Job containerization and orchestration
to reduce TTC and operational costs

Jorick Spitzen, Simon Barendse

March 23, 2019

Abstract

Predictive analytics has become a common practice among retailers. Veneficus provides insights and assists clients in making decisions based on facts. The analyses required for this are often computationally intensive and take a lot of time to run. In order to provide a scalable, efficient and cost effective method for performing these analyses, the company must employ the latest techniques in cloud computing like Docker and Kubernetes. This project shows the process of setting up a autoscaling compute cluster, as well as an interface that data scientists with little to no knowledge of Kubernetes or other orchestration frameworks can use. By using the scale of a cloud based cluster and Greedy scheduling algorithms, we show that it is possible to perform large scale analytics in a way that Time to Completion is constant in the number of jobs being executed.
## Contents

1 Introduction  
   - 1.1 Introduction 4

2 The problem  
   - 2.1 The client 5
   - 2.2 The project 5
   - 2.3 The analysis 5

3 Objectives  
   - 3.1 Time to Completion 6
   - 3.2 Scalability 6
   - 3.3 Cost effectiveness 6

4 Existing Solutions  
   - 4.1 Local workstation execution 7
   - 4.2 Multiple Docker execution 7

5 Solution Architecture  
   - 5.1 Jobs 8
   - 5.2 Containers 9
   - 5.3 Data 9
   - 5.4 Cluster 9
   - 5.5 API 10
   - 5.6 Scheduler 10
     - 5.6.1 Allocation 10
     - 5.6.2 Job execution order 10
   - 5.7 Client 11
   - 5.8 User Interface 11

6 Implementation  
   - 6.1 Stack 12
     - 6.1.1 Django 12
     - 6.1.2 PostgreSQL 12
     - 6.1.3 Celery 12
     - 6.1.4 Redis 12
     - 6.1.5 Swagger 12
     - 6.1.6 Angular 12
     - 6.1.7 Google Cloud Platform 13
     - 6.1.8 Kubernetes 13
   - 6.2 Quality Assurance 13
   - 6.3 Code quality assessment by SIG 13

7 Performance Analysis  
   - 7.1 First run 14
   - 7.2 Results 14
   - 7.3 Run time consistency 14
   - 7.4 Scheduling 17
   - 7.5 Second Iteration 22
Results

8.1 Time to Completion

8.2 Scalability

8.3 Cost effectiveness

Conclusion

Ethical considerations

Discussion

Embedding

Predictability of run times

Memory requirements

Scalability

Original project description

Info sheet
1 Introduction

Veneficus is a data analytics and consultancy company operating out of Rotterdam. It employs a combination of data scientists and software developers to serve clients operating in retail, e-commerce, real estate, insurance and the public sector. The projects vary substantially, ranging from assortment optimization of supermarkets, location analysis of potential store locations, sales forecasting, predicting property values and dynamic pricing.

To streamline the process of analytics projects, the company has developed an array of web-based tools, packaged into a single framework. These tools handle tasks such as data gathering, reporting, user permission management and much more. They are primarily built in Python, making use of the Django framework. The analyses performed by the data scientists are largely based in R, though some data scientists are switching to Python.

Analytics have become more and more important\(^1\) and data volume has been rapidly increasing over the past few years\(^2\). For a company like Veneficus, that means having to massively scale their computational capacity to handle all this data. They must keep operational costs under control and can not simply wait longer for an analysis to finish.

The total amount of work to be done is determined by business requirements and the efficiency of the analyses. Throughout this report, we will discuss Total Machine Time (TMT) as the total amount of computation time the system as a whole consumes. The paralellization offered in our solution allows us to separate this concept from the Time to Completion (TTC), which we define as the amount of time between the start of the first job and the end of the last job. While TMT is often given by the amount of data needed to be processed and can only change by optimization of the analyses themselves, TTC can be vastly lower than TMT. For example, 10 jobs taking 10 hours each will have a TMT of 100 hours, but when run in parallel on 10 machines, have a TTC of only 10 hours.

This project will aim to apply advances made in the field of cloud computing to the processing needs of a data science company. The goal, as further explained in section 3, is to provide the data scientists with an easy and affordable way to run large scale analytics quickly and minimize Time to Completion.
2 The problem

In order to deliver a product to Veneficus that is not only usable, but actually used, this project was linked to an active client project of Veneficus. The project in question is a computationally heavy project and the analysis code is still under development. Because the code is not yet finished, we can discuss and influence certain design and optimization choices with the data scientists.

