university of groningen

faculty of mathematics and natural sciences

Efficient cloud scaling using the Lambda Architecture

Master's thesis

October 2015

Student: H.B. van Apeldoorn

Primary supervisor: Prof. Dr. A. Lazovik

Secondary supervisor: Dr. A. Ampatzoglou

Secondary supervisor: Ir. J.S. van der Veen (at company TNO)

ABSTRACT

Cloud computing takes in an ever more important role in the IT division of companies. In the past everyone typically had their own dedicated computers. Nowadays computing power can be bought from cloud providers though, saving companies from the hassle of having to maintain their own machines. Computing power does not come free of charge however. In cloud computing, as in many areas, efficiency is thus key.

All the major cloud providers offer an automatic cloud scaler to use the cloud efficiently. Whenever computational load is high, it adds computing power and whenever load is low, it does exactly the opposite. This has proven to be a proper scaling method. Cloud providers only offer on-demand scaling however. Historical cloud usage patterns are not used in scaling clouds.

In this thesis we therefore investigate the usage of a combination of historical and recent cloud usage data for automatic cloud scaling. The Lambda Architecture is proposed as a way to process both types of data. To prove the feasibility of this architecture for cloud scaling, a software solution is implemented in which more efficient scaling can be reached. Furthermore, simulations are ran on the software solution using only on-demand scaling as well as using Lambda Architecture for scaling.

ACKNOWLEDGMENTS

I would like to express my gratitude to my daily supervisor at TNO, Ir. Jan Sipke van der Veen, for his guidance and technical expertise during my thesis. I would also like to thank my supervisor at the RUG, Prof. Dr. Alexander Lazovik, for his enthusiasm and feedback on the subject and on the written work. This greatly helped me during my research. Furthermore, thanks go out to Dr. Apostolos Ampatzoglou as second reader of my thesis project.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

DEFINITIONS

COMPUTING UNIT / MACHINE A single computing unit capable of carrying out computations. This is typically one virtual machine within a computing cluster.

COMPUTING CLUSTER A group of computing units that are geographically located in the same place. This can be a public cluster like Amazon's cluster in Frankfurt, but also a private cluster that is hosted by a company itself somewhere within their own building.

AWS Amazon Web Services. A cloud provider offering scalable computing power. AWS sells usage of their computing clusters to customers.

1 INTRODUCTION

Over the last few years cloud computing has become ubiquitous. It is used for checking mail online, logging into your company's working environment, doing large-scale computations, streaming a movie and many more. The number of applications is endless.

Cloud computing can be defined as the providing of software and hardware resources as services across distributed IT resources[1]. It is a rapidly expanding field that offers immense computing power. Cloud computing is offered to customers as one of the following three distinct services: Software-as-a-Service, Platform-as-a-Service or Infrastructure-as-a-Service, where the last named offers users the most freedom. Although all three services offer cloud computing in a different way, they typically share the feature that the user has to pay the public cloud provider for its use.

Publicly available cloud computing arose at the beginning of the twenty-first century. One of the first public cloud computing providers, and also one of the largest, is Amazon Web Services (AWS)(5). AWS started out in 2006 and have expanded their services ever since[2]. Other companies such as Microsoft[3] and Google[4] quickly followed with their own platforms. All these companies charge their customers for every hour and for every machine that is used. It is thus imperative for customers to use the given computing power as efficient as possible. This is where automatic cloud scaling comes in.

1.1 AUTOMATIC CLOUD SCALING

Computing power demand usually varies over time. There are times when extra computing power is needed, but more importantly, often there is a great amount of idle computing power. One of the great advantages of the cloud is the fact that it is scalable. Whenever load is low, computing power can be removed and the opposite also holds true for high load. To this end most public cloud providers offer an automatic cloud scaler. This is a piece of software that can monitor cloud usage and scale computing power based on the load on such

computing units. Some cloud providers even allow the user to set a policy of their own for scaling[5].

Both cloud providers' own policy and a user set policy depend on recent data. These policies do not look at historic patterns in data usage, but instead measure the load on the computing units at this very moment. The cloud scaler receives a continuous flow of information about the current load on the used computing units. This reactive kind of data processing is called stream processing. Another type of processing in the same field is batch processing. It is currently not used for automatic scaling as it lacks the possibility to adopt to sudden changes in load. In this thesis we therefore look at a recently coined paradigm called the Lambda Architecture which combines both types of processing.

1.2 LAMBDA ARCHITECTURE

Lambda Architecture is a term coined by Nathan Marz in 2012[6]. It is used in the field of data-processing. It describes a paradigm which takes advantage of both batch processing and stream processing. Typically a continuous stream of data is expected as input in the Lambda Architecture. This data is then dispatched to two layers; speed layer (stream processing) and batch layer (batch processing). Batch processing ensures that a massive amount of data can be processed while stream processing provides the latest up-to-date data. Processed data from both layers is then made available in a uniform way in the query layer. Other programs can extract data from the Lambda Architecture by performing queries on the query layer.

1.3 PROBLEM STATEMENT

Ever since the emergence of cloud computing computer scientists have been looking for ways to use the cloud as efficient as possible. Clouds possess an enormous amount of computing power which users can order when they need it. In general users have to pay for every computing unit(5) of the cloud they use though. It is therefore desirable to use as little resources as possible to do computations.

The size of computations that need to be done tends to vary over time. As such, the amount of computing power one may need from

the cloud also varies. Being able to adapt the amount of computing power to fit your needs is key in achieving high efficiency. The easiest way to achieve this would be to simply have a programmer manually shut down computing instances when there are little computations to be done, and start up extra computing instances when a bigger batch of computations needs to be done. A human has its limits and flaws of course when it comes to rapid decision making. Software programs and techniques have been developed instead to take over this task.

Software programs are indeed well-versed in rapid decision making. They cannot create an ideal situation however. They always have to balance the trade-off between latency and costs. More computing units offer more computing power and thus lower latency. At the same time this increases costs. One would want to have as little latency for as little money as possible.

To balance this trade-off between latency and costs, software programs typically look at how busy your computing cluster(5) is currently. In case it is very busy, extra machines may be added. In case there is excess computing power, available machines may be removed from your cluster. To make these decisions a program could possibly also look at data from longer ago and look at certain trends in this data to make a prediction on how busy the cluster will be.

Briefly summarized, there can be made a distinction between two kinds of historical data: data from over a longer period of time (for example, previous year) and data from a very short recent period (for example, the last ten minutes). The former can be processed well by using the MapReduce paradigm. MapReduce is not suited for the latter. The Lambda Architecture is capable of utilizing both. It has separate layers for processing data batches and processing streaming data. Furthermore, most of the work that has been done in this field of research applies to scaling with a single cloud. It is possible that one may want to use more than 1 cloud provider, for example for financial reasons or not being dependent on one cloud provider.

One possible scenario where the aforementioned applies to, and which is used in this thesis, is at a picture/video uploading website. This use case is discussed in section 3.1. Summarized this comes down to the following: Users from across the globe upload pictures and

videos to this site. In general pictures and videos are quite large files which means that it will take some time to upload them to the website. Users want to have their pictures and video uploaded as fast as possible (thus with as low latency as possible). On the other hand, the owner of this website wants to use his computing power as efficient as possible. More computing power requires more machines which requires more money. Location of computing power also takes an important role here.

Having your computing cluster geographically close to your users reduces the amount of latency these users experience. The owner of the picture/video uploading website would therefore like to have servers running in many different parts of the world. Ideally there should be more servers where there are currently more users active. E.g. when you currently have a large number of users active in Europe it is desirable to also have a major part of your servers active in Europe. The number of users is always changing however. Also, In some cases it may be beneficial to redirect your users to another computing cluster that is further away and thus gives a higher response time when under equal load. One of these cases is when you have another cluster that is geographically further away from the majority your currently active users but it has excess computing power.

These problems are not solved well by a standard autoscaler as by default it will always choose to add extra computing units to the location where demand is the highest and the autoscaler will not look at the possibility of efficiently distributing users to other computing clusters. A solution to this problem is to use the Lambda Architecture.

As mentioned previously, the Lambda Architecture provides a way to use both historical and recent data to make a decision on how to scale your different computing clusters. A smart scaler can then take both kinds of data into account and determine whether a good result is achieved both latency-wise and cost-wise when extra machines are started in the same cluster or when requests are offloaded to other computing clusters. This could provide cost saving as distribution load is spread more evenly amongst the clusters. In this thesis we therefore investigate the usage of the Lambda Architecture for automatic scaling in the cloud to efficiently reduce latency.

The main research question answered in this thesis is: *Can we im-*

prove automatic scaling to efficiently reduce latency using the Lambda Architecture? Sub research questions can then roughly be structured as follows:

- Does using both batch processing and stream processing at the same time generate significant overhead?
- How does combined processing compare to only batch processing?
- How does combined processing compare to only stream processing?

1.4 THESIS STRUCTURE

This chapter has provided a brief introduction into the subject of cloud scaling and the Lambda Architecture as well as the research questions. To answer the research question the rest of the thesis is structured as follows: Chapter 2 discusses work previously done in this field. Chapter 3 covers the theory and design. Then the implementational work of this thesis is discussed in chapter 4. After that the outcome of the simulations ran is depicted and evaluated in chapter 5. Recommendations for future work are done in chapter 6. Finally, the conclusion and answers to the research questions are given in chapter 7.

2

RELATED WORK

Many aspects of cloud computing are contained within this thesis: cloud scaling, handling Big Data, batch processing. The list is quite extensive. As such, there are many papers that can be considered related work. The only aspect which proved to be hard to find scientific material for, is the Lambda Architecture.

2.1 LAMBDA ARCHITECTURE

Several scientific paper databases (IEEE Xplore, Elsevier ScienceDirect, Google Scholar, dblp) have been thoroughly searched for research about the Lambda Architecture. During this search the keywords *Lambda Architecture* were used. No articles have been found that specifically treat the Lambda Architecture. This is possibly due to a few reasons. The first reason is that this architecture has been developed quite recently and as such research may still be underway. The second one is that this article is not that well known yet. The creator of the Lambda Architecture, Nathan Marz, has just finished his book (May 10, 2015) which describes the inner working of the Lambda Architecture. Furthermore, the name was coined a few years ago. However, this kind of architecture may already be used by others but they may use a different name. Searching with different keywords, for example *batch and streaming processing* does provide results. This gives an indication that there has been research about this architecture before, but not under the name *Lambda Architecture*. The rest of this paragraph thus describes techniques, architectures, software, middleware that fulfill similar roles as the Lambda Architecture, providing both stream and batch processing.

