI, when a user wants to submit a distributed application relying on e.g. Apache Spark, he would be dismissed from having to define the set of machines on which his software will execute. Instead, OBI will take care of dynamically provision computational resources as they are needed, trying to always allocate as few virtual machines as possible, in order to be able to reduce the costs as much as possible. It means that OBI aims at reducing the cost of cloud computing operations to the short amount possible.

Relying on the knowledge of the environment and the workload in which OBI will be deployed, we were able to build a set of predictive machine learning models (better described in the next sections) which assist our system in taking smart decisions for what concerns resources allocation. Indeed, with OBI we adopted a novel approach for cloud computing resources allocation in which we perform two main tasks: we pack homogeneous jobs into the same clusters; we dynamically provision more resources (or eventually we deallocate them) as they are needed.

Moreover, as we will better explain in later sections, OBI strongly relies on Preemptible Virtual Machines (since it is currently being used in a Google Cloud Platform environment) to further reduce the operational costs.

This thesis work is organized as follows:

[5] https://support.google.com/googlecloud/answer/6056635?hl=en, accessed on 10^{th} October 2018

- In Chapter 2 we will discuss already existing solutions similar to OBI and we will go through the literature on which OBI's components are based
- In Chapter 3 we will briefly describe the experimental environment in which OBI was deployed and tested
- In Chapter 4 we will provide an overview of the OBI architecture and of its components
- In Chapter 5 we will highlight the key features on the module which brings Machine Learning feature to OBI
- In Chapter 6 we will analyze our results
- In Chapter 7 we will describe the lessons we learned and how to improve our system even further

Chapter 2

State of the Art

In this chapter, we will briefly go through the state of the art in the two problems we had to face for bringing machine learning capabilities into job scheduling optimization: job duration prediction and statistical cluster resources autoscaling. Moreover, at the end of this chapter, we will analyze one already existing solution which could be compared to OBI.

2.1 Job Duration Prediction

In the context of OBI, by job duration prediction we mean the process which, given a certain prior knowledge about a Spark job we are trying to execute on our cloud computing platform, returns as output an estimation of how long that job could possibly take to complete.

The prior knowledge we have comes from the fact that we usually know which kind of jobs our platform will be required to execute. Being able to categorize jobs into similar groups has an important relevancy. For example, in [27] the authors have been able to predict execution duration for their jobs with less than 25% error in 95% of the cases they were analyzing just by grouping them into similar groups. By similar groups we mean jobs which execute the same code but on differently sized input.

In our case, we are able to cluster our jobs into groups characterized by a similar workload and we also have full access to their source code. Therefore we can extend our prior knowledge with additional information about the input data that each of them will have to process. Several studies, such as [27] and [34], proved that having knowledge of the input of a certain job can drastically improve the predictive capabilities of a model.

In the literature, there are also other ways of characterizing the workload. Indeed, while the previously mentioned approach focuses on jobs input, other approaches,

e.g. [18], look at the workload as a cluster scale property. For example, in [18], the term workload refers to the number of submitted jobs in the whole cluster. In their case, they aimed at allocating the proper amount of resources in the cluster by trying to predict how many jobs would be submitted at a certain point in time. However, this case is quite different from ours, since we do not have those kinds of variability in our environment. Indeed, as we will explain in more details in Chapter 3, we have enough knowledge both of our jobs and of when they will be executed that we will not need to predict the number of jobs which will be submitted to OBI in a certain time window.

After the prior jobs knowledge is defined, the next natural step in designing a system capable of predicting jobs run-time duration is building a predictive model. Predicting the duration of cluster jobs is a problem of supervised learning[1] and, as such, it requires a set of data points to produce general hypotheses which are capable of generating reliable predictions for unseen input values. Our problem, in particular, is a regression problem[2], which differs from other supervised learning problems, e.g. classification[3], from the fact that the so-called target variable (the output of our model) is continuous rather than discrete.

Datasets for our kind of problem are generally rare and not easy to obtain. Moreover, each particular problem has so specific requirements that it is impossible to find a set of data points that could be so general to fit any possible case. In fact, as we already, said our model will exploit prior jobs knowledge. Therefore it is quite obvious that a specific dataset has to be built.

For the reasons highlighted above, all the previous studies which tried to perform job duration prediction, e.g. [12] or [27], had built their own datasets. Such datasets, of course, contains the information about the job duration (which is the target variable of the final model) plus other values, depending on the approach which was chosen.

In the literature there exist also alternative approaches, which do not rely on details about the workload of each job but tries to reconstruct their execution flow by simply looking at their source code. For example, in [12], the authors tried to represent each job as a sequence of stages and, on top of them, they tried to build a Hidden Markow Model[4] (HMM) capable of representing the job evolution. Based on their HMM, they were able to predict when a running job would terminate with up to 93% accuracy. However, the major downside of such an approach is that each

[1] https://en.wikipedia.org/wiki/Supervised_learning, accessed on 12^{th} October 2018

[2] https://en.wikipedia.org/wiki/Regression_analysis, accessed on 12^{th} October 2018

[3] https://en.wikipedia.org/wiki/Statistical_classification, accessed on 12^{th} October 2018

[4] https://en.wikipedia.org/wiki/Hidden_Markov_model, accessed on 18^{th} October 2018

job log will have to contain specific messages, in order to be able to define a set of states and transitions between them.

The complexity of the HMM based approach caused us not to use it in our further analysis.

2.2 Machine Learning based Autoscaler

Autoscaling is the practice of automatically adding (scale up) or removing (scale down) resources for an application deployment to meet performance targets in response to changing workload conditions [32].

According to [20], there exist several different autoscaling approaches. The most popular among cloud service providers is certainly the one which involves the definition of certain threshold-based rules. Those rules basically define a number of resources which, if exceeded, trigger the scaling operations. This is the default and general approach adopted by Google [4] and Amazon [1] for their platforms.

However, the focus of this study is not on threshold-based autoscaler approaches but rather on those approaches relying on statistical learning algorithms. The approaches in which we are interested in can be divided into two parts: those based on reinforcement learning [17] and the ones based on more traditional machine learning classification and regression algorithms.

Reinforcement learning is a particular machine learning approach which different from other techniques e.g. supervised learning by not having any training set at all, but simply relying on experience and environment observations. As in any reinforcement learning approach, also in the case of autoscaling, we have to define three components: the environment, the state space and the reward function. Of course, the different definition of the three mentioned components highly influence the final algorithm and has a huge impact on its performances.

In [14], for example, the authors considered the state to be a triple of value (w, u, p), where w is the total number of users requests observed in a certain time window, u is the number of allocated VMs in the same time frame and p measures the performances in terms of average response time. Based on this state definition, in [14] they try to choose the best action among a set of three different possibilities (add a new VM, remove an existing VM or do nothing) in order to maximize the reward.

Reinforcement learning based approaches have the great advantage that they do not require any training dataset as they learn on their own. However, it could be quite expensive to train them as a lot of tries have to be made for it in order to produce acceptable results and, in our environment, doing a lot of tries also corresponds to spending a lot of money.

As already mentioned, an alternative statistical based approach to autoscaling

is to use more traditional classifier or regressor models. Moreover, autoscaling problems could be treated as a resource usage prediction, using e.g. time-series analysis, in which, after being able to know how many resources will be used at a certain time, we can react consequently in advance. This is the case of [15], in which the authors propose a model which takes into account both the current time window resource usage metrics and the historical data in order to improve its performances.

Along with resource usage forecast methods, also more automatic approaches have been proposed, in which the model is able to predict not only the future resources usage but also the scaling factor (which is the number of nodes we need to add or remove).

An example of this is [33], in which the authors developed a system, called MLScale, which claims to be an application-agnostic, machine learning based autoscaler. MLScale is composed of two major components: a neural network performance modeler (whose architecture is shown in Figure 2.1), and a set of regression models for predicting cluster metrics after scaling operations. The neural network is used to predict a performance metric (which in the case of MLScale is the response time), given a certain cluster status. The input to the mentioned neural network are the following parameters average by the number of nodes in the cluster at the moment of the predictionion:

- Number of requests received per second
- CPU usage averaged across cores
- Number of interrupts generated per second, 4. CTXSW: Number of context switches per second, 5. KBIn: KB of data received per second,
- Number of packets received per second
- KB of data sent per second
- Number of packets sent per second

The neural network prediction is triggered every time a timeout expires and, if the predicted performance falls below a certain SLA, the next phase is executed.

In its seconds stage, MLScale tries to predict the optimal scaling factor by predicting the resources usage after scaling with $m\prime = c_0 + c_1 m\frac{w}{w+k} + c_2 m\frac{k}{w+k}$, where m is the metric before scaling, $m\prime$ is the metric after scaling, k is the scaling factor, w is the number of available nodes before scaling and c_0, c_1 and c_2 are three parameters learned through a linear regression problem resolution.

The MLScale collected the dataset they needed by simulating jobs execution using clusters with different naïve scaling policies such as response time-based scaling or CPU threshold based scaling [33].

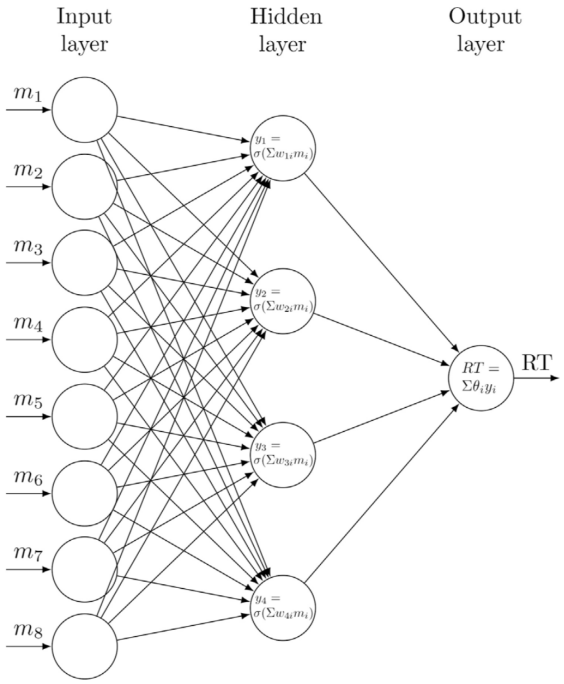

Figure 2.1. MLScale Neural Network Architecture

As we will show later, our autoscaler was heavily influenced by the one described in the MLScale paper.

2.3 Existing Solutions

Efficient cluster resources scheduling is quite a relevant problem in modern industries relying on cloud computing services. As we already mentioned, those problems are usually quite specific to the requirements of a certain company (or a certain team within a company) may have. Therefore it is really difficult to provide a system capable of generalizing its features to the majority of the possible cases.

We could not find any real alternative to OBI in the literature as it looks like no company ever decided to pack together several well-known studies into one single cluster resources manager if we consider machine learning based approaches.

The only system we could find which is somehow similar to OBI is Spydra [9], an attempt to migrate to Google Cloud Platform while hiding the complexity of ephemeral clusters creation process made at Spotify.

Chapter 3

Experimental Environment

In this chapter, we will explain what is the experimental environment in which OBI runs. Also, we will highlight the characteristics we found in the jobs which we had the need to take care of.

Moreover, along with the tools and the platforms which were used, we will give a brief introduction of Delivery Hero, the company in which OBI was developed.

3.1 Delivery Hero

All the effort made while designing and developing OBI were done in Delivery Hero[1] (whose logo is shown in Figure 3.1) head quarter in the center of Berlin, Germany.

Delivery Hero was founded in 2011 with the main mission to revolutionize food delivery industry by offering easy to use them in more than 40 countries spread in 4 continents.

Specifically, OBI was developed in the Data & Insights team of the German department of Delivery Hero with the goal of minimizing the costs of running the daily tasks which make data ready for the analysts who will then perform data mining on them with the goal of supporting business decisions. However, OBI was also made open source and is now available through the company's official Github profile[2] with the hope that somebody else could find it useful and that the other teams within the company will use and contribute to its development and maintenance.

[1] https://www.deliveryhero.com, accessed on 19^{th} November 2018

[2] http://github.com/deliveryhero/obi, accessed on 18^{th} December 2018

Figure 3.1. Delivery Hero Logo

3.2 Google Cloud Platform

Google Cloud Platform (GCP) is the cloud computing service of choice in Delivery Hero. It is the Google suite of tools and services which allow any customer to run its code on a distributed architecture of data centers spread all around the world which are the same Google uses for its popular services.

GCP provides a long list of services[3], among which we used mostly the following:

- Compute Engine: which was used for deploying OBI on a set of machines running Kubernetes[4] as well as for running the jobs scheduled by OBI scheduler
- Storage: which was used as a distributed file system to share OBI's Machine Learning models and to store the datasets we used
- Stackdriver: used to retrieve historical clusters resources metrics while building our datasets
- Dataproc: responsible for handling Hadoop clusters, it is the component on which OBI operates for performing its cost-effective optimizations
- Pubsub: used to automatize the collection of points in our job duration dataset as explained later in this work
- Service Accounts: used in place of credentials for granting OBI with the necessary permissions to perform its tasks

[3] https://cloud.google.com/products/, accessed on 10^{th} October 2018

[4] https://kubernetes.io/, accessed on 10^{th} October 2018

An important concept within GCP environment is the idea of projects. Indeed, each customer or customers group is assigned with a unique project id (chosen by the user himself) which serves as a names space identifier to group resources. Resources in different namespaces can not access each other unless the customer creates his own network interface [3]. It is important to mention that, in our case, we have always operated within the context of the same project and OBI itself, at the moment, does not work across multiple GCP projects. Therefore a strict requirement is to deploy OBI within the same project of the resources it will have to handle.

3.2.1 Dataproc Clusters

Dataproc is the GCP service which can be used to manage Hadoop clusters to run Spark or Hadoop clusters. It claims [2] to be fast and cost-efficient, requiring around 90 seconds for a cluster to start and costing around 1 cent per computational core per hour. Moreover, Dataproc is well integrated into the GCP environment, allowing users to easily use Google Cloud Storage or Stackdriver services.

The clusters created through the Dataproc services use YARN [31] as cluster scheduler. A peculiar characteristic of YARN is its failure resistance features for its components.

YARN scheduler executes in a particular node of the cluster called master node. This node hosts the so-called YARN Resource Manager, which is responsible for managing all the other executor nodes. Each executor node has its own Node Manager, which is in charge of communicating with the Resource Manager through a heartbeat service.

Whenever a new application is submitted, a new service called Application Master is allocated which is in charge of orchestrating an application execution life cycle and recovering it in case of failures by properly requesting resources to the Resource Manager.

Resource are handled in terms of containers, which are fixed bundles of memory and cores to be assigned to an application.

Although YARN offers reliable failure recovery mechanisms for the above-mentioned components, it does not directly offer any service in case of containers failure. Indeed, this failure recovery mechanisms are left to the application framework which, in our case, is Apache Spark [35].

Failures of containers are not a particular problem in our environment because of the Resilient Distributed Dataset (RDD) abstraction used in Spark. Indeed, Spark RDDs always contain enough information to reconstruct the data on which the user was operating from persistent storage, having a quite simple and efficient failure recovery process.

Therefore, in our environment, the failure of few nodes is not a problem. This is quite an important feature, especially because our original goal was to be able

to work with Preemptible Virtual Machines (better describe later in this chapter) which could be removed from our cluster without enough notice thus potentially causing troubles to our jobs.

Preemptible Virtual Machines

Preemptible VMs offer the same machines and services as the normal Dataproc's VMs with the constraint that they could be killed at any moment of their life. Before getting killed, preemptible VMs receive a notice and they have 30 seconds to gracefully shutdown, otherwise, after the timeout has expired, they are forcibly shut down and all the processes running on them would be killed [6].

Each preemptible VM instance could last at most 24 hours[5], after which it will be automatically killed. However, it must be stated that as soon as a preemptible VM is killed, Google Dataproc reallocates another machine with the same resources for replacing the one which was preempted. Of course, in the interval of time during which the machine is unavailable, Google does not bill the resources.