2.1 The client

The client is a large globally operating conglomerate of non-food retail stores. It has over 10,000 stores worldwide and sees billions of customers per year both online and in stores. In the Netherlands, they operate several chains of stores, totaling hundreds of locations. After discussing with the client, we have decided not to name them in this report.

2.2 The project

The project Veneficus is doing for the client focuses on one chain of stores. This chain is very promotion-oriented, so it is reasonable for them to want insights into how effective these promotions actually are. Their current forecasts are limited to looking for similar periods and promotions in the past and copying those sales numbers. Veneficus was hired to model these promotions and their effect on sales more accurately.

In later iterations, this model will be expanded to include more drivers. For example, in a process known as pull-forward, promotion in a given week will cause consumers to stock up on that product, hurting revenue in the following weeks. Also, promotion on a given brand or product will lead to a decrease in revenue from competing brands and similar products; this process is referred to as cannibalization. These effects and more need to be included in the effectiveness forecasts if the client wants a clear picture of what their promotions add in terms of revenue and profit.

2.3 The analysis

The analysis to be performed is the training of a Bayesian inference model that forecasts sales (volume) and revenue (value) for given products, weeks and promotions. Products are divided into categories, which are divided into 458 different subcategories.

While the training of the model is by nature a sequential process and not trivially parallelized, all subcategories can be trained independently of one another. This means that the analysis, when viewed as a whole, can be run on hundreds of machines in parallel.

The framework used for the Bayesian inference is called Stan\textsuperscript{3}. It allows data scientists to write their model code in a very human-readable way. This code then gets compiled into C++ code for faster execution. The Stan framework is an open-source project and can be used in many programming languages such as R, Python, Matlab and many others.

There are 458 models to be trained; initially this will be done on a monthly basis, with an option to increase this to a weekly iteration schedule. Run times for the model training vary wildly between 20 minutes and 10 hours. At the beginning of the project, this made the best estimate of total workload to be performed between 150 and 1,000 hours, with an absolute worst-case scenario of 5,000 hours of computation having to be performed on a monthly basis.
3 Objectives

In this section, we will describe what this project sets out to achieve. We will discuss the metrics that we hope to improve and discuss the targets to which the project members will be held accountable. For convenience we will denote our collection of jobs \( N \) by their run time \( n_0, n_1, \ldots, n_k \) and start time \( s_0, s_1, \ldots, s_k \). Our Total Machine Time is given by equation (1):

\[
TMT = \sum_{i=0}^{k} (n_i)
\] (1)

3.1 Time to Completion

Time to Completion (TTC) is the time interval between the first job starting \( s_0 \) and the last job finishing. When the last job finishes depends on our way of running the jobs. When running all jobs sequentially in a single thread, we would have (ignoring overhead from orchestration) our TTC becomes equal to TMT.

\[
s_0 = 0
\]

\[
s_i = s_{i-1} + n_{i-1}
\]

\[
TTC_{seq} = s_k + n_k - s_0 = \sum_{i=0}^{k} (n_i) = TMT
\] (4)

Whether the client needs this done monthly or weekly, this is likely to far exceed the required time window, as our lowest estimate for TMT is 150 hours. In order to achieve an acceptable TTC, we need parallelization. When running multiple jobs at once, starting them at a regular interval \( c \), our TTC would be given by the last job to finish, see equation (6):

\[
s_i = i \cdot c
\]

\[
TTC_{par} = \max_{i} (i \cdot c + n_i)
\] (6)

It is worth noting that a large value for \( c \) will have a very large impact on TTC. In fact, if \( c \geq \max_i(n_i) \) then \( TTC_{seq} \leq TTC_{par} \). However, if \( c \) stays small (in the order of seconds) and we can schedule the jobs efficiently, its effect on TTC becomes negligible. The shorter running jobs starting one or two hours after the first jobs likely will not push back the TTC and thus \( TTC_{par} \) should be independent of \( k \).

3.2 Scalability

If the number of subcategories \( k \) were to increase, or there should arise a need to perform multiple tasks on each subcategory, such as forecasting several different time frames, the solution will have to scale in such a way that the TTC is still of the form given by equation (6).