The paper written by Dahiphale et al. describes a technique they call Cloud MapReduce[7]. Contrary to regular MapReduce, Cloud MapReduce does not only process in batches, but also allows for stream processing. This shows great similarities with the set up of the Lambda architecture. Cloud MapReduce uses queues to support both batch and stream processing. New streaming data is treated by a StreamHandler thread. Whenever new data is found, it is split up

in chunks and then pushed to an input queue. Batch data is pushed into similar input queues. All data is then processed through several queues and mapping and reducing.

Another similar approach is taken by the Hadoop Online Prototype (HOP). This technique is described in the paper by Condie et al.[8]. Instead of waiting for entire batches to finish a MapReduce process, HOP returns intermediary results. Furthermore, it is capable of handling continuous queries. This means that data that is available just now, for example streaming data, can be added to the MapReduce process. The most appealing difference with default MapReduce is that reduce units can start to produce intermediary results as soon as mappers have some data available, also called online aggregation[9]. Reduce units do not have to wait until the entire mapping procedure has been finished.

The paper of Urbani et al. describes the middleware they have written, called AJIRA[10]. AJIRA addresses some of the shortcomings of the original MapReduce. The most important shortcoming of MapReduce is that it does not allow the creation of new processes at runtime. AJIRA does allow this. This means that AJIRA cannot only handle batch processing but stream processing as well. AJIRA's most important unit is an *action*. Multiple actions perform a generic task in the MapReduce process, for example grouping. A sequential set of orders is called a *chain*. These chains can then be executed on computing nodes. These make up the MapReduce like process AJIRA performs.

There are a few implementations of the Lambda Architecture however. (These are not documented in scientific papers as of yet.) The one that is currently the most mature is Summingbird [11], although this one is also far from finished. Summingbird uses the Lambda Architecture to handle large amounts of data. A number of different programs can be used both for the speed layer (E.g. Storm[12]) and the batch layer (E.g. Scalding[13]).

2.2 SCALING OF CLOUDS

Dejun et al. have done research on how workload should be scaled among heterogeneous clouds[14]. In the paper they describe a way to determine how much workload should be assigned to a newly added

computing unit. This proves to be useful information for our own work. Since multiple clouds are used, there is a high probability that not every computing unit is able to take on the same working load as any other.

The work done by Gandhi et al. describes the effect of horizontal and vertical scaling of workload on computing units[15]. Horizontal scaling is done by adding more computing units to handle additional traffic. Vertical scaling is done by adding more resources to the current computing units (E.g. by adding memory). Models are provided to determine when an increasing workload should be answered with vertical scaling and when it should be answered with horizontal scaling.

MODAClouds[16] takes a different approach. It provides system developers with a framework in which they can develop their applications. This framework ensures that your application is cloud-agnostic. As such it requires little effort to deploy your application on one cloud and to port it to another. MODAClouds considers risks, price and QoS to determine where an application should be deployed.

Many papers have been written about cloud computing. Cloud computing is a very profitable and ever-growing business which means that beside papers, there are also many commercial solutions for cloud computing problems. Some cloud providers also provide their own scalers, such as Google's[17] and Amazon's autoscaler[5].

InterCloud[18] focuses on offering a user a private network of clouds. InterCloud offers an extra software layer that can communicate with different clouds and cloud providers. InterCloud also includes automatic scaling and load distribution to reach reasonable QoS levels. What InterCloud exactly offers has been described in the paper[19].

RightScale[20] offers similar functionality as InterCloud. This software supports many of the large cloud computing providers. This includes some of the cloud providers that are used in the concept section of this thesis; Amazon Web Services(AWS)[21] and OpenStack[22]. RightScale can offer a combination of public, private and hybrid clouds to their customers.

Visualization of performance metrics of computing units can be done

by a combination of tools; collectd[23] and ElkStack[24]. collectd is used to collect performance metrics and store them in files. Elkstack provides visualization of your log files. It consists of three parts: Elasticsearch, Logstash and Kibana. Elasticsearch provides the user data analytics. Logstash is used for managing events and logs and parsing them. Kibana provides visualization of the previously mentioned data. This combination proves to be a useful aid for developers to see how computing units are performing.

3

DESIGN

The following section describes the concept of using the Lambda Architecture for automatic cloud scaling. Furthermore, a use case is presented to clarify this usage.

3.1 FOLLOW THE SUN

To demonstrate the use of the Lambda Architecture for automatic scaling, we pick the *Follow-the-sun* principle as use case. The *Follow-the-sun* principle entails that your business is active wherever the sun is shining. This ensures that you always have at least one business active around the world. For example, there could be three places around the world where you have offices, London, Mexico City, Singapore. At 9 am (GMT) employees start their working day in London. Eight hours later when the sun sets in London employees finish their working day. Sun rises in Mexico City at that time (9 am, GMT-8) and employees start their day there. Eight hours later the same applies to employees in Singapore. After another eight hours it's 9 am (GMT) again and the cycle starts over.

This can also be applied to VM's (Virtual Machines,*computing unit* is used often throughout this thesis as a more general term). Users are typically active during the day. In general, placing VM's geographically far away from the user results in a higher number of network and thus in higher latency because of transmission delay and retransmission. Placing VM's closer to the users will likely result in lower latency. Hence it is beneficial to also apply the *Follow-the-sun* principle to VM's. To be able to deploy such tactics cloud providers have to have computing units available to cover every part of the world. Fortunately, most bigger cloud providers have data centers all over the world.

Amazon Web Services offers VM's in North America, Europe, Africa and Asia[25]. These regions can fulfill the *Follow-the-sun* principle together. The same applies to Rackspace. This cloud provider has data centers in North America, Europe, Asia and Australia[26]. The third

cloud provider that is used in this thesis is GoGrid which offers VM's in only Europe and North America[27]. Using multiple clouds in this use case, allows us to take advantage of the different locations of these cloud providers.

In this thesis we deploy applications on three different regions throughout the day. It is possible that deploying to even more regions increases the quality you can offer to your users. However, constantly turning on and shutting down machines is rather expensive since cloud providers in general charge you for a certain amount of hours as a minimum. It is thus not desirable to use VM's in as many regions as possible during a day. Most cloud providers provide VM's in at least three regions which can cover the entire 24 hours: North America, Europe and Asia. It is therefore desirable to follow this pattern. In this thesis we activate VM's in these three regions depending on the time of day. The Lambda Architecture is used to determine when and how many machines we activate in each of these regions.

To implement realistic computer usage during the day, data from Internet exchanges is used. We use this data to create user profiles. A user profile defines the activity of a user of an application on our clouds during the day. Typically we see high Internet usage between 9 am and 5 pm and little usage at night.

Since we use VM's in regions North America, Europe and Asia, we choose to gather information from Internet exchanges in these regions. To gather data about European users, the Amsterdam Internet Exchange[28] is used. For the America's we use the New York Internet Exchange[29]. Finally, for the region Asia we use the Internet Exchange of Tokyo[30]. These Internet Exchanges provide statistics about all the Internet traffic that is routed through them. The absolute amount of data that flows through an Internet Exchange is not interesting for this thesis. Rather, we want to look at the relative size of traffic at a certain moment of the day. For example, if an Internet Exchange shows that data traffic throughput is four Tb/s at 5 pm and one Tb/s at 5 am, then we can model a user profile that puts a four times as heavy load on the cloud at 5 pm than at 5 am.

It can be observed from the three Internet Exchanges[28][29][30] that every day of Internet traffic is roughly the same. Figures 1, 2 and 3 show the amount of internet traffic over the course of May 11th 2015

(Monday). Furthermore, we are mainly interested in having a period during the day of higher load and a period of lower load. The user profiles can thus be based on any day given in the statistics of the Internet Exchanges.