Generally, Google Cloud Platform avoids preempting too many instances from a single project and, while choosing which instance to preempt, will prefer the most recent ones [6]. This strategy helps minimize lost work across your cluster. Moreover, despite the fact that preemptible VMs have no guarantee of SLA at all, Google claims [6] that, for a single GCP project, the average weekly preemption rate varies between 5% and 15%. However, those estimations were obtained simply by observations and there is no certainty on the fact that they could represent realistic scenarios.

In conclusion, the main advantage and the only reason for using preemptible VMs is their price. Indeed, their price is fixed and they are up to 80% cheaper than normal VMs, while still offering a similar service in terms of resources.

As we said earlier, when we discussed Dataproc, Spark applications running in YARN have quite effective and automatic failure recovery mechanisms. While experimenting with preemptible VMs instances we observed that the preemption of a machine is not a big problem and in very few cases it leads to the failure of the entire application we were running. In fact, we observed application failure only when a relevant number of nodes were preempted all at the same time. However, since Google states that it tries not to preempt too many machines from the same users, we labeled those cases as very unlikely and we provided no mechanisms for handling them in OBI. If such a case happens, the user would just be informed that his job crashed and he will have to re-submit it.

[5] https://cloud.google.com/preemptible-vms/, accessed on 10^{th} October 2018

3.2.2 Workload Characteristics

OBI will be in charge of executing the Spark jobs for our team in Delivery Hero. Specifically, those jobs are part of an ETL[6] (Extract, Transform, Load) pipeline and are mainly of three types:

- CSV-Load jobs
- IL Generator jobs
- BQ Generator jobs

The input of the CSV-Load jobs comes from 17 different input sources, which we will call backends, while the input of the IL and BQ Generator jobs comes from 3 different backends which are a subset of those on which the CSV-Load jobs operate.

CSV-Load jobs can have a certain type, each of which acting on one of the 17 possible backends. Indeed, we can have the following CSV-Load jobs classification:

- CSV-Load recreate_file
- CSV-Load find_changes
- CSV-Load update

IL Generator jobs, instead, may have up to 28 different types (depending on the backend on which they work).

Figure 3.2, Figure 3.3 and Figure 3.4, respectively show the average daily duration of the CSV-Load recreate_file, find_changes and update jobs for a period of time between 2^{nd} September 2018 and 4^{th} October 2018. As we can see, recreate_file jobs are certainly the fastest ones, with an average duration 13.96 seconds in the considered period. On the opposite side, update jobs are the slowest, lasting on average 950.70 seconds against the 574.73 seconds on the find_changes jobs.

The above average duration values are lowered by the fact that most of the 17 backends which are used by CSV-Load jobs are quite small thus reducing the average measurement. However, if we consider the biggest backends only, a CSV-Load find_changes or update job can last up to 1 hour.

IL and BQ Generator jobs, instead, are generally quite fast to execute and they last on average around 2 minutes, with the slowest jobs never lasting more than 10 minutes. All the backends on which those jobs operate shows similar behaviors,

[6] https://it.wikipedia.org/wiki/Extract,_transform,_load, accessed on 22^{nd} October 2018

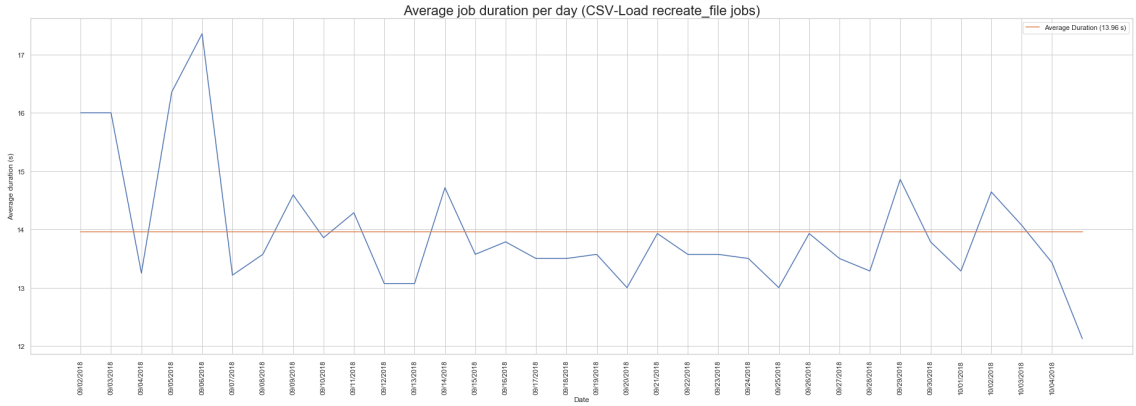

Figure 3.2. Daily duration for CSV-Load recreate_file jobs

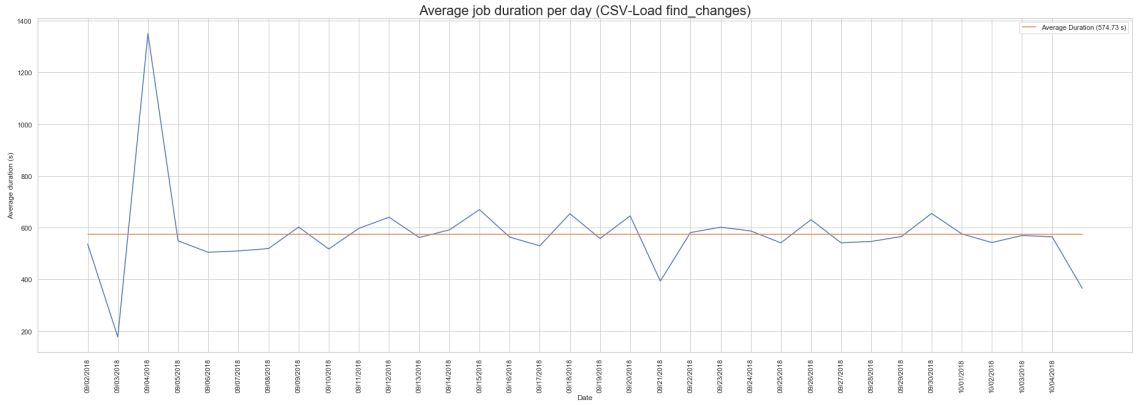

Figure 3.3. Daily duration for CSV-Load find_changes jobs

meaning that there is no backend which is considerably slower with respect to the others.

Moreover, we know a priori that some IL and BQ generator jobs could be grouped together as they do not show any inter-dependency. For this reason, we will not provide a job duration prediction for those jobs. In fact, we decided to pack them into similar clusters by simply using a count based scheduler policy, which is better-detailed in Section 4.1.1.

The above mentioned duration values are obtained by observing the daily execution of each pipeline job. In this context jobs execute in clusters which are often over-provisioned, meaning that most of them are given with more resources than needed.

More detailed analysis of the duration of the jobs we will be submitting will be shown in Chapter 5.

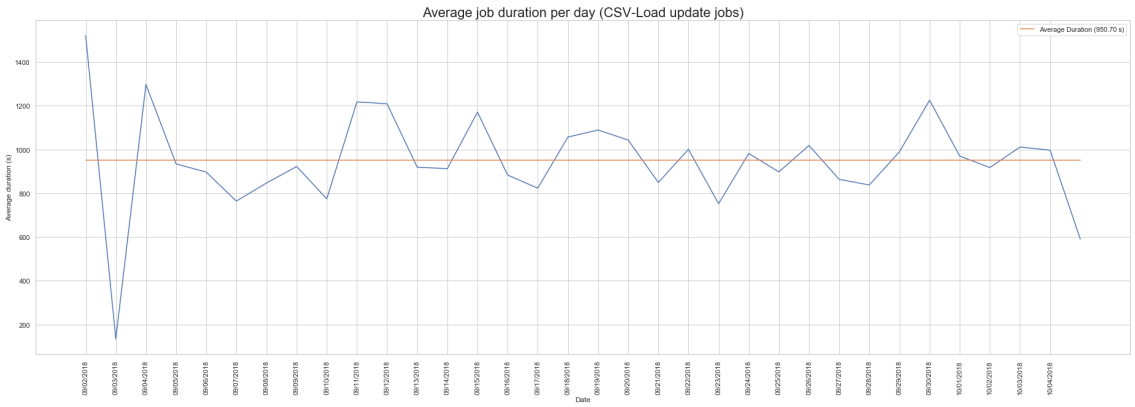

Figure 3.4. Daily duration for CSV-Load update jobs

Chapter 4

Architecture

In the current chapter, we will discuss the general OBI architecture and we will give a brief explanation of what each component does. The details about the predictive module, which is the hearth of OBI's smart decisions, are left for Chapter 5.

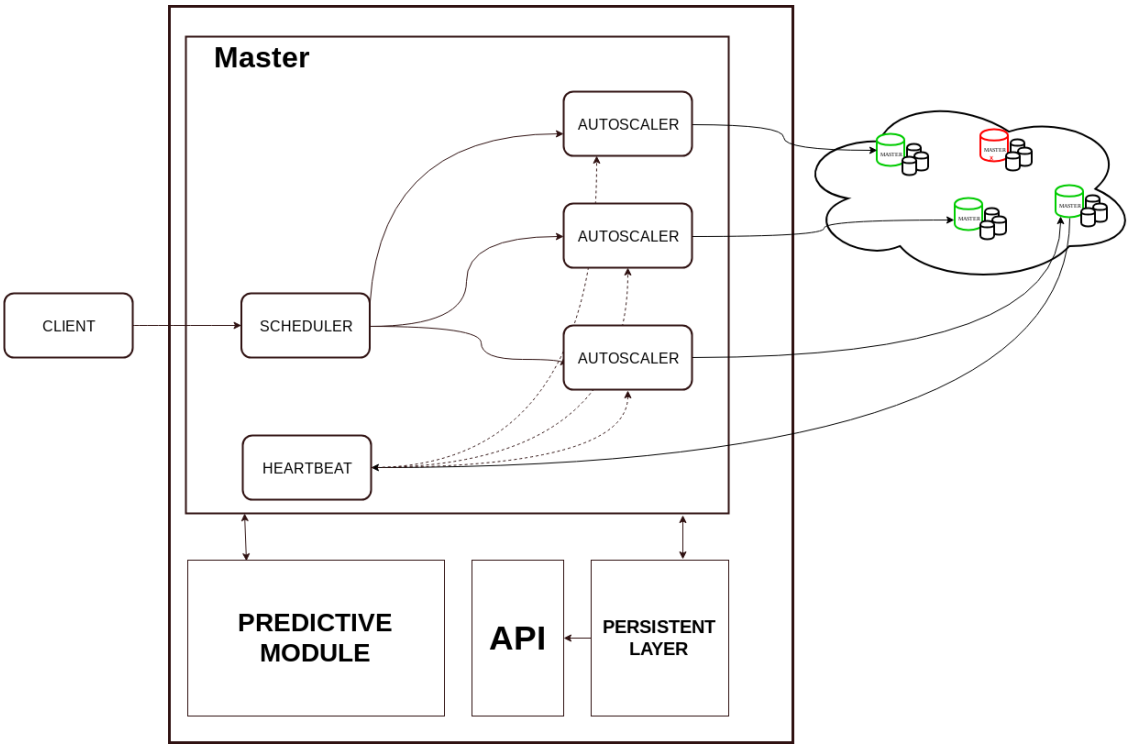

Figure 4.1. High Level OBI Architecture

The general design pattern we adopted while designing the OBI architecture,

visible in Figure 4.1, was the microservices pattern [23]. Microservices architecture requires that each component of the system is developed independently from the others. In this way, the developers have the advantage of being able to work autonomously with the drawback of having to define common communication interfaces between the different components. This also allowed us to use different languages according to the needs of each component.

Before delving into the details of each OBI's component, we must state the three principles around which it was designed:

1. **Modularity**: each component had to be fully replaceable by another one, as long as it respects a certain inter-component communication interface

2. **Run everywhere**: the whole system had to be independent of the deployment environment we choose for our case (Kubernetes)

3. **Lightweight**: the system should optimize operational costs, therefore it does not have to have a huge impact on them

The reason behind the above mentioned principles is that, despite the fact that we built OBI especially for our environment, we want it to be easy to understand and to adapt to other teams within our company or elsewhere.

Regarding the modularity aspect, each component shown in Figure 4.1 can be replaced by an analogous one, as long as it implements the same communication interfaces. Although the scheduler and the autoscaler have to be part of the master component, as it will be shown in the next sections, they are easily extensible through user defined policies.

Since each architectural component is fully independent from the others, the "run anywhere" principle can be easily implemented by deploying each component on the desired system as long as it is not isolated from the others.

Moreover, as it will be clearer later, OBI's components are designed to implement the minimum functionalities possible, requiring little to no time to start and be fully operational.

We will now dedicate a section to each of the OBI's components.

4.1 Master

The OBI Master component is the main endpoint to which users send requests. Indeed, it is responsible for collecting job submission requests and for dispatching them to the appropriate services. In fact, the master component is a collection of services including the scheduler, the set of autoscalers and the heartbeat module.

Since the master, as already said, is a sort of interface to the core services offered by OBI, its major role is to implement an interface between the outside world and OBI. In particular, we choose to design it to be a RPC[1] (Remote Procedure Call) server capable of dispatching requests. We favoured RPC over plain-text messages because all our components are developed and evolve together, therefore we can still tolerate a certain degree of coupling between them. In fact, by using RPC communication, a slight modification in the server has to be reflected on the client. This design choice was made for all the other components (except for the API one) for the same mentioned reasons.

The RPC tool of choice was gRPC [5], a high-performance framework which can work easily across multiple languages and platforms, relying on Google Protocol Buffer[2] serialization mechanism. These particular characteristics made gRPC our choice for implementing the communication between each of the OBI components.

4.1.1 Scheduler

The scheduler service is the first one to which the master forwards job submission requests from the clients. It is the one responsible for handling the prioritization of the jobs which are managed according to their level of importance.

In general, the scheduler has two separate ways of handling job submissions:

- if a job having maximum priority is submitted, the scheduler allocates a new cluster whose life cycle will be entirely bound to that job and whose resources will never be shared among other
- if jobs with lower priority are submitted, the scheduler tries to pack them in bins, alongside with jobs of the same priority band

While the first approach does not need further explanations and it is generally used for those jobs which are considered to be crucial, it is worth going into more details of the second one, which represents one of the major sources of the benefits we get by using OBI.

Each bin in the OBI's scheduler has two main characteristics: a policy and a timeout. The policy describes the way jobs should be packed together into bins. The timeout, instead, defines the waiting time before a bin of jobs is submitted. The most interesting part of each bin is certainly the policy.

Policies may be of two types:

[1] https://en.wikipedia.org/wiki/Remote_procedure_call, accessed on 22nd October 2018

[2] https://developers.google.com/protocol-buffers/, accessed on 22nd October 2018

- count based: jobs are added to the bin up to a certain number
- time based: jobs are added to the bin based on their estimated duration, e.g. we may want to have bins of jobs and we want each bin to last at most one hour

While the first policy type relies on a counter which is updated as we add more jobs to the bin, the second one relies on the job duration predictions coming from the predictive component. Both those policies have the goal of making the clusters as homogeneous as possible, trying to avoid overload a single cluster while putting only small jobs into others.

The different scheduler priority bands, as well as their associated policies and timeout, are both configurable by the users. The only restriction OBI imposes is to have a maximum priority level for dealing with highly important jobs. However, each lower priority band can have its own policy and its own parameters e.g. maximum amount of jobs per bin or maximum duration per bin.

4.1.2 Heartbeat

Each time OBI allocates a new cluster, it forces its master node to send back heartbeat messages at a regular interval of time. Those heartbeat messages contain the status of the cluster including the resources utilization in the moment of sending and the number of allocated nodes.

Each heartbeat message contains information about the available, used and pending memory and processors, as well as the number of active nodes.

The heartbeat services serve to two purposes. Firstly, it is used by the autoscaler to collect information about the cluster status while deciding whether to scale or not. Moreover, it is used by the master itself as a failure resistance mechanism. In fact, if the master fails, it may be subject to losing information about the currently running clusters. However, the heartbeat mechanism plus the persistent storage module allow a failed master to recover its status immediately after its start-up process.