Furthermore, increasing \( k \) should not cause significant overhead to the data scientists that trigger and/or monitor the analysis.

3.3 Cost effectiveness

The analysis is known to require a lot of RAM, so we select a 1vCPU 6.5GB virtual machine at Google Compute Engine as our benchmark machine. This machine costs $0.068 per hour, so a reasonable expectation of the total costs of an analysis would be \( 1000 \times 0.068 = 68 \) dollars.

We should expect these costs to scale linear with TMT, with an as small as possible constant, meaning our computational costs involve little to no overhead.
4 Existing Solutions

Before the start of this project, a job batch of the size described in section 2.3 has never been done by Veneficus. During the exploratory phase, individual subcategories were processed on the workstations of the assigned data scientists. Later, a cloud-based solution was implemented, but was still severely limited in terms of scalability. This section will explore these solutions, their advantages, drawbacks and design choices.

4.1 Local workstation execution

During the development of the model and the code required to train it, the two data scientists working on this project were running a single model on their own laptops. These are fast machines, running an Intel Core i7-8550U processor and 16GB of RAM. The CPU’s have a base clock of only 1.80GHz but are capable of boosting that to 4.00GHz under heavy load. Running the code locally has definite benefits during development, as it improves observability of code execution through things like breakpoints and print statements. Furthermore, the high maximum clock frequency of these machines meant that local runs were occasionally completed much faster than cloud-based solutions. However, these performance gains were inconsistent, as the data scientist would be performing other tasks during execution, and the CPU might have to perform context switches quite often to deal with these other tasks.

Clear limitations of this method are scalability and overhead. A command has to be manually executed for each subcategory. If we estimate one minute per subcategory for triggering the run, keeping an eye on whether it has finished, failed, or is still running, and keeping track of which subcategories have been executed would add up to a full eight hour working day of overhead per batch run.

4.2 Multiple Docker execution

When the limitations described in section 4.1 started to outweigh the benefits, a new solution was implemented. A four core virtual machine was spun up on the Google Compute Platform and four instances of RStudio were run on this machine. The data scientists would log into these web-based IDE’s, load the codebase and data, and trigger the jobs. This solved the issue of the analysis locking up resources with regards to the data scientists and their laptops It improved, but did not completely solve the scalability issues. It also did not solve the other issues of monitoring- and triggering overhead.

Furthermore, a four core, high memory machine on Google Compute Platform will cost roughly $150,- per month, regardless of whether the system is actually being utilized or sitting idle.
5 Solution Architecture

Because the analysis of a single subcategory can not be parallelized, the complete analysis is split into many jobs, each processing a single subcategory. The analysis of a subcategory is run in a Docker container, that is executed as a job on a Kubernetes cluster. The jobs are scheduled through an API we have built. In this section we will further describe these components. A diagram of the full architecture can be found in figure 1.

![Figure 1: The architecture of the system](image)

5.1 Jobs

Because the processing of a single subcategory can not be parallelized, the complete analysis is split into jobs. A job will analyze a single subcategory. We define a batch as a set of jobs that together perform the complete analysis. A job is defined as:

- Source code
- A set of dependencies
- A command
- Arguments to the command (optional)
- The expected run time (optional)
- The required memory (RAM) (optional)
- A deadline (optional)

In the case of the analysis described in section 2.3, the command is `Rscript main.R`. One of the dependencies is therefore that R is installed. Next to R these jobs need to have a list of R packages installed as well as the proprietary code developed by the data scientists at Veneficus. As arguments this job expects the week for which to analyze the promotions and the subcategory to analyze. The week to analyze is constant within a batch (usually the previous week), the subgroup differs between jobs.

The TTC and cost effectiveness are often contradictory goals. Running on preemptible machines is very cheap, but having the machines preempted will hurt TTC. To be able to know when TTC is more important than cost effectiveness a deadline can be given to the jobs. This deadline is determined by delivery date at which the client expects the results from Veneficus. When a deadline is given, an expected run time is also required, to enable the scheduler to reason about the deadline. How this is done is explained in section 5.6. How the run time is forecast is discussed in section 7.2.
5.2 Containers

To make the jobs reproducible and transparent, we have chosen to create Docker images that define the job environment. The images contain the proprietary source code of the analysis that is written by the data scientists at Veneficus, as well as the R packages installed from CRAN that are dependencies of this code. The command that is run in the jobs is also defined in the image. By choosing for docker images, the environment in which the jobs run can be shared in version control systems such as Git. It can be defined outside of our product, enabling the data-scientist to, for example, add a dependency without having to install this dependency in the cluster. The image is built and pushed to the registry when the Dockerfile is merged to the master branch of a GitLab Git repository. This is done automatically by GitLab CI.