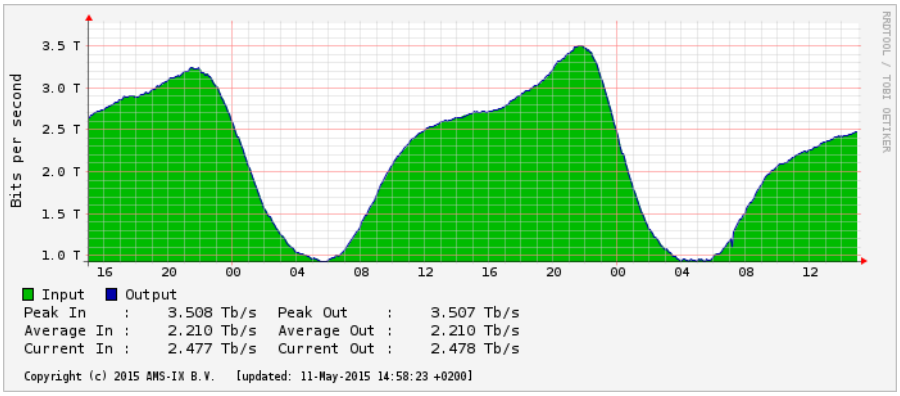

Figure 1: Internet traffic of Amsterdam Internet Exchange

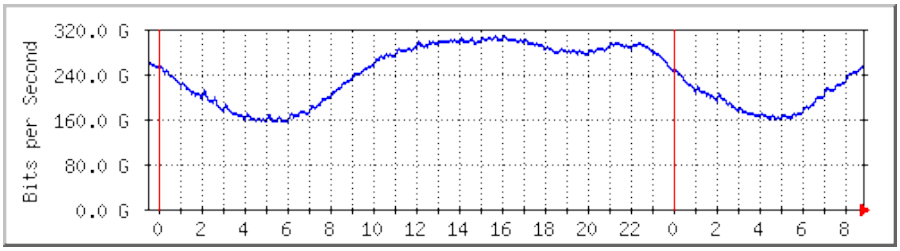

Figure 2: Internet traffic of New York Internet Exchange

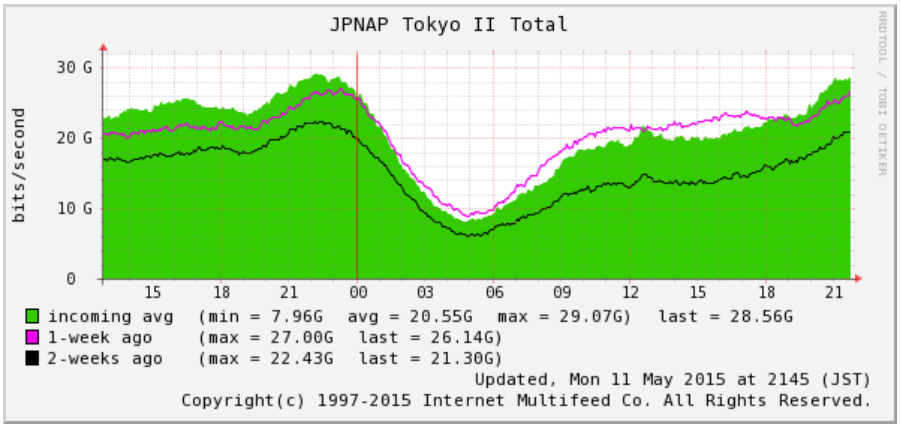

Figure 3: Internet traffic of Tokyo Internet Exchange

3.2 MULTIPLE CLOUDS

Cloud providers typically offer many useful tools to make it easier to maintain and adjust your cloud. Amazon Web Services for instance offers 64 different services[31]. All cloud providers have their own set of products. These cannot be used across clouds however. An AWS Elastic Load Balancer for example cannot balance the load on computing units owned by GoGrid.

One of the selling points of using the Lambda Architecture in this subject is that we can use it across multiple different clouds. It is therefore recommended to not use any cloud-specific tools. The implementation in this thesis is therefore completely cloud agnostic. Furthermore, a cloud scaler typically takes advantage of many features that a cloud provider offers. This makes it hard for the approach in this thesis using the Lambda Architecture to beat such a cloud scaler one on one. Yet this approach can be used on multiple clouds whereas the cloud scaler will only work for that one specific cloud provider.

3.3 FEEDBACK LOOP

A feedback loop is used in the IT world mainly for evolution and maintenance of software[32]. Here we use it to adjust the size and placement of the cloud appropriately to achieve automatic cloud scaling. Data is gathered from the cloud computing units and then processed and in turn used to scale those same cloud computing units. This means that the feedback loop consists of multiple components. These are displayed in figure 4. The main components are the cloud computing instances, the Lambda Architecture block, the rules and constraints set and the cloud scaler. Information is gathered from the cloud computing units. The data is then sent to the Lambda Architecture block which is explained in detail in the next section. The data is processed and the relevant data is then made available for queries for the rules and constraints set, which is elaborated on later in this chapter. Finally, instructions are given to the scaler to adjust the cloud units appropriately.

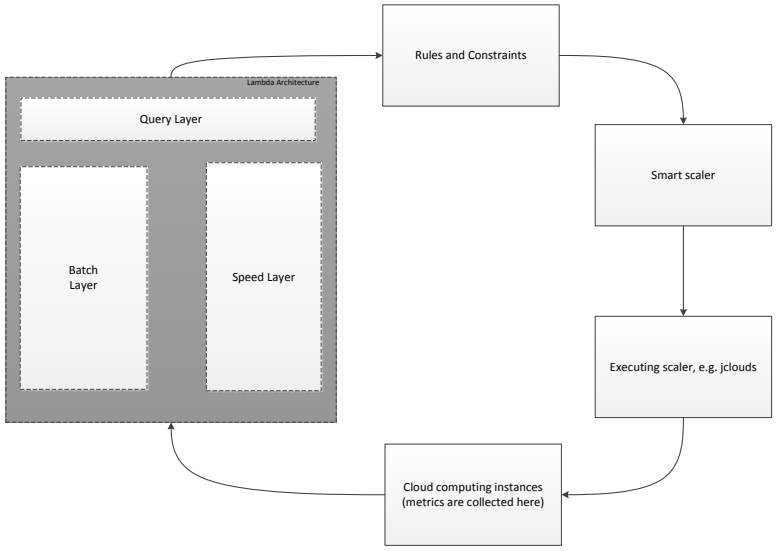

Figure 4: Feedback loop using the Lambda Architecture.

3.4 LAMBDA ARCHITECTURE

The Lambda Architecture dictates that data should be processed in two layers: the batch layer and serving layer for all older data and the speed layer for only the most recent data. The part of the feedback loop that is structured by the Lambda Architecture receives metrics taken from the cloud computing instances. This new data is pushed into both layers. Figure 5 shows a zoomed in picture of all components of the Lambda Architecture used in this thesis.

New data is gathered from the different cloud providers every few seconds. There is no need to wait with pushing data until metrics are gathered from all cloud providers. The metrics that are gathered should be obtainable in all cloud providers to maintain consistency. Naturally, these metrics should give a good insight into how busy the currently used computing units are and thus if we need to scale up or down. The chosen metrics are dependent on the needs of the scaler of the feedback loop.

3.4.1 *Batch layer*

New data that is pushed to the batch layer is stored in the master database at first. The master database contains an ever-expanding set

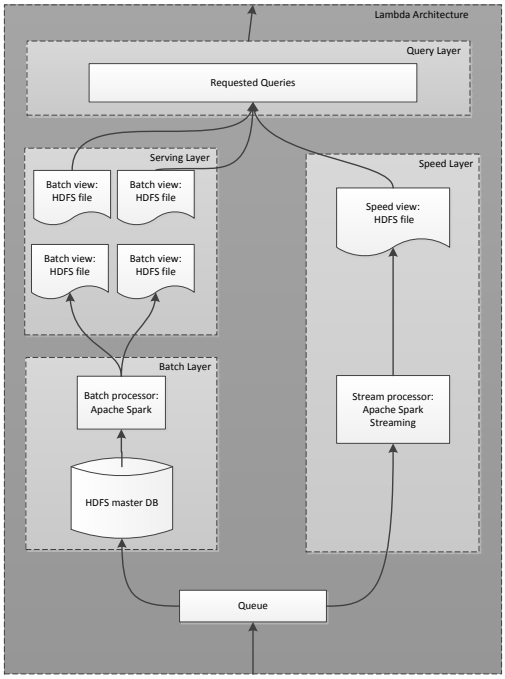

Figure 5: Feedback loop using the Lambda Architecture.

of historical data that is gathered during the time the program is used. In this layer functions are precomputed to be used in the serving layer later on. One of these functions may be to calculate the average CPU usage of all computing instances of all cloud providers for a week. This can provide a useful insight into the possible CPU usage for next week and can tell us if we need to scale up or scale down.

3.4.2 *Serving layer*

The serving layer uses the precomputed functions from the batch layer. The precomputed functions take a considerable amount of time to compute. The serving layer therefore faces high latency. The speed layer makes up for that to ensure that queries always use the newest data. The serving layer consists of a number of batch views that can answer a query together with the real-time view of the speed layer.

3.4.3 *Speed layer*

The speed layer receives data directly from the cloud computing units. There is no database or any other intermediary tool in between. The speed layer provides low latency updates and thus allows you to do computation on data in near real-time. The speed layer handles data from the last hour. Batches in the batch layer are expected to finish well within one hour. Data that is older than one hour is thus already used in functions of the batch and serving layer.

3.4.4 *Queries*

A query is done on the real-time view from the speed layer as well as the batch views from the serving layer. In this way a query can always be done on the most recent data. The result of a query is sent to the next part of the feedback loop which consists of applying rules and constraints on the result.

3.4.5 *Data throughput*

The Lambda Architecture consists of several components that all have their own rate of throughput given the number of computing units over which such component is deployed. A vital factor is the amount of data that is passed through the Lambda Architecture. This factor not only determines the size of computing units needed to deploy the Lambda Architecture, it proves or disproves the feasibility of some components as a whole as well. Data throughput may be relatively low such that queries done directly on the files in the serving layer are of sufficient speed. In this case we could skip having an in-memory key-value data store between the serving layer and the queries that come from the rules and constraints section. There would be no gain of having such a data store in between. On the other hand if we see that data throughput is still fairly high, we need an in-memory key-value data store because queries and thus scaling of the cloud would take too much time. Data throughput can thus answer the question if an extra component such as an in-memory key-value data store is needed.

To get an idea of the data throughput some calculations need to be done. Please note that these numbers are only to show the order of

magnitude of the data input. They are not used to produce exact numbers. The first one is to calculate how often data should be collected from the computing units in the cloud: The major cloud providers mostly use billing by hour[33][34][35]. Even if you only need a machine for 5 minutes and terminate it afterwards, you will be billed for an entire hour. Scaling down is thus a slow process. Scaling up however can be done more rapidly. It takes most cloud providers just a few minutes to boot an instance with the proper software installed [36][37]. Therefore data should be collected in the order of magnitude of 1-10 seconds. This ensures having enough information to know whether to scale up or scale down.

Secondly, the amount of data that one computing unit sends every few seconds is important. On a local machine a few megabytes of data were collected for the last four hours. Data was gathered for a total of 6 metrics. This amount roughly comes down to a few kilobytes of data for one time interval. This number has to be multiplied by the number of computing units. There are some large companies that rely heavily on the cloud. Imgur uses around 500 servers to host their content[38]. Dropbox uses approximately 10,000 servers to store all user content and serve their website and the same holds for Netflix[39][40]. Taking these numbers into account, data throughput in the Lambda Architecture for a large resource intensive website generates tens of megabytes per second of data.

This stream of data is passed both to the master database in the batch layer and to the streaming processor in the speed layer as well. For the master database this means the following: This data should typically be stored in the master database for years until it is decided that this data is not relevant any more. The size of the master database would thus be in the order of magnitude of hundreds of terabytes.

Both in the streaming processor (streaming layer) as in the batch processor (batch layer) the output is severely smaller than the input. This is caused by the fact that averages are produced as output where each average takes many raw measurements as input. As an example, one averaged cpu cluster load over one hour is calculated using the following data:

$$clusterAverageHour = \frac{N * 60 * 60}{T}$$

In which:

- N = the number of units within one cluster
- T = the interval in seconds at which data is collected

For our example of a large website, 10,000 servers and a ten seconds time interval, this means you would need $\frac{10,000*60*60}{10} = 3,600,000$ raw measurements to produce just one averaged cpu cluster load over one hour. This illustrates the major difference in size between in- and output of the batch and streaming processors.