4.1.3 Autoscaler

Autoscaling is one of the most important features offered by OBI Master component and it is the one responsible for dynamic allocation of cluster resources according to the needs of the running jobs. Therefore, it is also one of the services which has the biggest impact on cost reduction for the managed cloud computing platform.

Every time OBI's scheduler decides to create a new cluster, it attaches to it an autoscaler running in a separate routine. Then the autoscaler will choose its actions based on a given policy.

In fact, the autoscaler service is pluggable with policies which determine how the actual scaling is performed. Several policies have been developed in OBI, most of them threshold-based and one based on machine learning models as better explained in Chapter 5.

After an autoscaler is attached to a running cluster, its policy is triggered at regular intervals of time. Whenever the policy is triggered, it checks the cluster status by accessing a list of the most recently received heartbeats and takes scaling decisions accordingly. Since heartbeats are sent each 10 seconds and the heartbeats window analysed by the autoscaler contains 6 heartbeat slots, each time the autoscaler is triggered it has at its disposal the information about the last minute of life of the cluster.

The life cycle of an autoscaler routine ends when a cluster is deallocated from the cloud computing service provider infrastructure.

4.2 Database

One major challenge with OBI being a distributed application was failure resistance. Indeed, OBI containers and machines may fail but this does not have to have relevant impact on the overall system functionalities. Here failure resistance is addressed only from a point of view of the Master component. In fact, other components were designed in order not to have a hard state which would require restoration procedures.

In order to keep it simple, OBI's master is not replicated and was developed to be as lightweight as possible, requiring a negligible amount of time (less than 1 second) to start and be ready to serve clients requests. This particular characteristics are those around which we developed our failure resistance solution.

In order to make OBI fully failure resistant, we needed to store its information about running jobs and clusters in a persistent way, ensuring that they would not be lost in case of failures of the master. Although cluster information can be easily reconstructed after failures thanks to the heartbeat mechanism, we still needed to solve the problem of jobs information.

The solution for failure resistance we adopted was to design an additional component into OBI's architecture: a database capable of holding all the information which may be needed by OBI to reconstruct its state in case of failures.

OBI's database was designed to hold the information shown in Figure 4.2. Basically, for each cluster we needed to hold its status and its cost among other operational variables. As it is visible in Figure 4.2, for the cluster we choose to use a composite primary key made of both the name and the cluster creation timestamp just because cluster names are randomly generated and they may overlap after several job submissions. The job table is bigger and it had to contain all the jobs details

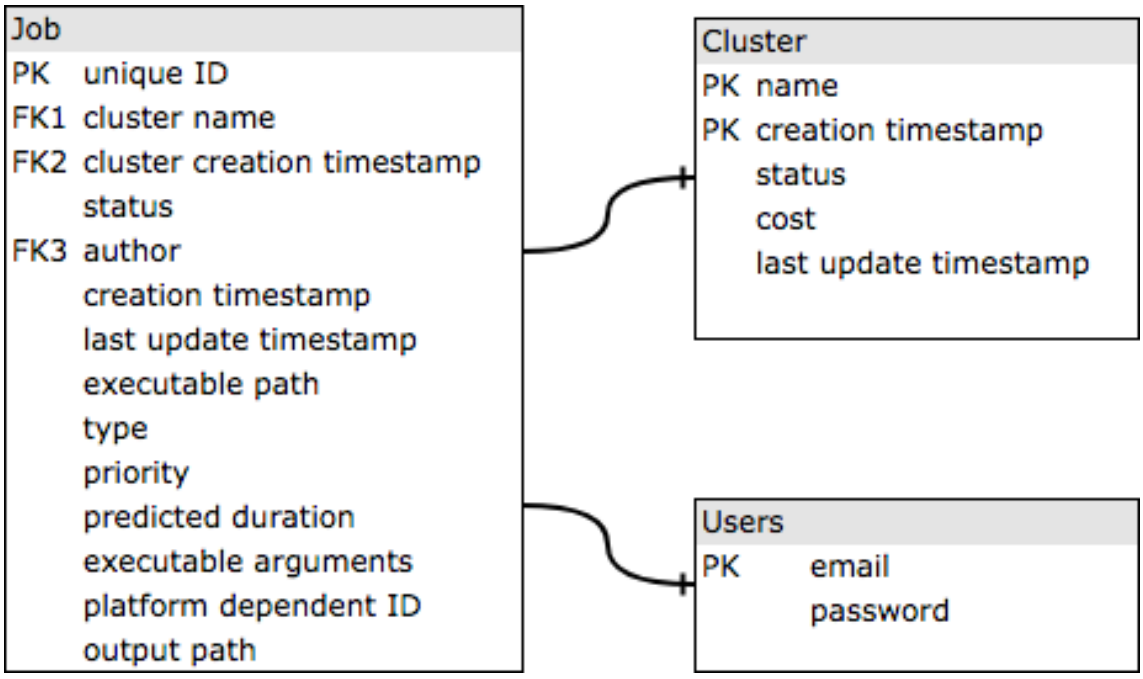

Figure 4.2. Database tables high level description

e.g. the priority, the executable path, its arguments and an ID to recover the job from the platform to which it was submitted, plus links between jobs and clusters and between jobs and users. The users table was added purely for monitoring capabilities. Indeed, OBI's administrators may be interested in understanding which user spent more money than others.

By adopting a plain database store on some persistent disk we would have had to design a whole new system, capable of guaranteeing failure resistance also at a database level using replication and synchronization mechanisms. Designing such a solution, however, would have gone out of the scope of OBI as its complexity could have required to start a whole new project. For this reason, we adopted Stolon [10], a manager for PostgreSQL[3] high availability which guarantees failure resistance by replicating the same database several times.

Stolon is composed of three main components as depicted in Figure 4.3:

- keeper: it manages a PostgreSQL instance converging to the clusterview computed by the leader sentinel
- sentinel: it discovers and monitors keepers and proxies and computes the

[3] https://www.postgresql.org/, accessed on 22nd October 2018

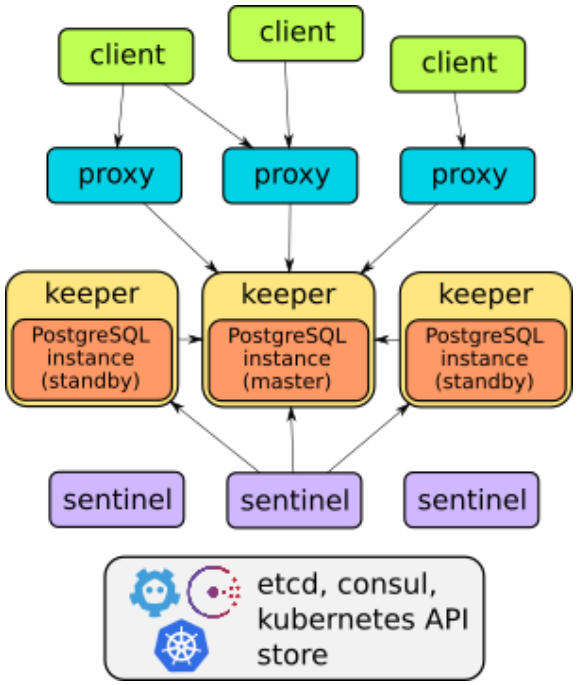

Figure 4.3. High Availability Stolon Database Architecture

optimal clusterview

- proxy: the client's access point. It enforce connections to the right PostgreSQL master and forcibly closes connections to old masters

Since we are using a microservices architecture embracing the modularity principle as described above, the Stolon database component is fully replaceable by any other database manager of choice. Indeed, the OBI Master has to be configured to know where the database is and our choice of using Stolon was just driven by the motivations highlighted above. Of course, any other user of OBI who should not prefer using Stolon over another solution is totally free to replace it.

4.3 API

Despite the fact that it was not directly in the interest of the main OBI project, we created the additional requirement to make OBI internal state available for external access through an HTTP based API. This is the reason which led us to include in OBI a service we called API which contains a web server exposing endpoints for reading OBI's state. In particular the endpoints we provided are:

- *login* to perform user's authentication. In fact, only authenticated users can access the other endpoints
- *clusters* and *cluster* respectively used for obtaining a list of clusters or a specific cluster's details
- *jobs* and *job* respectively used for obtaining a list of jobs or a specific job's details
- *user* to obtain a user's email address, given its OBI's identifier

The API component is the only one whose communication is not implemented through RPC. In fact, as we already mentioned, communication to the API is handled by an HTTP server, receiving and sending JSON[4] messages.

The reason behind the choice of abandoning RPC frameworks in favor of an HTTP server is the fact that we though that the API we developed could have been particularly useful for developing a web interface, showing in real-time what OBI is doing.

In fact, the API component was later extended with more endpoints serving a one-page web application exploiting the API's endpoints for collecting and displaying OBI's internal state. The interface we developed was very minimal and only had one page displaying the list of OBI's clusters where each element, if clicked, results into an UI component showing the jobs and their information within the cluster. All those information can be seen by the user only after he has authenticated with valid credentials.

The web interface we developed is shown in Figure 4.4. We used Node.js[5] to create our web server and React.js[6] to build the web interface.

4.4 Client

Since OBI is deployed as a sort of server listening for job submission requests, we had the requirement of designing a client for it. Our client is a simple CLI (Command Line Interface) which provides the user with all the facilities to submit a job to be executed in a cloud computing environment.

Submitting a job is as easy as giving the client the path to the job's script, its argument and the desired priority and then OBI will take care of any other operational aspect.

[4] https://www.json.org/, accessed on $22^{nd} October 2018$

[5] https://nodejs.org/it/, accessed on 6^{th} December 2018

[6] https://reactjs.org/, accessed on 6^{th} December 2018

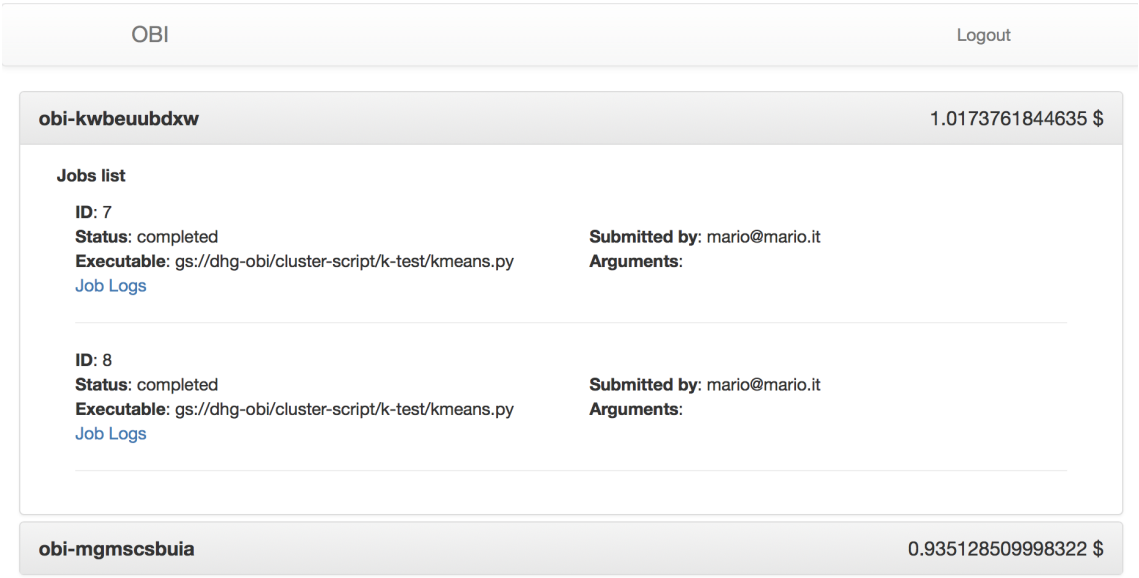

Figure 4.4. OBI Web Interface

Along with the submission of a job, OBI's client also allows a system administrator to create a new deployment of OBI according to his needs. At the moment of the writing of this thesis, OBI's client for administrators only supports Kubernetes[7] as a deployment platform. However, the client was designed to be extensible and in future, it will be possible to create additional deployment possibilities.

[7] https://kubernetes.io/, accessed on 22nd October 2018

Chapter 5

Predictive Component

In the current chapter we will describe the component in which OBI's smart decisions are taken. In particular we will describe the Machine Learning models we built both for predicting the job duration and for implementing our autoscaling feature. In both case, we will also give explanations regarding the datasets we used and how they were generated.

In both cases, we will also provide a formal evaluation of our models. Instead, their results in real-world scenarios will be evaluated in Chapter 6.

The predictive models described below are all accessible to other OBI components through a gRPC interface. Indeed, the predictive component implements a web server exposing the inter-module communication interface required for other components to generate predictions on demand.

Since the predictive component is state-less, in a general OBI deployment, it is replicated and it is accessed through a load balancer in order to guarantee a fair incoming traffic toward each of the replicas. This is done both for improving efficiency in case a lot of jobs are submitted at once (thus generating a lot of prediction requests all together) and for failure resistance reasons.

In our deployment we choose to use two predictive components replicas but the amount of replicas in configurable at the moment of the creation of the specific OBI deployment.

5.1 Job Duration Prediction Models

As we already explained in Section 3.2.2, OBI jobs may be of one out of three different types and, according to the knowledge we had about the different types, we will provide job duration prediction analysis only for the CSV-Load jobs, as for the other types it is not a required feature. Moreover, we could only create models for the CSV-Load jobs because the code base for the other jobs we run in our ETL

pipeline is still a work in progress and, until it reaches a stable point, it does not make sense to perform on them the same analysis we describe in this chapter.

Therefore, in the following chapters, by jobs we will mean CSV-Load jobs, focusing our analysis solely on them and ignoring the other types.

5.1.1 Dataset

As we have already mentioned, an already available dataset did not already exist. This reason, along with the fact that job duration prediction problems are tightly coupled with the specific experimental environment, led us to the creation of our own dataset.

The first problem while creating a dataset, is finding a source for mining the required information. The only possible option from which to obtain details about the execution of our jobs, were their output logs, which are structured as follows:

```
18/05/09 19:13:49 INFO ...
18/05/09 19:13:52 INFO ...
...
18/05/09 19:26:09 INFO ...
18/05/09 19:28:02 INFO ...
```

Given the above structure of the jobs output logs, it was straightforward to obtain their duration because each log line is tagged with its date and time (in the format year/month/day hour:minutes:seconds). In fact, the first line in the log file was assumed to be the starting point of the job while the last was assumed to represent the last output right before the conclusion of the job. This is a reasonable assumption because most of the lines in the logs are written by YARN/Spark itself and we noticed that the first and the last lines always corresponded to details regarding the starting up and the termination of the jobs.

However, the job duration itself was not enough for our problem. Indeed, the duration is supposed to be our target variable (the value we want to predict) and we can not think of building a model relying only on it.

As additional variable, we tried to obtain two main details about the jobs: the cluster status when they were submitted and the input files on which each job operated.

The cluster status was obtained as a set of metrics, describing the usage of the YARN clusters in the specific moment when each job was submitted. The required historical metrics were obtained through Stackdriver[1], the Google Cloud tool which

[1] https://cloud.google.com/stackdriver/, accessed on 29^{th} October 2018

allows users to track the resources usage for all the available services, and are better describe below.

The input files features, instead, were obtained by looking at the workload type of our jobs and at the knowledge we have of them. Indeed, we already knew which kind of files each job receives as input, so we just compute the number of input files and their total size for each of the jobs in the dataset.