The container designed to run the analysis defined in section 2.3 starts off with a stable R version in an image created by The Rocker Project. On top of that image, it installs a list of R Packages from CRAN as well as the proprietary code developed by the data scientists at Veneficus. This R code reads input data from what it considers its local file system. The directory is actually a mounted network drive as further explained in the next section.

5.3 Data

The analysis needs to read several large, confidential data files. It would be very inefficient to first download these from a secure location for every job. The solution for this is to instead have one container running, which mounts a Google Persistent Disk in a read-write-one mode and downloads the data to this disk only once. It offers the data as an NFS server to the rest of the cluster. Another advantage of this approach is that the data is already local to the cluster, resulting in a very low latency. The analysis containers will mount this NFS server in a read-write-many mode to allow all containers to read the files simultaneously. This mounting is handled by Kubernetes so, as mentioned previously, the R script that runs in the container can assume the files exist in its local file system.

5.4 Cluster

The Kubernetes cluster can create virtual machines in so-called pools. A pool describes the type of machine that will be created, so machines in the same pool are the same and interchangeable. The scheduler will send jobs to Kubernetes and Kubernetes will allocate them to a machine in the appropriate pool. Which pool a job is allocated in is determined by the scheduler and sent with the job to Kubernetes in the form of tolerations (covered later) and resource requests.

An important way in which the pools differ is that some of them run on preemptible machines instead of regular instances. These preemptible instances are considerably cheaper (20% of the costs) than regular instances, but might be terminated at any time. We will elaborate on this in section 5.6.

Some of the analyses require more memory than others. The scheduler described in section 5.6 requests more memory for these nodes, which makes the Kubernetes scheduler allocate these jobs to a pool with nodes that have enough memory.

The cluster contains the following four pools:

**default-pool**  The default-pool is running on non-preemptible, 1vCPU machines with 3,75 GB of RAM. This pool is used to run the Kubernetes master, as well as static infrastructure such as the NFS server mentioned in section 5.2.

**n1-highmem-pe**  This pool makes up the majority of the cluster when fully operational. It consists of preemptible 1vCPU machines with 6,5 GB of RAM. This pool has specified a so-called taint. Jobs will only be allocated to a tainted machine when the job specifies a toleration for that given taint. This means that the scheduler (see section 5.6) can control whether a job will be run on a preemptible node or on a more stable node.
**n1-highmem**  This pool contains nodes with the same resources as the n1-highmem-pe pool, but runs on regular instances instead of preemptible instances. The jobs are therefore not terminated, giving the scheduler more certainty about the time needed for the analysis to complete on such a node.

**n1-ultramem**  There are jobs that require more than 6.5 GB of RAM. In order to successfully run these jobs, the n1-ultramem cluster contains instances with 24 GB of RAM. This is probably overkill for most jobs, and more pools could be added in between the highmem and ultramem pool, but this would require in-depth knowledge and estimation of memory requirements.

<table>
<thead>
<tr>
<th>Machine type</th>
<th>vCPU's</th>
<th>RAM (GB)</th>
<th>Regular price ($/h)</th>
<th>Pre-emptible price ($/h)</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>1</td>
<td>3.75</td>
<td>$0.052</td>
<td>$0.011</td>
</tr>
<tr>
<td>n1-highmem</td>
<td>1</td>
<td>6.5</td>
<td>$0.068</td>
<td>$0.014</td>
</tr>
<tr>
<td>h1-ultramem</td>
<td>1</td>
<td>24</td>
<td>$0.252</td>
<td>$0.052</td>
</tr>
</tbody>
</table>

Table 1: Overview of VM types and their prices

### 5.5 API

We run an API server that accepts and responds with JSON requests. The API can be used to create (a batch of) jobs and to retrieve data about jobs. Data aggregated per batch can be retrieved, such as the TTC, TMT, costs and progress of the batch. When further investigation is needed, the run time, costs and whether the job has been run on a preemptible can be retrieved per individual job. When the API server receives a request to create jobs, it stores the received data, such as job name, Docker image name, arguments, expected run time and deadline in a Postgres database. It passes the primary key by which this data can be identified to the queue, so that the data can later be retrieved by the scheduler.