The data made available for queries is thus in the order of magnitude of gigabytes. Queries typically only comprehend a small portion of the total query layer. This information can be used to answer the question posed at the beginning of this section: The data available at the query layer is not large enough to justify using an in-memory key-value data store for quicker query responses.

The previous paragraphs explain the data throughput given an interval, number of servers and number of metrics that is realistic. To get a better understanding of the amount of data that is generated table 1 is provided. The first row in this table describes the outcomes of previous paragraphs. Similar logic is used to obtain the results of other rows. The interval, number of servers and number of metrics are given as input parameters. It can be seen that reducing certain input values drastically reduces overall data throughput. Such cases require a different set up of the feedback loop. E.g. the previously discarded in-memory data store may be viable when the number of servers is doubled.

The columns of the table contain the following information:

- **computing units:** The number of computing units that generate data
- **metrics:** The number of metrics for which each computing unit generates data
- **Time interval:** Time between two consecutive moments of data gathering

- **Data size one unit:** Amount of data one computing unit generates for one time interval

- **Data size master DB:** Amount of data present in master database

- **Data throughput batch processor:** Amount of data that is generated by the batch processor each second

- **Size query layer:** Amount of data that resides in the query layer and is thus available to perform queries on

computing units	metrics	Time interval	Data size one unit	Data size master DB	Data throughput batch processer	Data throughput stream processer
10,000	1	1 second	tens of bytes	tens of terabytes	hundreds of megabytes	hundreds of megabytes
10,000	1	10 seconds	tens of bytes	tens of terabytes	hundreds of megabytes	hundreds of megabytes
10,000	6	1 seconds	hundreds of bytes	hundreds of terabytes	gigabytes	gigabytes
10,000	6	10 seconds	kilobytes	hundreds of terabytes	gigabytes	gigabytes
10,000	24	10 seconds	kilobytes	hundreds of terabytes	gigabytes	gigabytes

Table 1: Data throughput in the feedback loop given a number of computing units, number of metrics and time interval

3.5 RULES AND CONSTRAINTS

The main focus of this thesis implementation-wise is on the Lambda Architecture. The Lambda Architecture has been implemented many times before. Even before the term Lambda Architecture was coined, the combination of stream and batch processing has been used commonly to process data. The main focus of this thesis novelty-wise lies mostly in the rules and constraints section instead.

Rules and constraints tell us when extra cloud computing units are needed and when we can do with less. The more well-known cloud providers often provide the user an interface in which he can define some simple rules[17][5]. These rules are usually related to costs and several CPU usage statistics. Furthermore, most cloud providers also include an automatic cloud scaler[17][5]. An automatic scaler can then scale the cloud up or scale the cloud down based on these rules. These automatic scalers respond to recent events. E.g. if a very high CPU load is detected on all cloud computing units (say, greater than 95%) then extra computing units will be added. The Lambda Architecture offers us a method to not only take recent events into account but also historic events.

To demonstrate the usefulness of the historic events we take the aforementioned situation in which all cloud units have high CPU load. As an example we say that this situation occurs just before 5 pm at a given working day. If we only apply the knowledge we have from recent events, we would start up many extra machines to deal with the high CPU load. Using historical information we can see that every day after 5 pm cloud usage decreases dramatically due to people leaving their offices and thus that we do not need to start extra machines. This cuts back the costs while not showing a significant increase in latency for the users of the cloud units.

3.6 AUTOMATIC SCALING

The data that a certain query should retrieve is determined by a smart scaler. This scaler takes into account technical statistics such as how heavy the load is that the computing units have, but also economical

considerations such as the current price for hiring an extra computing unit in a certain cluster. It will likely also involve multiple queries that are joined, averaged and even more. Furthermore, it may be beneficial to start computing units in a cluster that is far away from the users since that cluster offers cheaper machines at that moment. Finding the sweet spot for when to request an extra computing and when not to, forms an econometrical case on its own and will thus not be covered in this thesis. In the implementation a severely simplified set of rules and constraints is used instead to complete the feedback loop.

Based on the queries done above, the smart scaler makes a decision to start or shut down computing units in certain clusters. This completes the feedback loop. The whole process is continuously repeated. At 1 past 9 this process is nearly the same. Except for the fact that the batch layer now also has incorporated data from 8 am to 9 am and the speed layer only contains data from 9 O'clock until 1 past 9. The rest of the day follows a similar process. On Tuesdays, different historical data is queried. That is, historical data from all Tuesdays instead of all Mondays.

3.7 APPLICATION TO THE "FOLLOW-THE-SUN" USE CASE

Earlier in this chapter we discussed the follow-the-sun use case. Specifically, how we could make use of the Lambda Architecture in this use case. The following paragraphs explains how data in this use case is passed through the feedback loop. Each component of the feedback loop is addressed and we show what each component does with the received data.

The scenario for this use case is the following: It's Monday 9:30 am GMT and we have several cloud computing units running in our clusters in America, Europe and Asia. At this particular time there is a division of cloud computing units in the following ratio: 3 active units in our cloud cluster in Europe, 2 in Asia and 1 in America. (It is currently night in America and thus little computing power is needed in that particular cluster. Also see the aforementioned Figures 1, 2 and 3 about Internet activity. The speed layer contains information of data units about the last half hour: 9 - 9:30 am. This layer holds information up to 1 hour. The batch layer contains information of the last year up to 9 am today.

All computing units in all clusters (America, Europe and Asia) generate data about the usage of their own units. In this scenario this is information about the load on such a computing unit, CPU usage, RAM usage, megabytes of up- and download per second. Data from units in Europe, Asia and America is given an identifier that belongs to that particular unit such that we know which data came from which cluster later on when we want to perform analysis on this data. All data is then sent into the feedback loop as depicted in figure 4. Cloud computing units send data into the feedback loop every few seconds. The units do not have to sync up before sending data. This is because all usage data from the clusters is put in queues. In turn, this data is consumed by storing it in the master database (batch layer) and redirecting it to the stream processor (speed layer). To keep this example simple, only averages of CPU are taken. In reality, many more parameters are taken into account depending on the kind of data that the smart scaler requests.

On the streaming side of the Lambda Architecture the raw data is processed through the stream processor. A smaller amount of analyzed data is the result of this process. Data that stems from the same computing cluster is joined and averages (CPU average usage, RAM average usage...) are taken. For our example this means we receive 3 data units from Europe, 2 from Asia and 1 from America about CPU usage. From Europe we receive 85%, 90% and 95% CPU usage. The stream processor calculates an average of this and puts out 90%. In Asia we receive 50% and 70% as input which the stream processor averages to 60%. From the American computing unit we receive a CPU usage of 70%. Logically, the stream processor outputs 70% as average. These averages are then stored in the query layer to be used in queries later on. Averages from the last half an hour that stem from the speed layer are stored in the same place. The duration of storing this information in the speed layer is equal to the time it takes the batch layer to do one round of computations which is one hour in this case. As soon as data is available for queries on the batch layer side, there is no need to keep it on the speed layer side. So at 10 am the information of 9 am till 1 to 10 am is not available anymore from the speed layer, but it is processed by the batch layer by then and thus still available for querying.

All data that is sent to the batch layer is stored in a database at first.

The exact same information as in the speed layer about the CPU usage statistics from all 3 clusters is appended to the master database. This is still in the same format as the way the computing units have provided this data. Similar to the speed layer, data that stems from the same cluster is joined in the batch processor through MapReduce jobs that have a notably smaller amount of data as output than their respective input. MapReduce jobs are executed as soon as the previous one has finished. This ensures that queries are executed on the newest data available. Again, averages are calculated in this use case and made available for querying. A batch job in this scenario takes CPU usage information from all Mondays from the last year till 9 am today and creates average for every cluster of every hour of every Monday.

In the query layer we now have both historical and real-time data available. We use a small part of this data that belongs to a cluster in America, Europe or Asia to satisfy a certain query. For our use case this means we take the previously computed averages from (9 am to 9:30 am) and from the historical data we take the gathered data on all Mondays right from the start until 9 am on this Monday. This data then gives an insight into the amount of computing units we need now (mainly based on real-time data) and the amount we likely need in the future (mainly based on the historical data) for this cluster. In this scenario we can see from recent data that the cluster in Europe is under pressure (90% CPU load). Furthermore, historic data provides us with the information that this cluster is under pressure (average of 92% CPU load) every Monday around 9:30 am.

The smart scaler then uses the information from the queries to determine two issues. The first one stems from the recent data, which is that the cluster in Europe is currently under a load that is deemed to be too high since it results in a too high latency for the end users. The second issue is obtained from the historic data, which shows that it is very likely that the cluster in Europe will still maintain a high load for the coming hours. In case we only had only detected the first issue, the smart scaler would have tried to temporary off load customers to the other clusters located in America and Asia. Taking the second issue in account, the smart scaler knows that it is feasible to add more computing units to the cluster in Europe since this high CPU load will likely last for hours.

Instructions are then passed to the executing scaler to add computing units to the cluster in Europe. As soon as the new computing units are added to the cluster these new computing units also start emitting raw data that can be used for future adjustments. This completes one cycle of the feedback loop.

4

IMPLEMENTATION

The following chapter describes the implementation of the software of this thesis. Where the previous chapter focused on the theory of this thesis, the current chapter shows how the Lambda Architecture can be used in practice for latency reduction in cloud scaling. Each subsection provides an insight into what techniques and software is used at the different components of the project. In various areas the implementation has been simplified. This is legitimized because the main goal of the implementation is to show the feasibility of the design discussed in the previous chapter.

4.1 GATHERING METRICS

Metrics are gathered from computing units, which are analyzed later on within the Lambda Architecture. The computing units from which metrics are gathered, are the ones that handle the client requests (e.g. uploading a photo in our aforementioned use case). To implement this, computing units should be deployed on several cloud providers such as Amazon Web Services (AWS). For feasibility purposes smaller set ups have been used: a ten computing units cluster and a single machine set up for obtaining the results in the next chapter.