Finally, we obtained a dataset of 3231 points, each one corresponding to a job, with the following information:

- **Cluster Identifier**: unique Google Cloud Dataproc related identifier of the cluster in which the job has been executed
- **Job Type**: in our case this field can only contain the value "CSV-Load". We left it in order to make our dataset easily extensible in future
- **Backend**: used to identify the job's workload type
- **Job Identifier**: unique Google Cloud Dataproc related identifier of the job
- **Job Subtype**: either CSV-Load "find_changes", "update" or "recreate_file"
- **Start Time**: date and time indicated in the first log line
- **End Time**: date and time indicated in the last log line
- **Job Failed**: boolean feature indicating whether a job failed or not, set to true if the last log line was related to any fatal exception
- **Number of Input Files**: number of files given to the job as input
- **Input Files Size**: total size in gigabytes of the job's input files
- **Cluster Metrics**: set of metrics describing the cluster's status when a job was submitted

The cluster metrics we used to describe the cluster status while running our jobs were the followings:

- Number of jobs running in the clusters
- Available memory in the cluster
- Percentage of cluster's allocated memory
- Available virtual cores in the cluster
- Number of pending virtual cores

5.1.2 Analysis and Modelling

Before doing any modelling related to job duration prediction, we wondered if we could exploit the information we had about failed jobs to predict failures. However, as shown in Figure 5.1, if we would classify the jobs termination status into succeeded or failed, we would have to deal with a highly imbalanced dataset, where the failed jobs are only 36, representing the 1.1% of the total. Moreover, we already know that our jobs are production ready jobs, which means that before being executed in the ETL pipeline they are extensively tested and it is very unluckily that some of them will fail. Moreover, we believe that the data points corresponding to failed jobs can be considered outliers and we dropped them while doing duration prediction modelling.

Therefore, the idea of classifying job termination status was set aside very soon.

Figure 5.1. Termination status for the jobs in the time duration dataset

Concerning the job duration prediction problem, instead, we decided to build three different models, one per job sub-type as mentioned in Section 3.2.2. In this way we were able to obtain better performances rather than modelling all the job types into one single model. This choice was also justified by the fact that the three different job sub-types correspond to three actual different workload types.

Moreover, we were able to use multiple models because we are able to determine which kind of job we are trying to execute by simply looking at its code before submitting it into a cluster. Indeed, as already mentioned, the jobs we submit are production ready jobs and rarely change.

Since we followed the approach of splitting our model into three separate ones, we will proceed explaining our models one by one.

CSV-Load update

Considering only the CSV-Load update jobs, we have 1257 data points in our dataset, where the oldest job was executed on September 4^{th} 2018. The job duration in this dataset is distributed as shown in Figure 5.2.

The duration distribution of our jobs clearly shows that there must be at least three groups of jobs which presents similar behaviour. This intuition is confirmed by Figure 5.3, which shows the average duration of CSV-Load update jobs for each different backend. Indeed, there are three backends for which jobs have a considerably longer duration with respect to others. Those three backends (b06, b11 and b03) are related to the jobs characterizing the right-most bimodal distribution from Figure 5.3. Instead, there is a third backend (b02) whose jobs are slightly faster than the previous two and which characterize the central peak in Figure 5.3. The left-most sub-distribution, instead, is related to all other backends, which have a similar and short average duration.

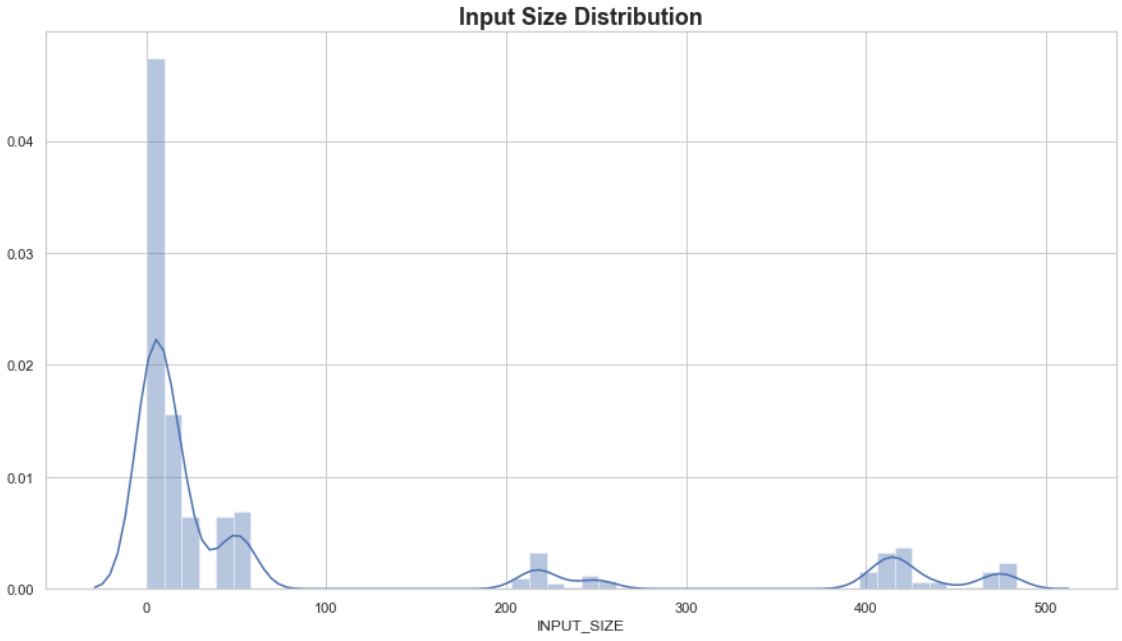

Figure 5.2. Job duration distribution for CSV-Load update jobs

Given what is shown in Figure 5.3, one first attempt at job prediction we made was to build two separate regression models: one for the jobs which we are short and another one for the longer jobs. In this way we split our dataset in two parts,

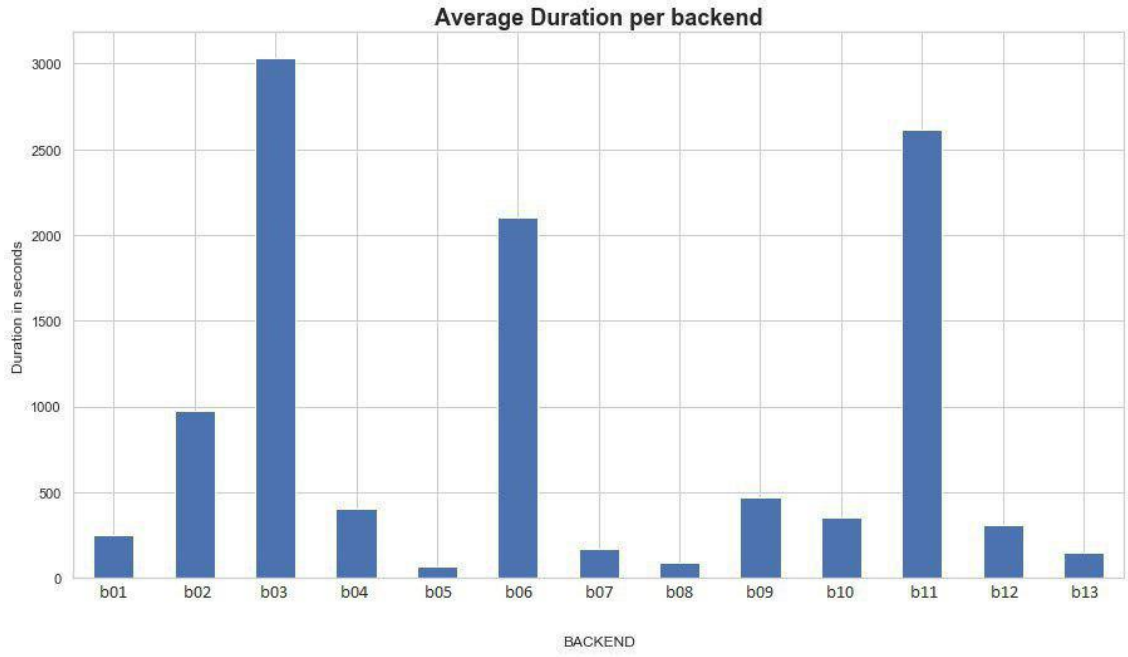

Figure 5.3. Average job duration for each backend for CSV-Load update jobs

where, one part contained all the datapoints for the long backends (b11, b06 and b03) and the other part contained the points related to all the other backends.

If we focus on the average duration of the jobs we consider to be short, we can notice an anomaly. Indeed, as shown in Figure 5.3, the jobs for the backend b02, last on average more than two times the average duration of the others. If we plot the distribution of the duration for small jobs, distinguishing them based on whether or not they are related to the b02 backend, we obtain what is shown in Figure 5.4 where, apart for an outlier value, there is a clear distinction between the duration of the jobs which operates on b02 and the others. Therefore we decided to add to our points a boolean feature which was set to true if the job for a b02 one, to false otherwise.

As also stated in Chapter 2, our major goal is to predict the duration of jobs which executes similar operations but on different inputs. In Figure 5.5 it is shown how the duration varies against input size variations. As we can see, apart for some outlier values, when the input size increases, the jobs tends to last more.

After having removed the outliers found above, we had a dataset of 751, where each point had the two features we selected: a boolean feature stating whether or not a job is related to b02 backend or not and the total size of its input files in GB. Based on that, we built several models which performances were evaluated in terms

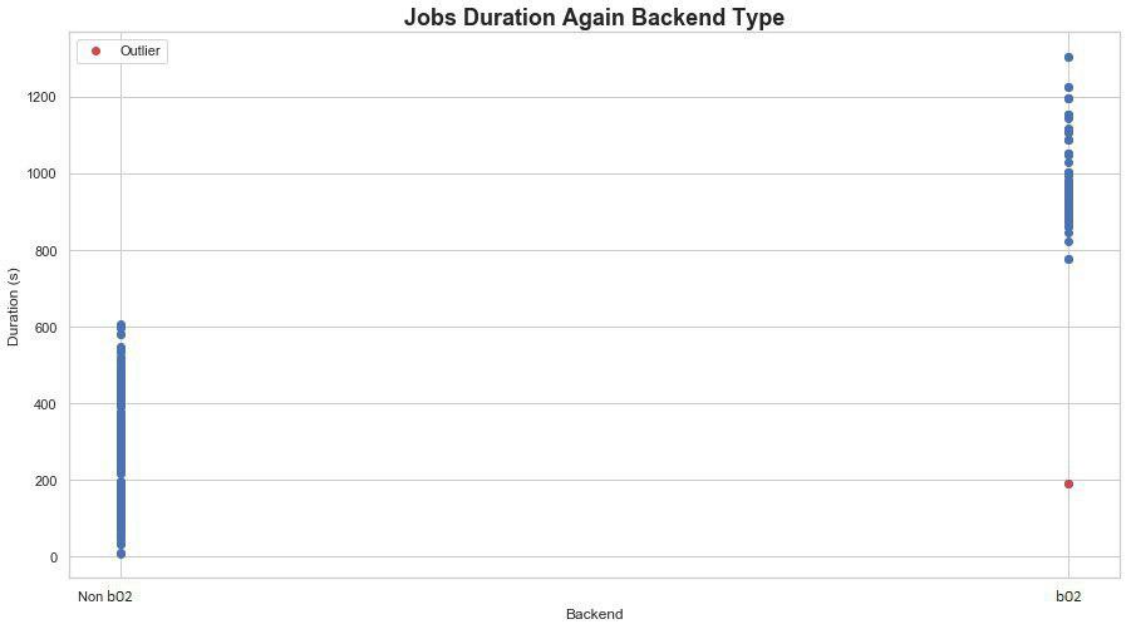

Figure 5.4. Job duration for small CSV-Load update jobs for b02 backend and for others

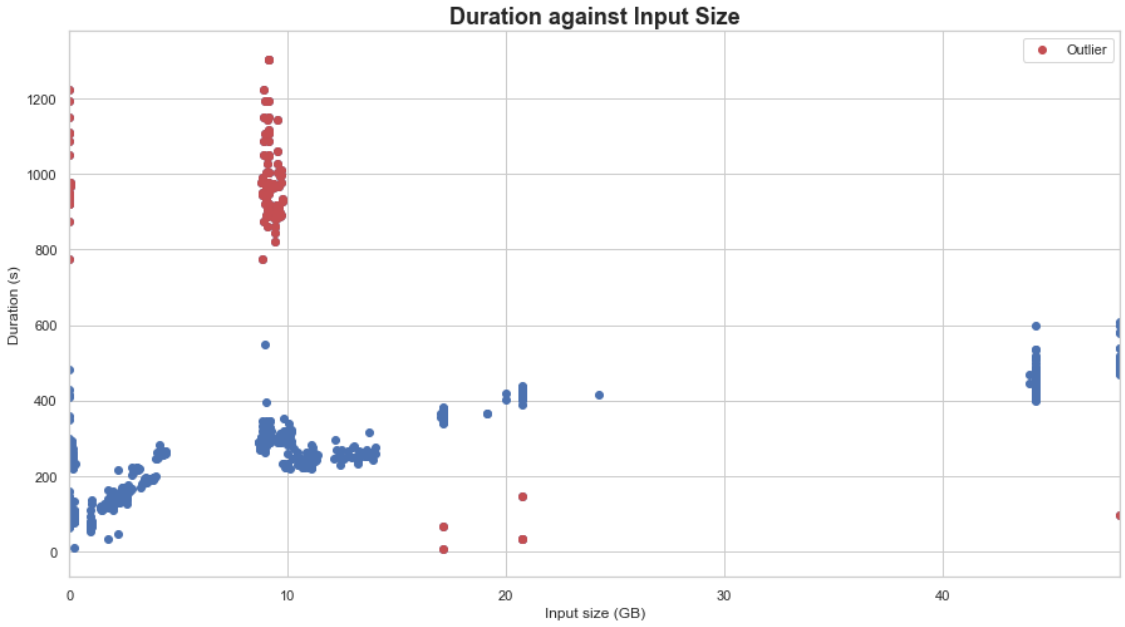

Figure 5.5. Job duration against input size for small CSV-Load update jobs

of two indexes: root mean squared error (RMSE), and R^2 score[2]. Before applying

[2] https://en.wikipedia.org/wiki/Coefficient_of_determination, accessed on 31st October 2018

any regression algorithm to our data, the target variable was transformed according to the BoxCox transformation [21], in order to provide it with a shape which can be easier to be understood by our model.

Model	**RMSE in seconds**	R^2 **score**
Linear Regression	12.49	0.63
Quadratic Regression L2	8.81	0.77
Bayesian Linear Regression L2	8.33	0.78
Gaussian Process	4.95	0.63
XGBoost	2.79	0.92

Table 5.1. Regression models results on CSV-Load update short jobs

The results shown in Table 5.1 (where L2 means that L2 regularization was applied to the respective model) were obtained after applying 10-fold cross-validation[3] to our dataset and after applying the techniques mentioned above to our dataset. Of course, since we transformed the target variable using the BoxCox transformation, the RMSE in seconds was obtained by applying the inverse transformation to the resulting BoxCox RMSE. At the end, the best results were obtained by using an XGBoost [11] algorithm whose hyper parameters were optimized using a Gaussian Optimization Process [25], a optimization technique which relies on fitting a Gaussian Process [28] on the function representing the model's performances.

Similarly to what we have done for CSV-Load update short jobs, we applied the same processing techniques also to the jobs we defined as long, composed of 394 data points. The only difference is that, for long jobs, we used a boolean feature which is true if the job was related to b06 backend along with the total input size in GB. We added this additional feature because we noticed that it was improving performances. Indeed, we tried adding the same feature for all the three available backends in this sub-dataset but the one we added was the only added improving our model's results.

The results for the model trained on long-lasting jobs are shown in Table 5.2, and they were obtained by performing 10-fold cross validation on our dataset.

In this case we had very poor performances basically with all our models. We believed that by cleaning our dataset a little further we could have been able to improve our performances, however, we would have had to reduce an already very small dataset of only 394 points. Since we also believed that the major cause of the poor performances was the shortage of data points, causing the model not to be able to properly generalize, we decided to change approach.

[3] https://en.wikipedia.org/wiki/Cross-validation_(statistics), accessed on $20^{th} November 2018$

Model	**RMSE**	R^2 **score**
Linear Regression	140825.70	0.06
Quadratic Regression with L2 Regularization	139167.57	0.08
Bayesian Linear Regression with L2 Regularization	140837.20	0.06
Gaussian Process	140825.70	0.06
XGBoost	143311.93	0.04

Table 5.2. Regression models results on CSV-Load update long jobs

In fact, in order to overcome the limitations caused by the lack of data points while splitting our dataset into two separate parts, we tried to model all the data points into one single model. In this case, for modelling the jobs backend, we performed a one-hot-encoding transformation on the categorical backend feature. It means that, for each backend, we added a new boolean feature, which is 1 is the corresponding job operates on the respective backend, 0 otherwise.

As an additional processing step, in the case of the unified model we applied the following outliers pruning:

- we removed all the b06 jobs which lasted more than 2500 seconds
- we removed all the b11 jobs which lasted more than 1800 seconds
- we removed all the b03 jobs which lasted more than 2400 seconds

We could apply this pruning because, by looking at real-world executions, we saw that the jobs operating on the three mentioned backend could never exceed the proposed thresholds. In the end, our dataset decreased from 1257 to 1137 data points.

Since from previous experiments we already knew that, for our data, the best model was the XGBoost one, we built it by optimizing its hyperparameters using the usual Gaussian Optimization Process and the results we obtain, which showed to be the best 10-fold cross validation performances, are shown in Table 5.3 where our model's performances are compared to those of a Naïve model always predicting the average duration value.

Model	**RMSE in seconds**	R^2 **score**
Naïve Model	2.00	0
XGBoost	1.12	0.95

Table 5.3. Regression models results on CSV-Load update jobs

In fact, apart from the very low RMSE value, the 0.95 value for the R^2 score is interesting because it let us know that our model, despite treating all backends at once, is capable of catching the 95% of the variations in the dataset's target variable.

CSV-Load find_changes

CSV-Load find_changes jobs behave similarly to update ones and the operations they perform are mostly of the same type. This is also visible from Figure 5.6 where, similarly to what happened for update jobs, we have three backends whose jobs last considerably longer with respect to the others while we have the b02 backend jobs which, on average, last twice the same time as the other small jobs.

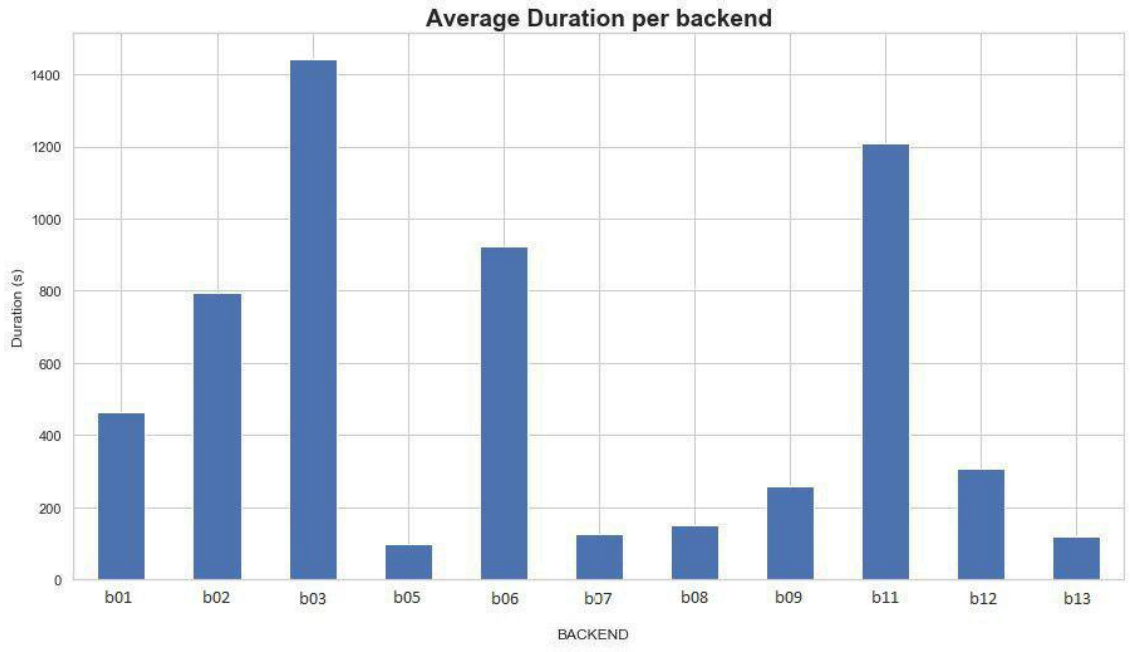

Figure 5.6. Average job duration for each backend for CSV-Load find_changes jobs

Indeed, also with find_changes jobs we can see how the jobs operating on the backends b06, b11 and b03 takes longer to execute if compared to the others, even if the duration distribution is more similar to a uni-modal Gaussian as shown in Figure 5.7 with respect to the duration distribution of the CSV-Load update jobs.

After having learnt the lesson that splitting the modelling of our jobs into two parts (namely short and long jobs) only increases the module's complexity without bringing any benefit, in this case we proceeded directly with building one single model.

In order to remove outliers from our dataset of 1037 jobs, we exploited our prior knowledge on them and the fact that we could observe the execution of CSV-Load jobs in production environment and we applied the following rules:

- we removed all b06 and b11 jobs lasting less than 900 seconds

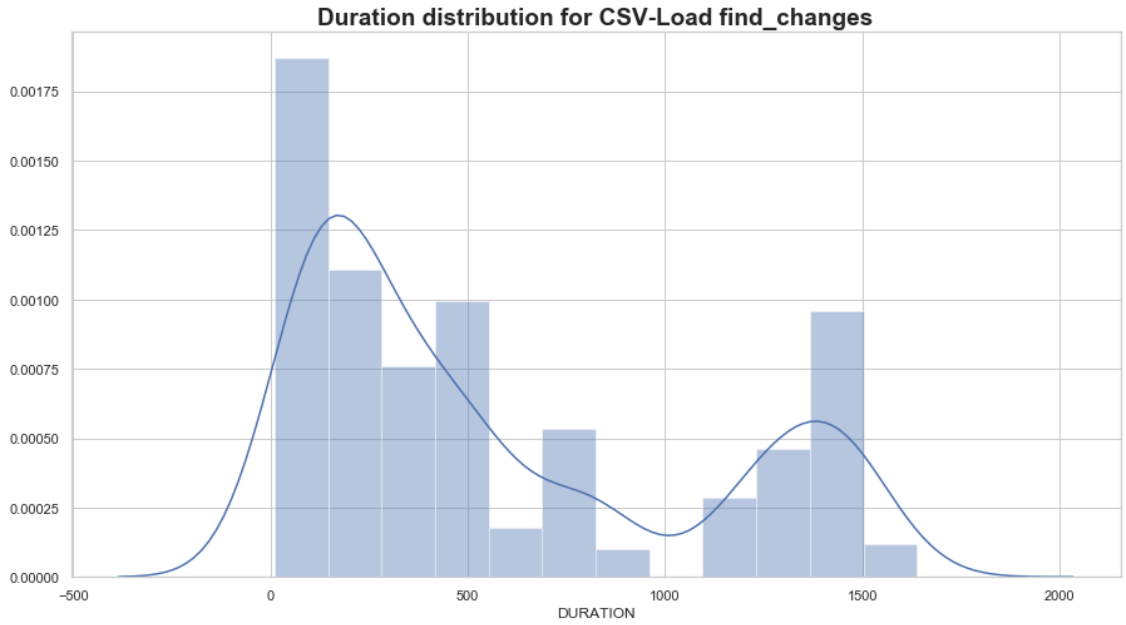

Figure 5.7. Job duration distribution for CSV-Load find_changes jobs

- we removed all b03 jobs lasting less than 1200 seconds

At the end we were left with 942 total data points.

Moreover, as we have already done for the update jobs, we have transformed the categorical backend feature into a one-hot-encoded set of boolean variable and we applied BoxCox transformation to smooth the distribution of the duration variable as shown in Figure 5.8.

Model	**RMSE in seconds**	R^2 **score**
Gaussian Process	1.22	0.91
XGBoost	1.11	0.96

Table 5.4. Regression models results on CSV-Load update long jobs

In the end, after training several models, we obtained the results shown in Table 5.4. Those results were obtained by performing a 10-fold cross validation on our dataset. Again, the best results were obtained with an XGBoost regressor model.

CSV-Load recreate_file

CSV-Load recreate_file jobs present different characteristics with respect to the previous two categories. First of all, they do not have any specific input size because they never read files whose size is comparable to those of the files given as input to the

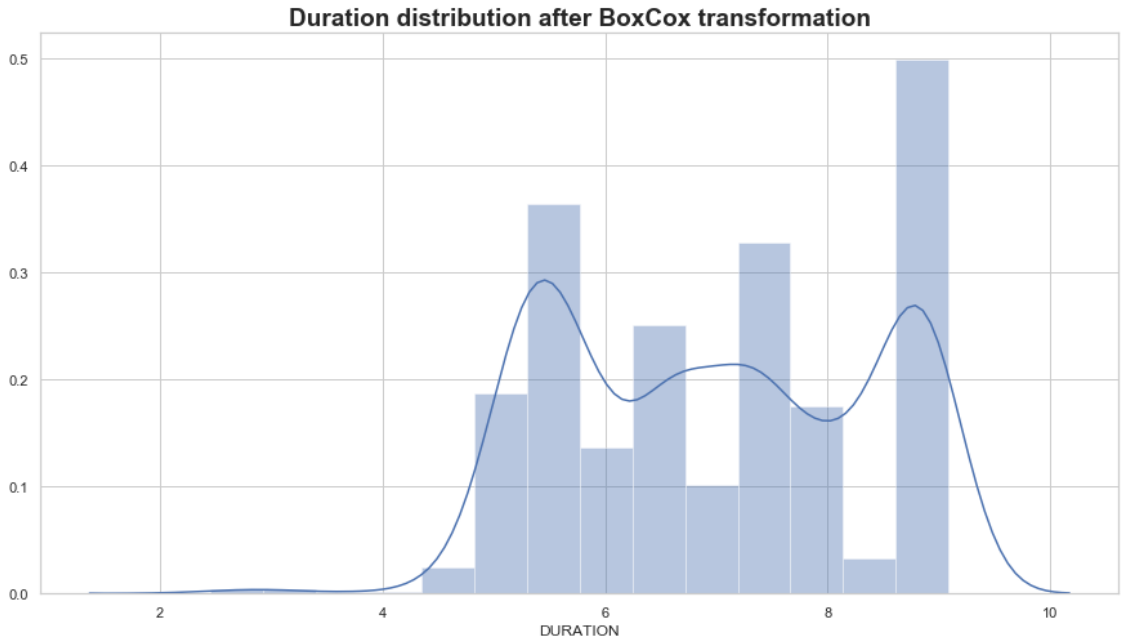

Figure 5.8. Job duration distribution for CSV-Load find_changes jobs after BoxCox transformation

other two job types. Moreover, they are usually much faster as shown in Figure 5.9 because they perform lighter operations. Because of this characteristics, we could not apply the same exact analytical approaches we had applied on the other two job types.

Duration distribution for recreate_file jobs looks Gaussian in its lower interval but it has some anomalies as shown in Figure 5.10. This is caused by the fact that there are two points which have high duration values, respectively 43 and 71. For this reason, BoxCox transformation was applied on the duration variable, which resulted in the distribution shown in Figure 5.11.

In the case of recreate_file jobs, building a naïve model which predicted always the average duration value would have been probably enough for our use case. In fact, we do not need our model to be extremely precise and, while in the previous cases we had to apply particular data processing techniques for making our model able to properly discriminate jobs executed on different backends, in this case, our processing pipeline can be as simple as we can. This is due to the fact that, since recreate_file jobs are very fast, also an error of e.g.the maximum value of 71 seconds it tolerable for our application. Indeed, we will you job duration predictions in OBI's scheduler policies which are generally tuned to contain at least groups of jobs of at least one or two hours. Therefore, being wrong of about 1 minute is not a major drawback.

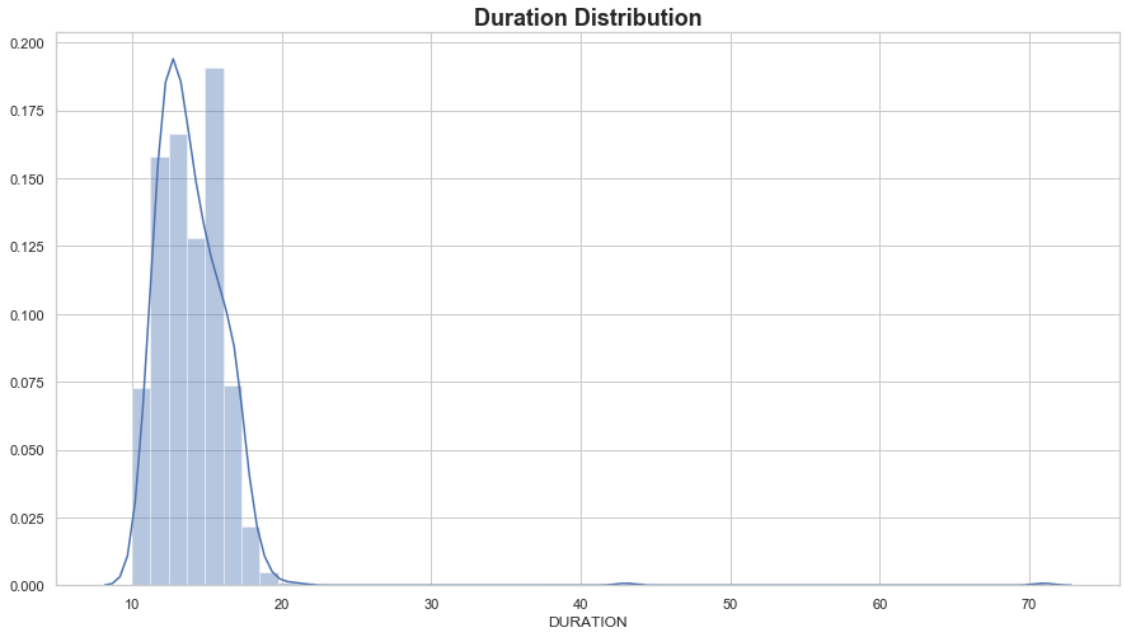

Figure 5.9. Average job duration for each backend for CSV-Load recreate_file jobs

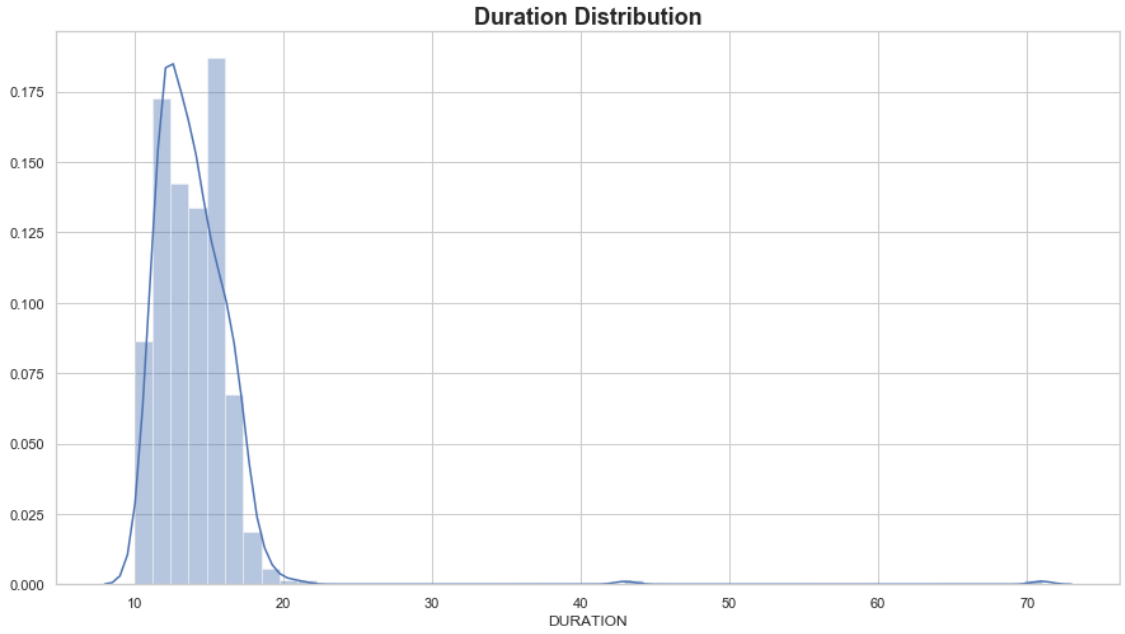

Figure 5.10. Job duration distribution for CSV-Load recreate_file

Since in the previous cases we noticed that an XGBoost model provided good performances, we have built one also for recreate_file jobs and compared it with the