In the implementation of this project only CPU data is gathered. Depending on the needs of the scaler other metrics could be gathered as well. Furthermore, the internet traffic patterns shown in the previous chapter in figures 1, 2 and 3 are used. A single process is used for sending data about how busy the different virtual clusters (America/Asia/Europe) are, to ensure that the way metrics are gathered is as realistic as possible. The essential part of the program is shown in listing 1. It can be noted that the Java program only sends one message every ten seconds and should thus create negligible load on the computing units handling the client requests. Furthermore, messages are kept small in size. A message consists of only one line containing the id, CPU load (as calculated in 1), location, provider and a time stamp for a computing unit at a certain moment in time.

Listing 1: Snippet of Java file to gather metric data based on web traffic

```
1  private float calcActivityDay(String location) {
2    TreeMap<Float, Float> activityDuringDay = new TreeMap<Float,
         Float>();
3    activityDuringDay = readActivityFile(location);
4    long MILLIS_PER_DAY = 24 * 60 * 60 * 1000;
5    Date now = Calendar.getInstance().getTime();
6    float timeNow = now.getTime() % MILLIS_PER_DAY;
7    timeNow = timeNow / 60 / 60 / 1000; // to hours
8    Set s = activityDuringDay.entrySet();
9    Iterator it = s.iterator();
10   Map.Entry<Float, Float> currentEntry = null, previousEntry =
         null;
11   while ( it.hasNext() ) {
12     currentEntry = (Map.Entry<Float, Float>) it.next();
13     float timeEntry = currentEntry.getKey();
14     if (timeEntry > timeNow) break;
15     previousEntry = currentEntry;
16   }
17   float multiplier = (timeNow - previousEntry.getKey())/(
         currentEntry.getKey() - previousEntry.getKey());
18   float activity = currentEntry.getValue()*multiplier +
         previousEntry.getValue()*(1-multiplier);
19   return activity;
20 }
```

Listing 2: Snippet of Java file to store metric data in master database

```
TextMessage textMessage = (TextMessage) message;
String text = textMessage.getText();
text = text.replace("\n", "");
Path filenamePath = new Path(Main.hdfsLocation + "hdfsdata/" +
    System.currentTimeMillis());
Configuration conf = new Configuration();
conf.addResource(new Path("/HADOOP_HOME/conf/core-site.xml"));
conf.addResource(new Path("/HADOOP_HOME/conf/hdfs-site.xml"));
FileSystem fs = FileSystem.get(conf);
FSDataOutputStream fin = fs.create(filenamePath);
fin.writeUTF(text);
fin.close();
```

4.2 PROCESSING

Data processing starts with consuming messages from a queue. This queue is populated by the metrics gathered as mentioned in previous section. The messages are parsed and then the data within these messages are sent to the Lambda Architecture.

4.2.1 *Consuming data*

First, the data is sent to the master database. This is done by a single thread, which is sufficiently fast for our implementation. Multiple threads could be used when scale size increases, but most other components of the Lambda Architecture require significantly more extra computing power when scaling. Scalability is thus not an issue specifically for this component. The functioning of this thread remains the same: it consumes messages and opens an output stream to write the data to the Hadoop Distributed File System (HDFS)[41], which is the master database. HDFS takes care of replication and having the data available on multiple nodes to be used with the batch processor later on. A code snippet of the essential part is shown in listing 2. Secondly, data is sent to the stream processor. Similar to the first stream, a single thread is started to consume messages. This consumer thread is used as input for the streaming processor.

Listing 3: Java snippet of batch processing using Spark

```
private static JavaPairRDD<String, Float> getCpuLoadData(JavaRDD<CPU> parsedData) {
    return parsedData.mapToPair(new PairFunction<CPU, String, Float>() {
  @Override
  public Tuple2<String, Float> call(CPU cpu) throws Exception {
      Date date = new Date(cpu.getTimeStamp());
      return new Tuple2<String, Float>(cpu.getLocation() + "/" + date.getMinutes(), cpu.
          getCpuLoad());
  }
    });
}
```

Listing 4: Java snippet of stream processing using Spark

```
private static JavaPairDStream<String, Float> getStreamCpuLoadData(JavaDStream<CPU>
    parsedData) {
    return parsedData.mapToPair(new PairFunction<CPU, String, Float>() {
  @Override
  public Tuple2<String, Float> call(CPU cpu) throws Exception {
      Date date = new Date(cpu.getTimeStamp());
      return new Tuple2<String, Float>(cpu.getLocation() + "/" + date.getMinutes(), cpu.
          getCpuLoad());
  }
    });
}
```

4.2.2 *Batch and stream processing*

One of the big drawbacks of the Lambda Architecture is its inherent complexity. Typically two separate code bases have to be maintained, one for the streaming layer and one for the batch layer. To alleviate this burden, Apache Spark[42] is used. Spark is typically used for batch processing. It also has a streaming module however. The code of the batch and streaming module are thus essentially the same except for some minor differences allowing you to essentially have only one code base. It is thus desirable to use Spark or another program that can handle both stream- and batch processing to decrease complexity. Listings 3 and 4 shows how code used for batch processing can be re-used for stream processing with little effort.

Listing 5: Pseudo-code of rules used by the Lambda Architecture scaler

```
if recentLoad > MAXIMUM_RECENT_LOAD and historicLoad >
    MAXIMUM_HISTORIC_LOAD then
    addComputingUnits(cluster)
else if recentLoad > MAXIMUM_RECENT_LOAD then
    relocateLoadToOtherClusters(cluster)
else if recentLoad < MINIMUM_RECENT_LOAD && historicLoad <
    MINIMUM_HISTORIC_LOAD then
    removeNewComputingUnits(cluster)
else if (recentLoad < MINIMUM_RECENT_LOAD then
    relocateLoadFromOtherClusters(cluster)
```

4.2.3 *Querying*

Data needs to be available for querying. As discussed in the design section[3] there is no need for a key-value store, yet we need to make this data easily accessible. This is done by writing the output of the Spark processor back into HDFS files. Due to the distributed nature of both Apache Spark and HDFS it is possible to write data from the processors to HDFS in parallel, ensuring that there is no bottleneck problem when attempting to do multiple writes at once during this process.

4.3 RULES AND CONSTRAINTS

The scaler uses a minimal set of rules and constraints that still show the functionality and benefits of using the Lambda Architecture in cloud scaling over any conventional cloud scaler. The essence of this set is shown in listing 5. A typical conventional cloud scaler would only use the rules specified in listing 6. In chapter 5 both sets are also used to draw comparisons between conventional scaling and scaling with Lambda Architecture. This section however only gives a quick overview of the functionality of these sets.

The LA scaler checks these four rules every ten seconds. When the current load of a certain cluster exceeds either the upper or lower threshold, actions are taken. The thresholds are set to 70 and 90 percent CPU load respectively in this case. The same rules also applies to

Listing 6: Pseudo-code of rules used by a conventional scaler

```
1 if recentLoad > maximum recent load then
2     addNewComputingUnits(cluster)
3 else if recentLoad < minimum recent load then
4     removeComputingUnits(cluster)
```

a traditional scaler. The novelty of the LA scaler lies in the other two rules. When both the current and historic load exceed certain thresholds, some smart actions are taken which cut down the necessary amount of computing units.

4.4 SCALING

The previous section discussed the rules and constraints for when to take action. This section offers an insight into the implemented actions. The number of possible different actions have been reduced to four: removing computing units, adding computing units, relocating load from and relocating load to a cluster.

The first two are straightforward and implemented by any regular cloud scaler, as seen in listing 6. Whenever there is high CPU load on the computing units within a certain cluster, add extra computing units to that cluster. Whenever there is low CPU load, do the exact opposite and remove computing units.

Similar to the previous two actions, relocating load from other clusters and relocating load to other clusters also function in exactly the opposite way. Applying both actions to the same cluster would make them cancel each other out. Listing 7 shows the bare essentials of the 'locating load' action. These actions shows a minimum working example of the follow-the-sun use case discussed in 3.1. When the function *relocateUnits* is called 1000 units of CPU load are relocated from one cluster to another.

Listing 7: Pseudo-code of moving CPU load between clusters

```
relocateUnits(scalingInstructions)
    fromCluster, toCluster = parse(scalingInstructions)
    relocate(fromCluster, -1000);
    relocate(Tocluster, 1000);
```

5

EVALUATION

Various simulations have been ran on the implementation discussed in previous chapter. Besides using the Lambda Architecture for automatic scaling, simulations have also been ran using solely stream processing and solely batch processing. This is done because the most important goal is to find and confirm areas where Lambda Architecture scaling can outperform traditional scaling (stream processing). Different Internet traffic profiles have been used to test the performance in different situations. In this case performance is measured by keeping the cpu load under certain levels. Each traffic profile has been tested for a few days. During this time information has been gathered about the number of machines and other resources to keep performance at a certain level. In this chapter we compare the results obtained by the different kinds of automatic cloud scaling.

The amount of different scenarios for cloud scaling is colossal. It is therefore not feasible to treat them all in this thesis. Instead, two datasets are picked which aid in answering the research questions posed. In this thesis batch processing, stream processing and a combination of both are discussed for automated cloud scaling. The two datasets have thus been selected in such a way that they clearly show the differences between these types of techniques. Although these datasets have an artificial nature they do help to prove the point of using Lambda Architecture in this thesis. Converting this to use natural datasets (in other words for example having real clients visit your website or having tasks taking up computing power) has been discussed extensively in chapter 3, with the follow-the-sun use-case in 3.1 as example, and should thus be possible.

5.1 BENCHMARKS

Two different datasets of Internet traffic simulation have been used. These two sets clearly show where the advantages lie of the Lambda Architecture. Both sets contain data about traffic at a certain moment in time. Data is normalized to produce benchmarks that can be compared with other methods or datasets. Both sets show an increase

in traffic during day and a decrease during night allowing us to utilize the Lambda Architecture for the follow-the-sun principle. Furthermore, it can be seen that the peaks and troughs of the output results align with the input traffic datasets.

5.1.1 *Datasets*

The first set is the one mentioned in the follow-the-sun use case section 3.1 using information from three Internet Exchanges (see Figures [28][29][30]). This data set has been selected because it comes very close to how the Lambda Architecture could be used in business cases.