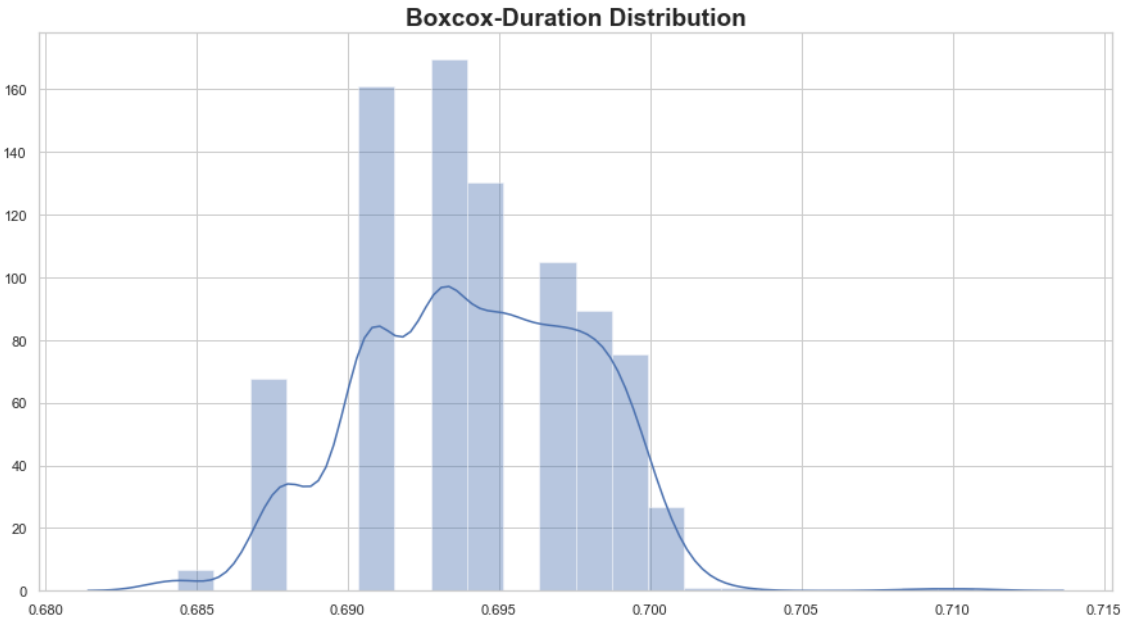

Figure 5.11. Job duration distribution for CSV-Load recreate_file after BoxCox transformation

naïve model. In this case, since we did not have any input size details available, we used as input features to our model the available memory and the available CPUs at the moment of the job submission, as well as a one-hot-encoded backend variable.

Model	**RMSE**	R^2 **score**
Naïve model	1.00	0
XGBoost	1.00	0.44

Table 5.5. Regression models results on CSV-Load recreate_file

The results we obtained after applying 10-fold cross validation are shown in Table 5.5. As we can see, our model has a R^2 score of 0.44, which means that it can address for the 44% of the variance in our target variable. Despite this result is not impressive, as we already said we could have accepted the results of the naïve classifier for our goal but we still created a model since we already had the data ready to be analysed.

Despite the model we built does not lead to impressive performances gains over the Naïve model, we still decided to use it for the reasons already highlighted above.

5.2 Statistical Autoscaling Model

In this section we will explain the process which led us to build our machine learning based autoscaler for OBI. We will firstly describe how we collected our dataset and then we will move our focus to its in-depth analysis.

5.2.1 Dataset

Similarly to what we have done for the job duration prediction problem, we built our own dataset also for the machine learning based autoscaler. In this case, however, we used a different approach.

In fact, while for the job duration dataset we analysed historical data, for the autoscaler dataset we have executed several experiments and we have collected the data in real-time.

In particular, we have executed our jobs in clusters starting with a minimal configuration. Each cluster was then automatically autoscaled according to the following naïve policies:

- timeout based autoscaler which simply scales up by one node every time a timeout expires until the job finishes
- timeout based autoscaler similar to the above one but the scaling factor was chosen among a random integer value in range $[-5, +5]$
- threshold based autoscaler which monitors the cluster resources utilization and dynamically decides the amount of nodes to scale
- threshold based autoscaler which monitors the cluster resources utilization and, if a scaling action in a certain direction (up or down) is needed, it grows the scaling factor exponentially

In order to try to maintain our dataset as balanced as possible with respect to the different autoscaling policies, we have executed the complete set of our jobs several times, each time using a different policy. However, since some autoscalers, e.g. the timeout based ones, scales with higher frequency, we have kept collecting points also while running OBI for experiments not directly concerning this thesis work in which the threshold based autoscalers were used.

Given the described information, our goal was to obtain a dataset such that each point could represent a scaling action, including the benefits it generated. Therefore, we needed to define a metric for the assessment of the cluster utilization. For this reason, we used the same metric described in [19], where the utilization of a cluster's resources, is evaluated in terms of the growth of the pending containers and the speed

at which containers are released over a certain time frame. Basically, the formula of our performance metrics is the following:

$$\pi = T - P \tag{5.1}$$

Where T is the so-called throughput or the speed at which YARN containers are released (containers completion speeds) and P is the growth of the pending containers within a time window. This same metric is used in the threshold based autoscaler used while collecting data for our dataset.

Based on the results of the above difference we may have three situations:

- $\pi > 0$: it means that the throughput is higher than the pending containers growth, meaning that we may need to downscale because we have more nodes than those we need to accommodate all the pending requests
- $\pi < 0$: it means that the pending containers growth is higher than the throughput, meaning that we may need to upscale because new container requests are made at a faster rate with respect to the rate of completion of the already allocated ones
- $\pi = 0$: it means that the cluster is being perfectly utilized in the considered time frame

Of course, the goal of our autoscaler is to reach a situation in which $\pi = 0$.

For each scaling action, our dataset contains the described performance metric both before and after the scaling action. Moreover, each data point contains the initial cluster configuration expressed in terms of usage metrics and in terms of available nodes as well as the delta of added/removed nodes, also known as the scaling factor. Additionally, to better understand what were the effects of each scaling action, each data point also contains usage metrics after the scaling action is completed. Indeed, each point representing a scaling action in our dataset contains the following information:

- Number of nodes in the cluster before the scaling operation
- Scaling factor which can be either positive or negative, respectively indicating upscaling and downscaling
- Performance indicator before the scaling operation
- Performance indicator after the scaling operation
- Cluster metrics before the scaling operation

- Cluster metrics after the scaling operation

The cluster metrics were collected while sending the scaling actions datapoints through the OBI's heartbeat mechanism. In particular, the metrics "before scaling" are collected in the same moment in which a scaling action is completed while the metrics "after scaling" are collected a time window after the first one which, in our case, meant one minute later.

The the collected metrics (both before and after scaling) are the following ones:

- Available memory in megabytes
- Allocated memory in megabytes
- Pending memory in megabytes
- Available virtual cores
- Allocated virtual cores
- Pending virtual cores

In the end we had a dataset consisting of 6078 data points.

5.2.2 Analysis and Modeling

The machine learning based autoscaler final goal is to predict the optimal scaling factor which would bring our cluster performances to the desired level. When the autoscaler is triggered, however, the only information we have at our disposal is the status of the cluster in that specific moment (which is represented by the metrics before scaling are in our dataset) and the performance measure. We will deal with this problem later in our discussion, for now we will just focus on how to predict the optimal scaling factor.

In order for our autoscaler to be able to find the optimal scaling factor whenever scaling is needed, we need it to learn the pattern inside our dataset which relates the current number of nodes in the cluster and how the scaling affected the performances metrics. To do so, the input to our model will be composed of the following parameters:

- Number of nodes in the cluster before scaling
- Performance metric before scaling
- Performance metric after scaling

- Usage metrics before scaling
- Usage metrics after scaling

The output variable, instead, will be obviously the scaling factor.

In this way we are assuming that our model will be able to learn the pattern which relates the current cluster configuration to a variation of it, in terms of performances and metrics. This variation, indeed, is what is explained by the actual scaling factor.

Given our problem statement, what we had to find was a proper regression model, capable of generating the desirable value for our scaling factor. However, before building the model, we explored our data, followed by a data preprocessing step to maximize our final model's performances.

The first analysis we did, was to compute some statistics on the scaling factor variable from our dataset, as shown in Table 5.6.

mean	$-7.066370 \cdot 10^5$
standard deviation	$3.895191 \cdot 10^7$
minimum	$-2.147484 \cdot 10^9$
25% percentile	-1.0
50% percentile	1.0
75% percentile	4.0
max	377.0

Table 5.6. Scaling Factor Statistics

The values from Table 5.6 clearly shows that our dataset has outliers. Indeed, it is very unluckily that we will ever need to remove $2.147484 \cdot 10^9$ nor to add 377 nodes, especially if we consider that the computing nodes which are used by OBI in our environment have the specifications shown in Table 5.7.

Memory	15GB
Cores	4
Disk Space	500GB

Table 5.7. Amount of resources per computing node

Given the above mentioned node configuration, we assumed that scaling up or down of 5 nodes should be sufficient for any of our possible use cases. Indeed, in this way we would add (or remove) a total of 75GB of memory, 20 cores and 2.5 TB of disk space to our clusters, which seems to be a reasonable amount of resources for any of our possible operations. Therefore, we decided to prune our dataset to those data points having a scaling factor in range $[-5, 5]$, resulting in the scaling factor

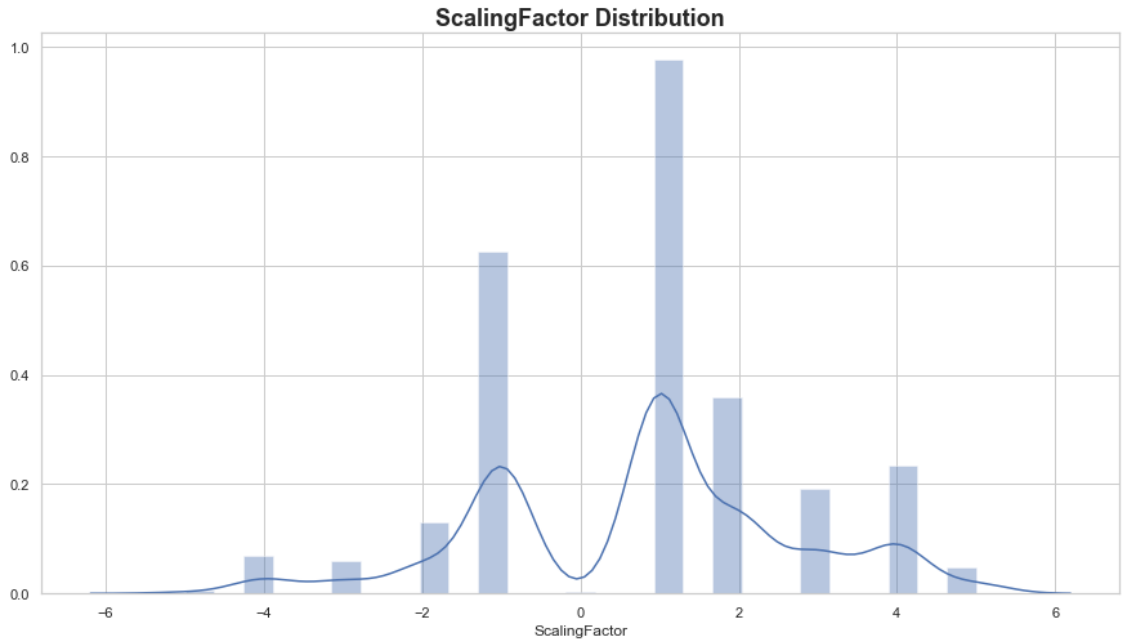

Figure 5.12. Scaling Factor distribution after pruning

feature to be distributed according to the distribution shown in Figure 5.12. After our pruning operation, we reduced our dataset from 6078 to 4910 points.

Inspired by MLScale [33], we tried to build a feed-forward neural network at this point having the following architecture:

- Input layer with 120 neurons, ReLU [7] activation function and 50% dropout probability
- Hidden layer with 100 neurons, ReLU [7] activation function and 50% dropout probability
- Output layer with 1 neuron and Sigmoid [8] activation function

The dropout probability was simply added as a counter measure to potential layers overfitting [30].

As we already mentioned above, our goal was to learn the patter between resources utilization, current cluster configuration and variation in terms of performance and the scaling factor. To do so, the input features of each data point to our neural network were the following values:

- Number of nodes in the cluster before scaling
- Performance metric before scaling

- Performance metric after scaling
- Usage metrics before scaling
- Usage metrics after scaling

Between the available metrics we had, we only choose those related to the available and pending memory and cores. The reason behind this choice will be made clearer later.

Before training the real model, however, we applied two preprocessing steps.

Firstly, we scaled the input variables. It means that, for each feature we computed both its mean μ and variance σ^2 and we applied the following transformation to each data value:

$$x = \frac{x - \mu}{\sigma^2} \tag{5.2}$$

In this way our features had 0 mean and unitary variance. The main reason for doing this, is that this process made all inputs comparable meaning that it reduced to a same scale both big and small values, e.g. both memory size and number of cores. However, there are a variety of other practical reasons for employing such a preprocessing techniques. By standardizing the inputs we can make training faster and reduce the chances of getting stuck in local optima, improving overall performances of our training process [29].

Moreover, along with normalizing input data, we applied data preprocessing also to the target variable. In fact, since the output layer of our neural network architecture has a sigmoid activation function, whose output range falls in the interval $[0, 1]$, we decided to normalize the scaling factor variable, in order for it to fit in the same interval as the output of the sigmoid function. Of course, this step requires that we apply the reverse transformation to the output of our network if we want to obtain the real scaling factors.

After the definition of the architecture and the application of the described preprocessing approaches, our neural network was trained using Adam optimizer, an algorithm for first-order gradient-based optimization of stochastic objective functions which provides state of the art performances while training neural networks with gradient descent [16].

The loss of our neural network was defined as the mean squared error (MSE) of our samples. While training, 20% of the original dataset was taken as validation set in order to provide better evaluation of our model's performances. Therefore we had 2514 data points for our training set and 628 data points for our validation set. The loss optimization process progresses can be seen in Figure 5.13.

As we can see, our network did not seem to overfit as both the validation and the training losses keeps reducing across each iteration until they reach a stable value.

In the end, following this approach we obtained a root mean squared error (RMSE) of 0.892433560011 nodes, which means that, on average, we mispredict the scaling factor by less than one node.

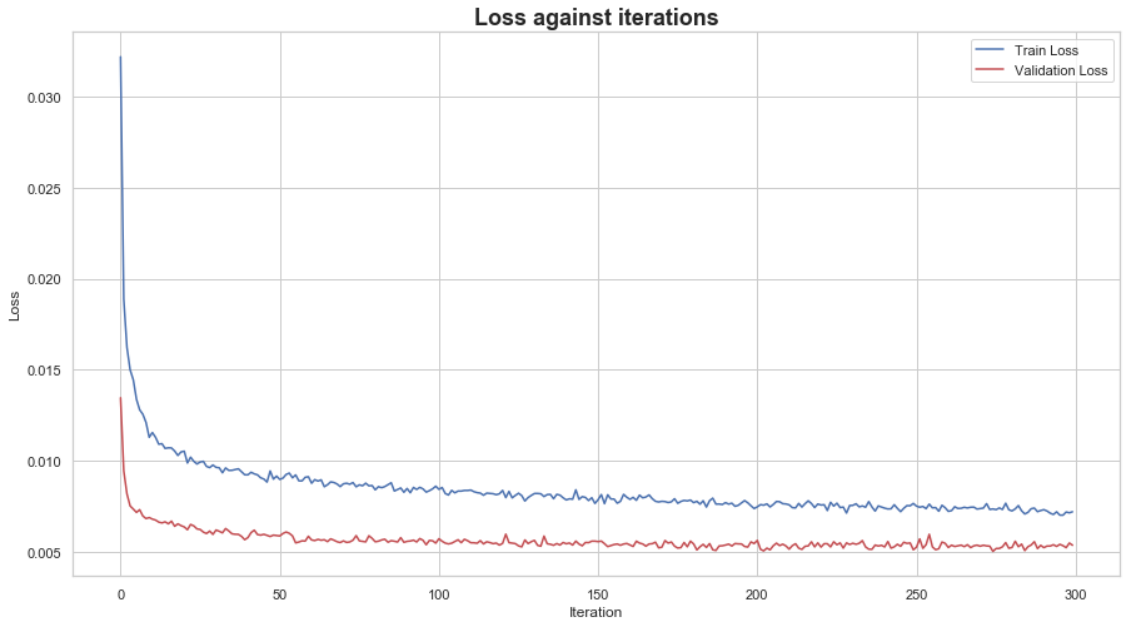

Figure 5.13. MSE loss for the autoscaler neural network

Despite the promising results, our model has one big limitation when being applied to real-world scenarios. In fact, as we explained above, in order to predict the scaling factor our neural network needs to know, among the other things, the performance measurement and the cluster metrics after the scaling operation has been applied.

The performance metric, however, is not a relevant problem. Indeed, since we explained that we want it to be 0, the performance measure after scaling can simply be put to 0 in all the real-world predictive cases.

Having the cluster metrics after the scaling operation, however, is a more challenging problem. Indeed, understanding the cluster status after scaling, requires the model also to be able to predict the cluster metrics after scaling, even before computing the actual scaling factor.

In order to predict the cluster metrics after scaling, we fit a linear regression for each metric, basing our approach on the following equation taken from [33]:

$$m\prime = c_0 + c_1 m \frac{w}{w+k} + c_2 m \frac{k}{w+k} \tag{5.3}$$

with:

- m metric value before scaling
- $m\prime$ metric value after scaling
- w number of nodes in the cluster before scaling
- k scaling factor
- c_i parameters estimated through the fitted linear regression on the specific metric

The linear regression coefficients were estimated, at training time, using the scaling factor from each data point. However, again this is an information we miss in real world cases.

To remedy to this problem, while deciding the scaling factor in real world experiments, our autoscaler sample 10 values from the scaling factor distribution shown in Figure 5.12, generates the same amount of after-scaling metrics predictions and then averages them, in order to give them as input to the neural network.

Predicting the metrics after scale somehow complicates our prediction process. Indeed, we asked ourselves if it was the case to attempt at predicting the optimal scaling factor only using the values for the metrics before scaling operation. We built the same exact neural network, applying the same preprocessing steps on the dataset containing only the metrics value before scaling and the RMSE we obtained was of 1.2806049779 nodes. Since we wanted to keep our RMSE below 1 and that computing 10 predictions using a linear regression model is computationally irrelevant, we decided to keep the model requiring the prediction of the metrics after scaling.

From the predictive component side, the autoscaler service can be triggered from OBI's master by simply sending to it the current cluster status and the current performance metrics. In general, the autoscaler is trigger every time a metrics window is received from the master which, in our case, means that the autoscaler service is requested to generate a scaling factor prediction each minute. However, in order to be able to tune out autoscaler, we added a parameter to its master's policy which we called "scaling trigger". This parameters is used to determine when to invoke the autoscaling service, if the user should not want to trigger it at every metrics window. Basically if we assign the scaling trigger a value of x, the autoscaler service will not be called unless our cluster performance measure is such that $|\pi| > x$.

In Chapter 6 we will show how we used different scaling trigger values to properly tune our autoscaler.

In order to be able to tune our autoscaler to be more or less conservative, we added a parameter to its autoscaling policy. Basically, the machine learning based autoscaler can be request based

From an implementation point of view, our neural network was implemented using Keras [13], while for all other operations, including linear regressions and preprocessing techniques, we relied on SciKit-Learn [26]

Chapter 6

Results Evaluation

In this chapter we will provide a formal evaluation of the results we obtained when we deployed OBI in our experimental environment described in Chapter 3. Firstly, we will provide an overview of the autoscaler behaviour on a randomly selected job of our ETL pipeline and then we will describe what were the performances variations we obtained while running the whole ETL pipeline through the system we designed.

Before discussing the autoscaler performances we must state one little modification we had to implement in order to make it working with our environment. Since our ETL pipeline jobs required specific dependencies we had to install them in our clusters before the jobs are submitted for execution. In the Google Cloud Platform and Dataproc environment we have to way of doing this:

- Initialization script[1]: Dataproc cluster can be attached with a custom script which is executed right after the cluster resources are allocated and just before any job is scheduled for execution; in this way we could install the necessary dependencies for every cluster when they are created
- Custom Images[2]: each Google Cloud Computing Instance (which is a Virtual Machine in Google's terminology) can be attached with a custom image which is a boot disk image that the user owns and controls access to; we could attach a custom image containing all the required dependencies to all our clusters nodes without having to reinstall everything every time

We noticed that installing all the dependencies every time a cluster is created through an initialization script introduced too much overhead. Indeed, cluster creation time increased by around 5 minutes since each node had to download and

[1] https://cloud.google.com/dataproc/docs/concepts/configuring-clusters/init-actions, accessed on 19^{th} November 2018

[2] https://cloud.google.com/compute/docs/images, accessed on 19^{th} November 2018

install everything which was needed. Thus we decided to use custom images.

The use of customer images, however, introduced another problem. Assuming that we are doing downscaling thus removing a node in a cluster the usual Dataproc behaviour is to gracefully decommission the node, letting the allocated containers in it to finish their tasks and then removing the node. In this way it is not possible for a job to crash due to unexpected behaviour while downscaling. Unfortunately, this behaviour is not supported by Dataproc while using custom images. Therefore, while using custom images an attempt to downscale will drastically kill the tasks running on the target node leading to likely job failures which, in our case, would slow down and reduce the overall performances of our ETL pipeline. Of course, a crash of this kind may have a big impact also on the cost side.

Despite not having the graceful decommission feature, we decided to use custom images anyway, in order to reduce to the minimum time possible the cluster creation overhead. In fact, this overhead would also have affected up-scaling, because newly added nodes still requires dependencies to be installed. However, in order to reduce the probability of having failed jobs while running our pipeline we decided to disable downscaling in our autoscaler. Basically, every time our neural network-based autoscaler decides to downscale, we saturate the scaling factor to 0, thus not doing any scaling action at all. We adopted this solution waiting for the graceful decommission with custom images features to be implemented in Dataproc.

For the reasons highlighted above, the results presented in this chapter do not present any downscaling action. The only downscaling patterns which could be visible from the below plots are due to the fact that we are using preemptible VMs which, as explained in Section 3.2.1, could be temporary removed at any time, with no clearly defined pattern.

It must be said that we did not observe major performance drawbacks in case of preemption. Indeed, it is a different case with respect to the downscaling action mostly because of the fact that, while our autoscaler may decide to remove more than one node, preemption is very unluckily to happen more than once in the same cluster.

Despite the fact that we could have adopted a less strict approach than saturating downscaling actions to "no scaling at all" actions like saturating negative scaling factors to -1, we still decided to disable downscaling because we did not want to have situations in which a small downscaling factor combined with a preemption would kill our jobs. Indeed, we tried to minimize the possibility of accidental job crashes.

In order to test our autoscaler we picked a random job from our ETL pipeline. Specifically we will show results obtained by running a CSV-Load update on the b06 input source in OBI. The average cost and duration of our test job are shown in Table 6.1. Moreover, we will show the results for different autoscaler configurations since, as we describe in Chapter 5, we have a configurable parameter for it. In fact,

Cluster Size	Duration	Cost
20 nodes	4.54 $	49m

Table 6.1. Test job execution statistics without OBI, average on a 10 days period between the 9^{th} and the 19^{th} November 2018

we tried our autoscaler using both highly conservative (high scaling trigger parameter) and less conservative (small scaling trigger parameter) configurations. Those results are shown in Table 6.2 and, each of them, is obtained by averaging the results obtained from 30 different execution of the same job with the same configuration.

Scaling Trigger	Cost	Duration
16	0.75 $	1h 13m
10	0.93 $	1h 2m
8	1.11 $	1h 9m
4	1.30 $	1h 12m
0	1.74 $	1h 16m

Table 6.2. Test job execution results with OBI's autoscaler, averaged on 30 executions

In our most conservative configuration (scaling trigger equals to 16) we observed no scaling at all, resulting our job to be executed with the initial cluster configuration for its whole duration. As a remainder, our clusters start with 2 non-preemptible nodes, each having 4 cores and 15 GB of memory. Therefore the most conservative configuration was certainly the cheapest one but it also took considerably more time to finish due to the resource shortage it had to face. Indeed, it lasts more than 20 minutes more with respect to the non-OBI configuration with 20 nodes.

Regarding the two less conservative configurations (scaling factor equals to 10 and 8), we can see from Figure 6.1 and Figure 6.2 that we actually have upscaling. Indeed, both configurations reach a maximum amount of 12 cluster nodes by the end of the job execution. However, while the configuration with scaling factor equals to 10 stays to 4 nodes for most of the job execution, the configuration with scaling factors equals to 8 is a little bit more aggressive, using 6 nodes for basically the same amount of time during which the other configuration uses 4 of them which explains the difference in the costs of the two execution. Surprisingly, from a duration point of view the jobs executed with the scaling trigger equals to 8 are slower with respect to the other configuration, despite having on average more nodes. We can explain this behaviour by considering that, whenever new nodes are added, input data needs to be sent to the newly allocated containers, causing some overhead. Since with the scaling trigger equals to 8 configuration we have more scaling operations (as well as a preemption as visible in Figure 6.2), this explains us why with a more aggressive scaling approach we may pay some overhead penalty.

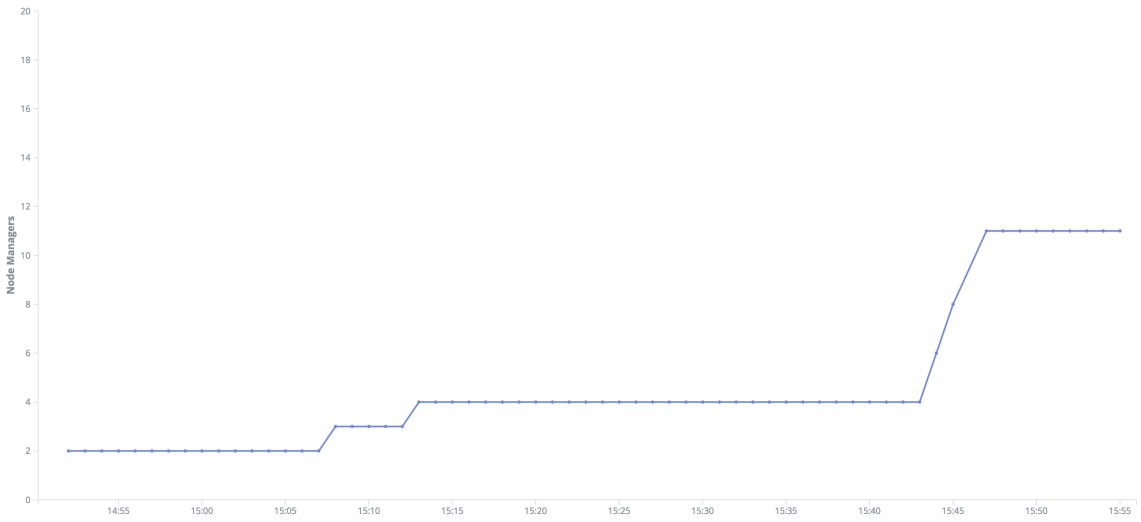

Figure 6.1. Example of cluster autoscaling with scaling trigger equals to 10

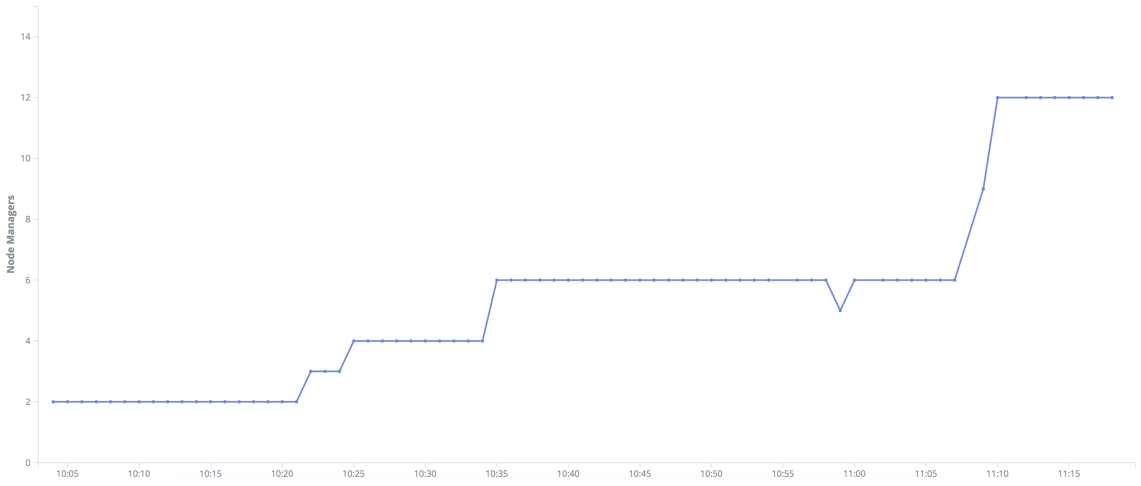

Figure 6.2. Example of cluster autoscaling with scaling trigger equals to 8

Finally, in Figure 6.3 and Figure 6.4 are shown the behaviours we have obtained while running the usual test job with a scaling trigger configuration of 4 and 0 respectively. As we can see, both configuration are much more aggressive with respect to the previous ones, reaching a maximum cluster size of 13 nodes. However, the configuration having scaling trigger equals to 0 is much more aggressive with respect to the other one, reaching the size of 13 nodes soon in the execution window, while the configuration with scaling trigger equals to 4 only reaches that cluster size by the end of the job execution. Also in this case, the cost pattern by which the lower the scaling trigger the higher the cost is respected and also the increase in