To turn each image into usable data, a tool has been used to convert timestamps and corresponding traffic load into a csv file. The csv file contains about 50 entries for a period of 24 hours. An interpolation is taken from the two closest entries at each moment in time during the simulation. Traffic load thus scales up and down gradually.

The first data set is not producing a clear difference in the results. The second data set is therefore chosen to show clear differences between traditional on-demand cloud scaling and both historical and recent Lambda Architecture scaling. The second set makes use of http logs of a busy ISP www server in the Washington DC area[43]. These logs span a period of two weeks. To be useful for the written implementation in this thesis data has been averaged to one day of data and normalized. For this set each entry in the produced csv file contains information for a period of ten seconds. Contrary to the first set no interpolation is applied when this set is used. Traffic load can thus contain many spikes.

5.1.2 *Dataset results*

The most important criterion for our implementation is the number of machines used. Information has thus been collected about this when the previously mentioned datasets are used as input. Figures 6, 7 and 8 shows benchmarks obtained by using the first dataset.

It can be observed that all clusters nicely adhere to the follow-the-sun-principle, showing highs during day and lows during night. Further-

more, it can be seen in the figures that all techniques obtain similar results for the first dataset. This is caused by two characteristics of this dataset; The first characteristic is that the dataset only provides data of one day. This means that all historic data perfectly predicts data traffic for the coming day since it will follow the same pattern as previous days. The disadvantages of pure batch processing are thus negated. If there were indeed any fluctuations in data usage between days, batch processing would not be able to take this into account. Furthermore, this dataset contains averaged and interpolated data. Which means that data traffic increases and decreases gradually. Typically stream processing is hindered by the fact that starting up and shutting down computing units takes time. However, since data increases and decreases so gradually, booting time of computing units poses no significant disadvantage to stream processing. Furthermore, Lambda Architecture cloud scaling cannot make use of shortly offloading users to other clusters in this scenario. A very gradual stream of Internet traffic thus poses no opportunities for improvement for the Lambda Architecture.

The second dataset contains a spiky stream of Internet traffic, meaning Internet traffic fluctuates strongly (every minute in this case). The results are shown in figures 9, 10 and 11. Contrary to the first dataset, this one shows some differences between the different techniques used. Due to the reactive nature of stream processing, it tends to add more computing power than strictly necessary. Stream processing tries to compensate for the high peaks in Internet traffic by adding more computing units. Batch processing and the Lambda Architecture on the other hand utilize the historic data at their disposal. These two techniques can see if the sudden increase in traffic is long-lasting.

All clusters in figures 9, 10 and 11 clearly illustrate that computing power usage has lower peaks when the Lambda Architecture is used than when stream processing is utilized. This is due to users being offloaded to other clusters, which also explains that the Lambda Architecture graphs show higher minimums than stream processing. It can thus be concluded that the Lambda Architecture possesses opportunities when traffic/computing power is fluctuating strongly.

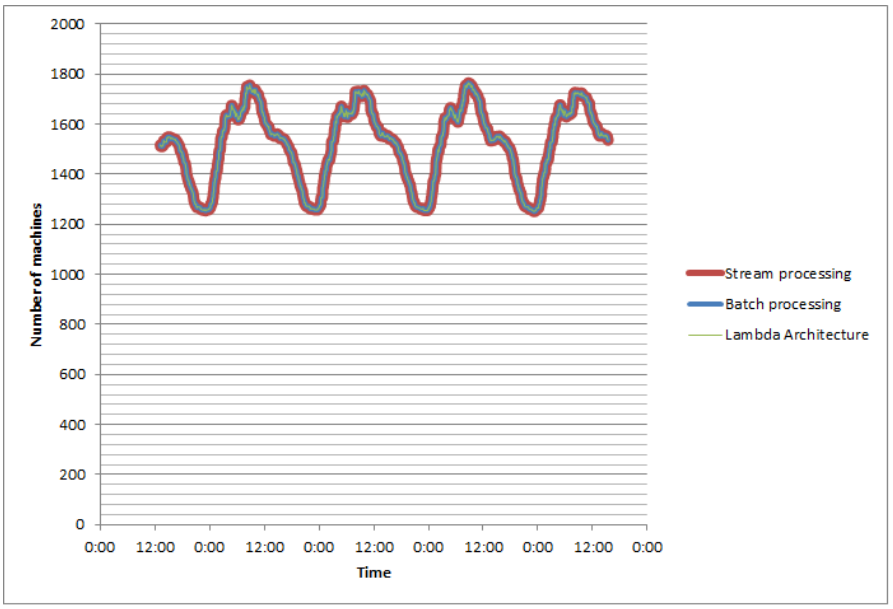

Figure 6: Comparison of number of machines used when using the first smooth data set. Cluster: Asia

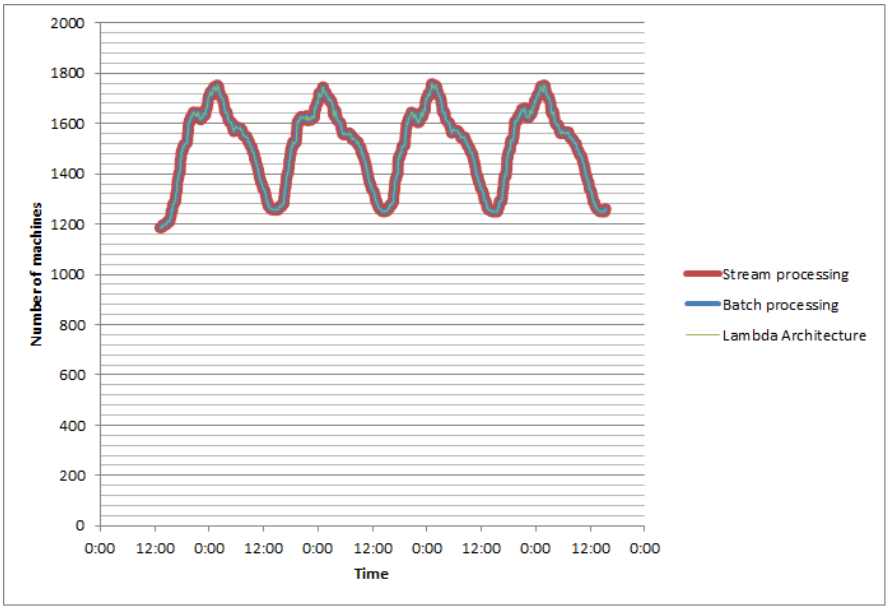

Figure 7: Comparison of number of machines used when using the first smooth data set. Cluster: Europe

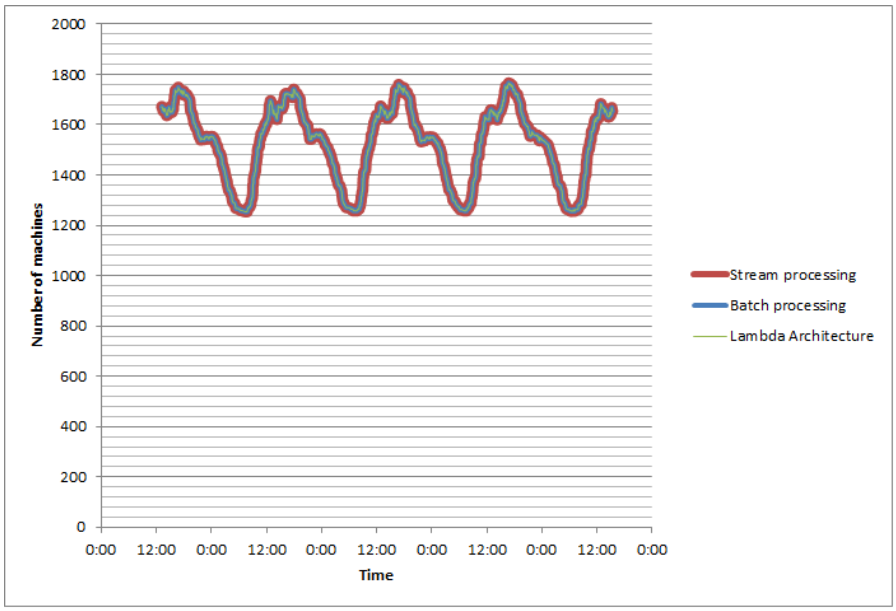

Figure 8: Comparison of number of machines used when using the first smooth data set. Cluster: America

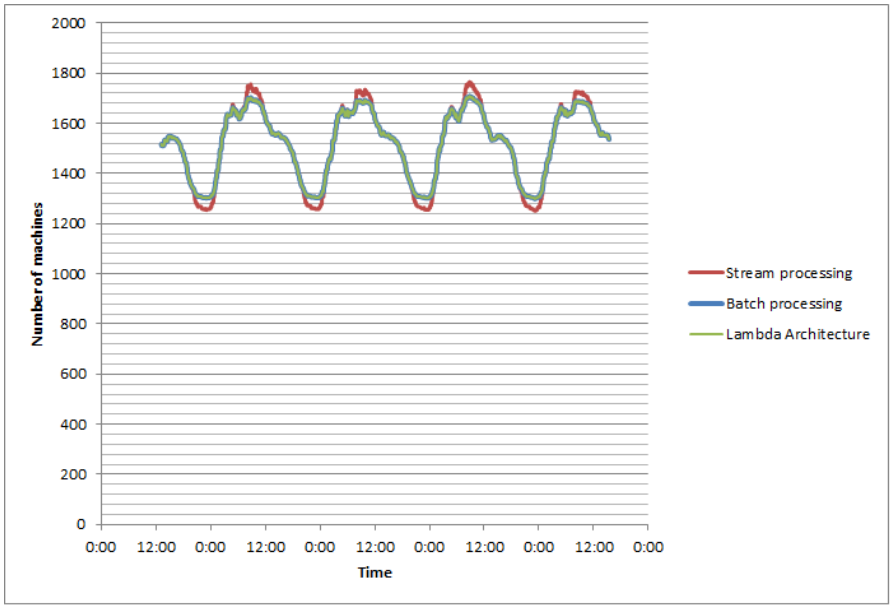

Figure 9: Comparison of number of machines used when using the second spiky data set. Cluster: Asia