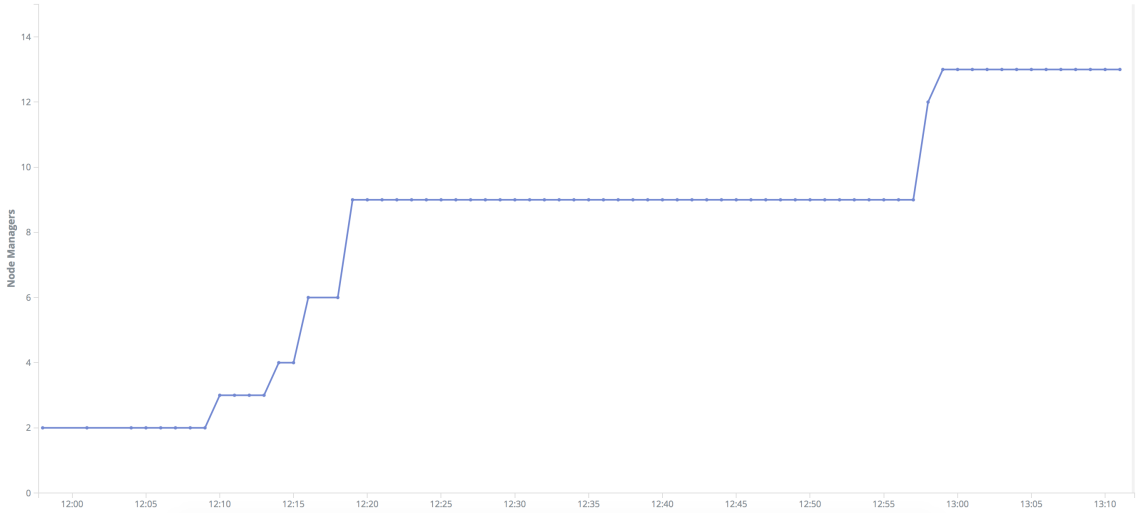

Figure 6.3. Example of cluster autoscaling with scaling trigger equals to 4

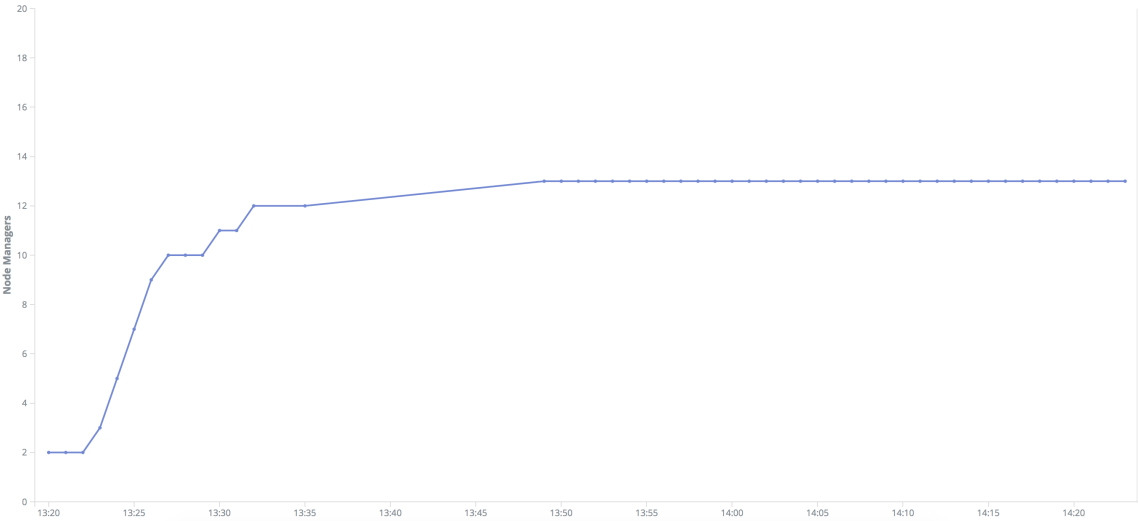

Figure 6.4. Example of cluster autoscaling with scaling trigger equals to 0

job duration is shown in Table 6.2, caused for the same reasons described for the previous case.

Mean	**Variance**
108.12	23.86

Table 6.3. ETL Pipeline cost statistics computed on a one month period of time between October and November 2018

Once we were done with obtaining results from our autoscaler, we moved to testing the whole ETL pipeline with OBI. In Table 6.3 we showed some statistics on the cost of the pipeline, obtained on a one month time frame. In Table 6.4, instead, we show the results we obtained while running our pipeline with different autoscaler configurations. Those results were obtained by executing the same pipeline twice a day and then comparing the results coming from non-OBI and OBI deployments.

In Table 6.4 we did not include results with scaling trigger equals to 16 as we did above because we noticed that some jobs lacked too many resources with this configuration and it was very likely for them to fail, thus resulting in failing pipelines. indeed, jobs at certain stages are quite memory intensive and, if the cluster is not scaled properly, we observed jobs failures due to insufficient resources.

Scaling Trigger	Cost	Cost with OBI	Duration	Duration with OBI
10	112.18 $	36.84 $	4h 38m	4h 56m
8	113.62 $	37.80 $	4h 47m	4h 16m
4	115.53 $	41.71 $	4h 41m	4h 24m
0	112.57 $	39.83 $	4h 43m	3h 51m

Table 6.4. Pipeline Duration for different scaling trigger values

One thing which is clearly visible from Table 6.4 and which may contradict with what we said earlier, is that the more we lower the scaling trigger the more we save time, often requiring less time than the standard pipeline in most of the cases. Despite each pipeline job individually takes longer to execute with respect to their non-OBI counter part, we were able to achieve duration savings by applying a reorganization of the way jobs are submitted to the platform. Indeed, before our work the ETL pipeline jobs were executed as if they were all dependent on each other, resulting in long sequences of jobs in which each one had to way for all the previous ones to start as shown in Figure 6.5.

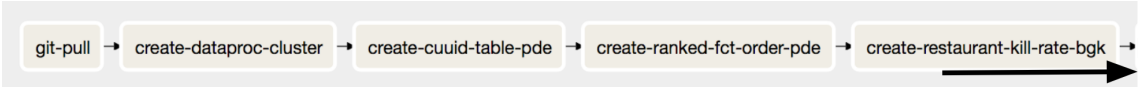

Figure 6.5. Sample ETL pipeline stage scheduling before OBI

However, while developing OBI we noticed that most of the jobs which were executed on a sequential fashion were actually not dependent on each other. Therefore, we rearranged the pipeline stage shown in Figure 6.5 in order to make it looks like what is shown in Figure 6.6, thus allowing a lot of jobs to be executed in parallel for better exploiting OBI's main features e.g. jobs bin packing.

Moreover, we managed to obtain improvements in terms of pipeline duration

also by trying to reduce at the minimum possible the operational overhead of creating clusters. Indeed, we tried to schedule together most of the stages considered lightweight in parallel and all the jobs contained in them were assigned with the same priority level. In this way we were able to execute individual clusters containing up to 14 jobs, all running in parallel.

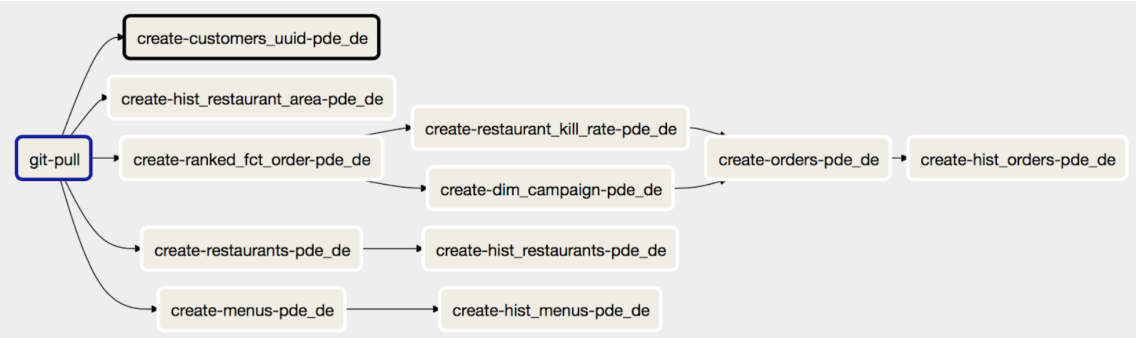

Figure 6.6. Sample ETL pipeline stage scheduling with OBI

In conclusion, what is clearly visible from Table 6.4 is the reduction in terms of costs while using OBI, which is what we were mostly interested in at the beginning of this project. Indeed, as we also observed while running our pipeline on a daily basis, we are able to save the 65.5% amount of money on average while using OBI instead of the legacy pipeline deployed in our environment before.

Chapter 7

Conclusions and Future Works

Our original goal was to see if statistical based predictive approaches were suitable for being implemented in a system having the final objective of saving money. Finally, we can say that according to our results we can conclude that applying predictive approaches to cluster resources scheduling is definitely possible and eventually recommendable. In our work, machine learning was used to generate estimations of the jobs duration and to dynamically resize clusters to perfectly fit the computational needs at any specific point in time. In this way we proved that by using the information our models generated we could take smart decisions which highly contributed into achieving our goal. In fact, we have shown that by exploiting the knowledge we had we were able to save up to 65% amount of money with respect to the deployment scenario without OBI.

The information we relied on are something that any production environment similar to ours have. Indeed, exploiting the fact that the code we were submitting did not change over time, we characterized our jobs based on their workload or, in other words, their input size. In this way we were able to understand how long jobs could possibly take to be executed and schedule them accordingly. As already said, this is quite a common scenario in any enterprise environment or team whose business relies on analysing data which is extracted by the same pipeline on a daily basis.

Since our scenario is likely to be common in any enterprise environment, the same approach we took to job duration prediction could be easily extended to other environments in which there would be the desire to deploy OBI. Indeed, the only thing which would be necessary would be to collect data relevant for the environment's jobs and built proper models on top of them. Those model would then be easily pluggable in OBI's predictive component.

On the other hand, the autoscaler we built is environment-agnostic as it has been trained using data coming from several different naïve autoscalers in conditions in which the type of executed job was not relevant to the problem. Therefore, the

approach we took toward building our autoscaler can easily be extended to other contexts, without requiring much additional effort.

In the end of Chapter 6 we also proved how a simple job scheduling pipeline reorganization can help improving duration as well as costs. Indeed, in cloud computing or, more in general, in distributed computation environments in which business critic jobs are executed, it is crucial to pay as much attention as possible to the way tasks are scheduled, in order to minimize the already high costs of maintaining such an infrastructure.

Despite the impressive results we obtained, there is certainly room for improvement in OBI.

First of all, as it was our original goal, we will implement support to other cloud computing services e.g. Amazon or Microsoft platforms depending on the requirements we will have from other teams. Because of the way OBI was designed, implementing this additional features should be as easy as providing the concrete implementation of the same interfaces we used for Dataproc but wiring it to the appropriate platform type.

Moreover, an interesting yet quite challenging thing would be to develop a job duration prediction model which is independent from environment's information. Despite sounding quite unrealistic, we believe that we could be able to exploit the information contained into Spark's execution DAGs which are generated before a job is actually executed. However, collecting those kind of information in real-time would be quite complex and would require direct contributions to Spark's source code.

Regarding the autoscaler, instead, we have two possibilities for improving.

First of all, we could extend our autoscaling dataset by collect points coming from even more autoscaler policies than those we used so far. This would help broaden the machine learning autoscaler possibilities of learning good behaviours simply because it would learn from more sources.

However, another more interesting possibility for the future would be to explore a reinforcement learning based solution, designing an autonomous agent capable of observing the YARN (or similar) environment and gradually learning what is the best scaling policy by itself. We believe that this approach may increase the performances we obtained with our autoscaler simply because, while we currently have learned a behaviour based on observations taken from other threshold based autoscalers, with a reinforcement learning autoscaler the search space would be infinite, in a sense that we would not be restricted by just certain behaviours coming from certain already implemented algorithms.

Bibliography

[1] Amazon, AWS Autoscaling. https://aws.amazon.com/autoscaling/, 2018. [Online; accessed 17 October 2018].

[2] Google Cloud Dataproc. https://cloud.google.com/dataproc/docs/concepts/overview, 2018. [Online; accessed 10 October 2018].

[3] Google Cloud Platform. https://cloud.google.com/docs/overview, 2018. [Online; accessed 10 October 2018].

[4] Google, Load Balancing and Scaling. https://cloud.google.com/compute/docs/load-balancing-and-autoscaling, 2018. [Online; accessed 17 October 2018].

[5] gRPC, A high performance, open-source universal RPC framework. https://grpc.io, 2018. [Online; accessed 22 October 2018].

[6] Preemptible VM Instances. https://cloud.google.com/compute/docs/instances/preemptible, 2018. [Online; accessed 10 October 2018].

[7] Rectifier (neural networks). https://en.wikipedia.org/wiki/Rectifier_(neural_networks), 2018. [Online; accessed 29 October 2018].

[8] Sigmoid function. https://en.wikipedia.org/wiki/Sigmoid_function), 2018. [Online; accessed 29 October 2018].

[9] Spotify Spydra, Ephemeral Hadoop clusters using Google Compute Platform. https://github.com/spotify/spydra, 2018. [Online; accessed 17 October 2018].

[10] Stolon, PostgreSQL cloud native High Availability and more. https://github.com/sorintlab/stolon, 2018. [Online; accessed 22 October 2018].

[11] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. *CoRR*, abs/1603.02754, 2016.

[12] X. Chen, C. Lu, and K. Pattabiraman. Predicting job completion times using system logs in supercomputing clusters. In *2013 43rd Annual IEEE/IFIP Conference on Dependable Systems and Networks Workshop (DSN-W)*, volume 00, pages 1–8, June 2013.

[13] François Chollet et al. Keras. https://github.com/fchollet/keras, 2015.

[14] Xavier Dutreilh, Sergey Kirgizov, Olga Melekhova, Jacques Malenfant, Nicolas Rivierre, and Isis Truck. Using Reinforcement Learning for Autonomic

Resource Allocation in Clouds: towards a fully automated workflow. In *7th International Conference on Autonomic and Autonomous Systems (ICAS'2011)*, pages 67–74, Venice, Italy, May 2011.

[15] J. Huang, C. Li, and J. Yu. Resource prediction based on double exponential smoothing in cloud computing. In *2012 2nd International Conference on Consumer Electronics, Communications and Networks (CECNet)*, pages 2056–2060, April 2012.

[16] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[17] Miroslav Kubat. Reinforcement learning by ag barto and rs sutton, mit press, cambridge, ma 1998, isbn 0-262-19398-1. *Knowl. Eng. Rev.*, 14(4):383–385, December 1999.

[18] Bingwei Liu, Yinan Lin, and Yu Chen. Quantitative workload analysis and prediction using google cluster traces. *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 935–940, 2016.

[19] Luca Lombardo. *Autoscaling mechanisms for Google Cloud Dataproc*. Politecnico di Torino, 2018.

[20] Tania Lorido-Botran, José Miguel-Alonso, and José Antonio Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of Grid Computing*, 12:559–592, 2014.

[21] R M Sakia. The box-cox transformation technique: A review. 41, 01 1992.

[22] Sarwar Morshed Mohammad Manzurul Islam and Parijat Goswami. Cloud computing: A survey on its limitations and potential solutions. *International Journal of Computer Science Issues*, 2013.

[23] Irakli Nadareishvili, Ronnie Mitra, Matt McLarty, and Mike Amundsen. *Microservice Architecture: Aligning Principles, Practices, and Culture*. O'Reilly Media, Inc., 1st edition, 2016.

[24] Amadi Emmanuel Chukwudi Ndukwe Ifeanyi G. Cost benefits of cloud vs. in-house it for higher education. *International Journal of Computer Science and Security (IJCSS), Volume (7) : Issue (1)*, 2013.

[25] Michael A. Osborne, Roman Garnett, and Stephen J. Roberts. Gaussian processes for global optimization. In *in LION*, 2009.

[26] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[27] Adrian Daniel Popescu, Vuk Ercegovac, Andrey Balmin, Miguel Branco, and Anastasia Ailamaki. Same queries, different data: Can we predict runtime performance? In *ICDE Workshops*, pages 275–280. IEEE Computer Society, 2012.

[28] Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2005.

[29] J. Sola and J. Sevilla. Importance of input data normalization for the application of neural networks to complex industrial problems. *IEEE Transactions on Nuclear Science*, 44(3):1464–1468, June 1997.

[30] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

[31] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16, New York, NY, USA, 2013. ACM.

[32] M. Wajahat, A. Gandhi, A. Karve, and A. Kochut. Using machine learning for black-box autoscaling. In *2016 Seventh International Green and Sustainable Computing Conference (IGSC)*, pages 1–8, Nov 2016.

[33] Muhammad Wajahat, Alexei Karve, Andrzej Kochut, and Anshul Gandhi. Mlscale: A machine learning based application-agnostic autoscaler. 10 2017.

[34] Kewen Wang and Mohammad Maifi Hasan Khan. Performance prediction for apache spark platform. In *HPCC/CSS/ICESS*, pages 166–173. IEEE, 2015.

[35] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.