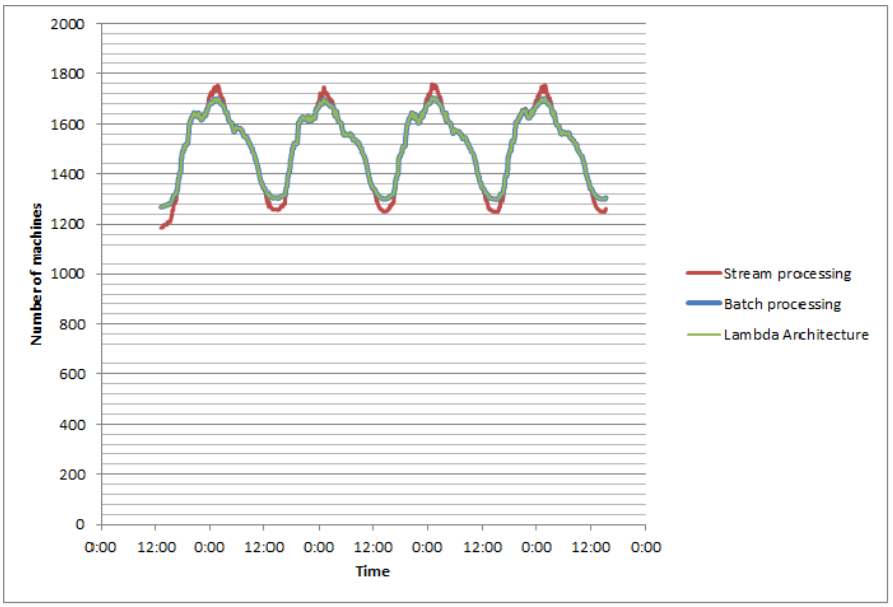

Figure 10: Comparison of number of machines used when using the second spiky data set. Cluster: Europe

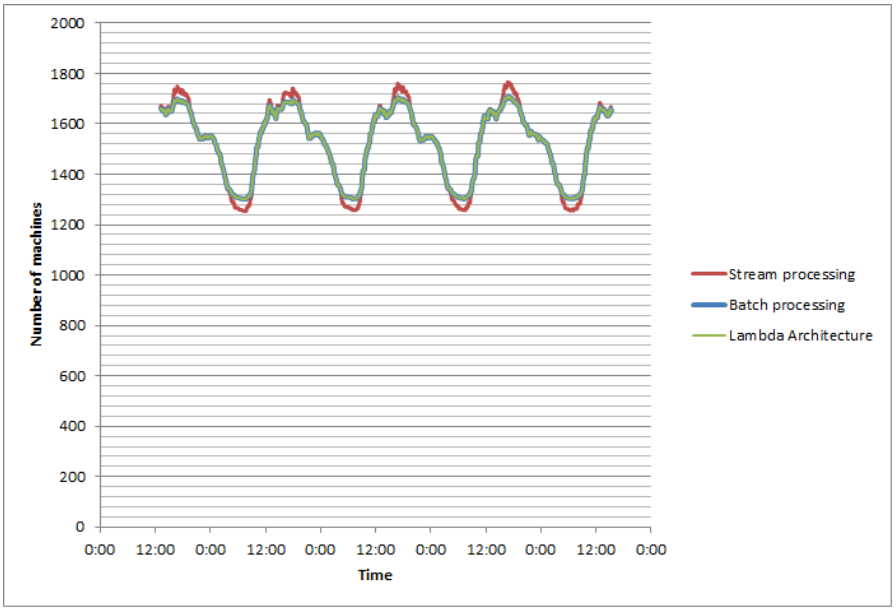

Figure 11: Comparison of number of machines used when using the second spiky data set. Cluster: America

5.2 PROCESSING EVALUATION

This section discusses the advantages and disadvantages of using stream processing, batch processing and the Lambda Architecture for automatic cloud scaling. The used data sets demonstrate some of the more interesting similarities and differences between the aforementioned techniques. Not all cases are covered by the datasets however. Further in this section other evaluated opportunities are discussed.

5.2.1 *Batch processing*

Traditional cloud scaler typically do not use any kind of historical data/batch processing. Amazon's AWS and Google's cloud platform for example allow you to set policies which only involve recent data[17][5]. Batch processing has produced good results however in the datasets discussed in previous section. One would thus expect that batch processing is a good option for the major cloud providers offering automatic scaling of their clouds. Using only historic data renders a considerable flaw however; Recent data about computing traffic may deviate strongly from historic data. Considering the use case in 3.1, there may be a big event in Europe, for example the European football Championship. This can cause sudden increases of traffic for football sites during a match. Since only historic data is available, there will be no action to adjust the computing units in Europe. Furthermore, the clusters in America and Asia will not be notified of this increase in load.

The final result is that the computing units in Europe will be severely overloaded, resulting in high latency and possible failures. The Lambda Architecture with its combination of stream and batch processing would have offloaded the many users in Europe to other clusters and possibly added extra computing units to deal with such an event.

5.2.2 *Stream processing*

The performance of stream processing in the tested datasets deviated from the other two techniques only marginally. Furthermore, stream processing does not require any processing of historic data and can scale your cloud on the fly. This makes it thus the technique of choice for most cloud scalers[17][5]. It does however have some financial

downsides. The provided automatic scalers are prone to immediately add machines whenever a certain threshold is reached (e.g. a 90% CPU load on the computing units). Computing units can be removed as soon as values have dropped below this certain threshold once again. But cloud providers typically charge for a full hour of usage, even though the cloud scaler may have only needed that extra computing power for just two minutes. Concluding, in the tests stream processing does not look like a bad choice, but starting and removing a few extra computing units strongly increases costs.

5.3 SCOPE OF IMPLEMENTATION AND DATASETS

This section disccuses the scope of the implementation and datasets with respect to the discussed design. The tested datasets only cover a small portion of the scenarios one may come across in automatic cloud scaling. The main goal was to demonstrate the feasibility of using the Lambda Architecture for automatic cloud scaling with the given implementation. To stay close to the core of the search, there are several areas which the datasets did not touch. Yet these areas are treated in the design section 3.

One of the simplifications of the implementation is the fact that all computing is done at the same location. Artificial delays have been added to simulate latency between users and clusters that are geographically far away from each other. An example of this is a user in the US that may be connected to a cluster in Europe because the American cluster is currently too busy. A hardcoded artificial delay is then added. In reality, this delay will fluctuate. The hardcoded delay does still demonstrate however that a penalty is incurred whenever a user is connected to a cluster that is far away.

Another factor is that both datasets use repetitive data. That is, Internet traffic load does not change from week to week but remains actually the same. Repetitive data is in this case useful as it demonstrates the possibilities of using batch and combined processing instead of traditional stream processing. The repetitive data yields (nearly) the exact same results for every day/week that a simulation is ran while fluctuations may occur in real-life situations. This is also demonstrated in the figures in this chapter. E.g. figures 6, 7 and 8 show a few days of data with a repetition of the same load and computing power ev-

ery twenty-four hours.

Furthermore, client requests are simulated. If client requests had to come from actual machines, then 10,000+ machines are necessary to come close to producing the desired behaviour and amount of requests. This issue can be further generalized to the fact that the implementation has been done in a research environment and not in a production environment. Although this can be seen as a limitation that comes natural to research in cloud computing.

6

FUTURE WORK

Utilizing the Lambda Architecture for automatic cloud scaling is a novel approach. As such, there is an abundance of research that can be done on the subject. This section discusses significant ones that build upon this thesis.

The usage of the Lambda Architecture in this thesis is mainly restricted to scaling with different clusters in different locations (see section 3.1). It may also prove to be useful when computations are done within a single cluster. For example, when traffic/computations are fluctuating extremely fast it may be beneficial to have historic data available to know that adding or removing machines in such scenarios is senseless.

In this work only one simplified metric is used to set the policy for automated scaling. That is, CPU load on computing units. Cloud providers typically allow for more advanced configurations. AWS for example even allows the user to set a policy[5]. An extension would thus be to investigate which other metrics are suitable. Furthermore, traditional cloud scaling may require certain metrics to efficiently scale up or scale down. There could be other ones that prove specifically effective for scaling with the Lambda Architecture.

Another addition, which to some extent builds on the previous suggestion, is to take external factors into account. In this thesis a feedback loop is used. The feedbackloop used in this thesis only takes information from the users and computing units into account. It may prove useful to add in external factors such as energy pricing. This could influence the price of renting computing power and in turn influence where and when requesting extra machines is sensible.

Some parts of the implementation have been simplified for practical reasons. While this thesis demonstrates that the Lambda Architecture can be used for cloud scaling, it cannot make a complete comparison to traditional cloud scaling. The traditional cloud scaling, which is using only stream processing, has also been dumbed down in this

thesis. It would thus be advantageous to see how the Lambda Architecture performs when it is given the same scaling policy that cloud providers give their own automatic scalers.

7 CONCLUSION

In this thesis an evaluation of the usage of the Lambda Architecture for automatic cloud scaling is provided. First an overview of the related work has been given. Then the concept and theory behind this thesis is discussed. An implementation and simulations are employed to substantiate this evaluation. This eventually answers the original research question: *Can we improve automatic scaling to efficiently reduce latency using the Lambda Architecture?* To answer this question, the three sub research questions have to be answered first.

- Does simultaneously using processing and stream processing generate significant overhead?
- How does combined processing compare to only batch processing?
- How does combined processing compare to only stream processing?

7.1 DOES USING BOTH BATCH PROCESSING AND STREAM PROCESSING AT THE SAME TIME GENERATE SIGNIFICANT OVERHEAD?

Chapter 3 describes that it is possible to use both batch processing and stream processing for automatic cloud scaling. The implementation as discussed in chapter 4 and the results in chapter 5 show that it is indeed feasible. Furthermore, table 1 shows the amount of data that flows through each part of the Lambda Architecture. Having to process all this data does not lead to a significant increase in number of computing units. Also, collecting the data does not put substantial pressure on the computing units from which this data is collected. It can thus be concluded that combining batch processing and stream processing generates only minimal overhead.

7.2 HOW DOES COMBINED PROCESSING COMPARE TO ONLY BATCH PROCESSING?

For the given data sets in chapter 5 batch processing and combined processing yield the same results. In other scenarios combined processing is likely to severely outperform batch processing due to batch processing's lack of on-demand scaling. However, given only repetitive data such as in previously mentioned data sets, batch processing would be the preferred option. Combined processing in that scenario only adds overhead and a quite complicated code base to maintain. In general, combined processing is thus preferred.

7.3 HOW DOES COMBINED PROCESSING COMPARE TO ONLY STREAM PROCESSING?

Stream processing has been the default way of automatically scaling the cloud. It is easier to use than combined processing due to the fact that only one codebase needs to be maintained, and not two separate ones as in the case of the Lambda Architecture. It can be seen in the data sets in chapter 5 however that combined processing outperforms stream processing in these particular cases. Combined processing thus is preferable whenever historic patterns can be found in the given data. In cases where data is entirely unpredictable, stream processing is the better option since it is easier to control and requires less data processing.

7.4 MAIN RESEARCH QUESTION

We have investigated the use of the Lambda Architecture for automatic cloud scaling in this thesis. Given the conclusions on the three sub research questions, we can now answer the main research question.

In this thesis we have shown that the Lambda Architecture can indeed improve automatic scaling and efficiently reduce latency. The scenario for which we have proven this is the use-case specified in section 3.1. Although it increases complexity of the code base, the Lambda Architecture does not decrease performance significantly. Furthermore, it is shown that the amount of computing power is reduced while showing no increase in latency for its users. Hopefully this work aids

others to find other areas and scenarios in which the Lambda Architecture also proves its significance.

Part I

APPENDIX

BIBLIOGRAPHY

[1] Aaron Weiss. Computing in the clouds. *netWorker*, 11(4):16–25, December 2007. ISSN 1091-3556. doi: 10.1145/1327512.1327513. URL http://doi.acm.org/10.1145/1327512.1327513. (Cited on page 1.)

[2] General information aws. URL https://aws.amazon.com/about-aws/. Last access: Oct. 6 2015. (Cited on page 1.)

[3] General information azure. URL https://azure.microsoft.com/nl-nl/. Last access: Oct. 6 2015. (Cited on page 1.)

[4] General information google cloud platform, . URL https://cloud.google.com/. Last access: Oct. 6 2015. (Cited on page 1.)

[5] Amazon autoscaler. URL http://aws.amazon.com/autoscaling/. Last access: Feb. 16 2015. (Cited on pages 2, 9, 22, 41, and 45.)

[6] Nathan Marz and James Warren. *Big Data: Principles and best practices of scalable realtime data systems*. Addison-Wesley Professional, March 2015. ISBN 9781617290343. (Cited on page 2.)

[7] D. Dahiphale, R. Karve, A.V. Vasilakos, Huan Liu, Zhiwei Yu, A. Chhajer, Jianmin Wang, and Chaokun Wang. An advanced mapreduce: Cloud mapreduce, enhancements and applications. *Network and Service Management, IEEE Transactions on*, 11(1):101–115, March 2014. ISSN 1932-4537. doi: 10.1109/TNSM.2014.031714.130407. (Cited on page 7.)

[8] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, 2010. (Cited on page 8.)

[9] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online aggregation. *SIGMOD Rec.*, 26(2):171–182, June 1997. ISSN 0163-5808. doi: 10.1145/253262.253291. URL http://doi.acm.org/10.1145/253262.253291. (Cited on page 8.)

[10] J. Urbani, A. Margara, C. Jacobs, S. Voulgaris, and H. Bal. Ajira: A lightweight distributed middleware for mapreduce and

stream processing. In *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*, pages 545–554, June 2014. doi: 10.1109/ICDCS.2014.62. (Cited on page 8.)

[11] Summingbird. URL https://twitter.com/summingbird/. Last access: Feb. 7 2015. (Cited on page 8.)

[12] Twitter storm. URL https://storm.apache.org/. Last access: Feb. 7 2015. (Cited on page 8.)

[13] Scalding. URL http://www.cascading.org/projects/scalding/. Last access: Feb. 7 2015. (Cited on page 8.)

[14] Jiang Dejun, Guillaume Pierre, and Chi-Hung Chi. Resource provisioning of Web applications in heterogeneous clouds. In *Proceedings of the 2nd USENIX Conference on Web Application Development*, June 2011. (Cited on page 8.)

[15] A. Gandhi, P. Dube, A. Karve, A. Kochut, and Li Zhang. Modeling the impact of workload on cloud resource scaling. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2014 IEEE 26th International Symposium on*, pages 310–317, Oct 2014. doi: 10.1109/SBAC-PAD.2014.16. (Cited on page 9.)

[16] D. Ardagna, E. di Nitto, P. Mohagheghi, S. Mosser, C. Ballagny, F. D'Andria, G. Casale, P. Matthews, C.-S. Nechifor, D. Petcu, A. Gericke, and C. Sheridan. Modaclouds: A model-driven approach for the design and execution of applications on multiple clouds. In *Modeling in Software Engineering (MISE), 2012 ICSE Workshop on*, pages 50–56, June 2012. doi: 10.1109/MISE.2012.6226014. (Cited on page 9.)

[17] Google compute autoscaler, . URL https://cloud.google.com/compute/docs/autoscaler/. Last access: Feb. 16 2015. (Cited on pages 9, 22, and 41.)

[18] Intercloud. URL https://www.intercloud.com/. Last access: Feb. 7 2015. (Cited on page 9.)

[19] Rodrigo N. Calheiros, Adel Nadjaran Toosi, Christian Vecchiola, and Rajkumar Buyya. A coordinator for scaling elastic applications across multiple clouds. *Future Generation Computer Systems*, 28(8):1350 – 1362, 2012. ISSN 0167-739X. doi: http://dx.doi.org/10.1016/j.future.2012.03.010. URL http://www.sciencedirect.

com/science/article/pii/S0167739X12000635. Including Special sections SS: Trusting Software Behavior and SS: Economics of Computing Services. (Cited on page 9.)

[20] Rightscale. URL http://www.rightscale.com/about. Last access: Jan. 18 2015. (Cited on page 9.)

[21] Amazon web services, . URL http://aws.amazon.com/. Last access: Feb. 7 2015. (Cited on page 9.)

[22] Openstack. URL https://www.openstack.org/. Last access: Feb. 7 2015. (Cited on page 9.)

[23] collectd. URL https://collectd.org/. Last access: Feb. 16 2015. (Cited on page 10.)

[24] elkstack. URL http://www.elasticsearch.org/webinars/elk-stack-devops-environment/. Last access: Feb. 7 2015. (Cited on page 10.)

[25] Aws geographical regions, . URL http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html. Last access: Feb. 19 2015. (Cited on page 11.)

[26] Rackspace geographical regions, . URL http://www.rackspace.com/about/datacenters. Last access: Feb. 19 2015. (Cited on page 11.)

[27] Gogrid geographical regions, . URL https://wiki.gogrid.com/index.php/Geographic_Load_Balancing. Last access: Feb. 19 2015. (Cited on page 12.)

[28] Amsterdam internet exchange. URL https://ams-ix.net/technical/statistics. Last access: Feb. 23 2015. (Cited on pages 12 and 36.)

[29] New york internet exchange. URL http://www.nyiix.net/mrtg/sum.html. Last access: Feb. 23 2015. (Cited on pages 12 and 36.)

[30] Tokyo internet exchange. URL http://www.jpnap.net/english/jpnap-tokyo-ii/traffic.html. Last access: Feb. 23 2015. (Cited on pages 12 and 36.)

[31] Aws services, . URL http://aws.amazon.com/documentation/. Last access: May. 12 2015. (Cited on page 14.)

[32] Betty H. C. Cheng, Rogerio de Lemos, David Garlan, Holger Giese, Marin Litoiu, Jeff Magee, Hausi A. Muller, and Richard Taylor. Seams 2009: Software engineering for adaptive and self-managing systems. In *Proceedings of the 2009 31st International Conference on Software Engineering: Companion Volume*, ICSE '09 COMPANION, pages 463–464, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3495-4. doi: 10.1109/ICSE-COMPANION.2009.5071063. URL http://dx.doi.org/10.1109/ICSE-COMPANION.2009.5071063. (Cited on page 14.)

[33] Aws billing, . URL http://docs.aws.amazon.com/AWSEC2/latest/CommandLineReference/ApiReference-cmd-StartInstances.html. Last access: May. 4 2015. (Cited on page 18.)

[34] Gogrid billing, . URL https://wiki.gogrid.com/index.php/Billing_Model. Last access: May. 4 2015. (Cited on page 18.)

[35] Rackspace billing, . URL http://www.rackspace.co.uk/calculator. Last access: May. 4 2015. (Cited on page 18.)

[36] Aws time to boot, . URL http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ComponentsAMIs.html. Last access: May. 4 2015. (Cited on page 18.)

[37] Gogrid time to boot, . URL https://wiki.gogrid.com/images/c/cd/GoGrid_DB2_9.7.4_GSI_Start-up_Guide_20111024.pdf. Last access: May. 4 2015. (Cited on page 18.)

[38] Imgur infrastructure. URL http://imgur.com/blog/2013/06/04/tech-tuesday-our-technology-stack/. Last access: May. 7 2015. (Cited on page 18.)

[39] S. Srinivasan. Cloud computing basics. 2014. (Cited on page 18.)

[40] Randy Bias. Netflix infrastructure. URL http://www.cloudscaling.com/blog/cloud-computing/cloud-innovators-netflix-strategy-reflects-google-philosophy/. Last access: May. 7 2015. (Cited on page 18.)

[41] Hadoop distributed file system. URL http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html. Last access: Jul. 27 2015. (Cited on page 29.)

[42] Apache spark. URL https://spark.apache.org/. Last access: Jul. 27 2015. (Cited on page 30.)

[43] Internet archive clarknet traffic. URL http://ita.ee.lbl.gov/html/contrib/ClarkNet-HTTP.html. Last access: Sept. 27 2015. (Cited on page 36.)

DECLARATION

I hereby declare that this thesis is my own work and effort and that it is not submitted anywhere else for any award. Usage of external sources is acknowledged.

Groningen, October 2015

Hessel van Apeldoorn