### 5.6 Scheduler

The scheduler reads the jobs to process from a Redis database using Celery, an open-source distributed task queue for Python. We have chosen to use Celery because developers working at the company have already familiarized themselves with Celery, meaning the transfer and adoption at the end of the project will be easier. Furthermore Celery meets our objective of scalability, as it supports having multiple workers out of the box. This makes it easier to balance the scheduling load over multiple schedulers when the load increases so much that it cannot be handled by one scheduler anymore.

The scheduler performs two tasks: it determines the order in which jobs will run on the cluster, and it instructs Kubernetes which pools the jobs should be allocated to.

#### 5.6.1 Allocation

When the scheduler takes a job from the queue, it determines whether it will run the job on a preemptible or a regular instance. If no deadline is given, the scheduler will opt for the cheapest option and allocate to a preemptible pool. If a deadline (and expected run time) is given, a decision needs to be made. If the time before the deadline is more than twice the expected run time, the job will be allocated to a preemptible node. In the worst-case scenario, the machine will be terminated (preempted) in the last second of the job, and take the job down with it. The scheduler will re-evaluate, observe that it now doesn’t have twice the runtime and choose TTC over budget by allocating to a non-preemptible machine.

#### 5.6.2 Job execution order

The jobs can not be sent to the Kubernetes scheduler all at once, the reason for this is explained in section 7.1. Therefore, only one job is scheduled every $c$ seconds and our TTC is given by equations 5 and 6. The scheduler takes a job from the queue that is sorted on expected run time, longest to shortest, i.e. such that given a batch
of jobs \( N = n_0, n_1, \ldots n_k \) it holds that \( n_i \geq n_j \) for all \( i < j \). We define the finish time of a given job \( x \), started at time \( t \) to be given by

\[
f(x, t) = t \cdot c + x
\]

(7)

It can be shown that this Greedy scheduler produces an optimal solution for total completion time by way of reductio ad absurdum:

1. Given a non-optimal sequence of job run times \( N = \{n_0, n_1, \ldots, n_k\} \) as given by the Greedy scheduler, i.e. such that \( n_i \geq n_j \) for all \( i < j \) and an optimal sequence \( O = \{o_0, \ldots, o_k\} \) that is as close to \( N \) as possible.
2. Because \( O \) is optimal and \( N \) is not, we know they are different. This means there must be a pair in \( N \) that is swapped in \( O \). In other words, there must exist an \( i \) and \( j \) such that \( i < j \), \( n_i = o_j \) and \( n_j = o_i \).
3. We observe that since \( n_i \geq n_j \), it must follow that \( o_j \geq o_i \).
4. If we swap these jobs in \( O \) we get the following completion times for solution \( O' \):

\[
j \cdot c + o_i \leq j \cdot c + o_j
\]

(8)

\[
i \cdot c + o_j < j \cdot c + o_j
\]

(9)

5. The right hand side of both of these equations is an actual run time from the optimal solution \( O \), so swapping these jobs in \( O \) produces a solution \( O' \) that is no worse than \( O \) and closer to \( N \) than \( O \).
6. This contradicts the definition of \( O \) given in step 1. We can therefore conclude that \( N \) is an optimal solution.

Although the run times vary because of the non-deterministic nature of the analysis, intelligent scheduling algorithms still produce a significant reduction of TTC. This has been shown in previous studies [5].

### 5.7 Client

A Python client is created to be used to communicate with the API to create jobs. A python script is written that reads a CSV file that lists the subcategories and the sales per subcategory. Based on these sales we predict the run time of the analysis. The script uses the client to create a batch of jobs that contains a job for each subcategory.

### 5.8 User Interface

A User Interface is created to be used by managers or data-scientists to look at summary-statistics of the batches. In the UI an overview of batches can be seen with the TMT, TTC and costs of running the batch. This information can be used when planning and budgetting projects. The UI shows the information in real-time, so it can also be used to monitor progress.

![Figure 2: The UI showing summary statistics of batches](image)
6 Implementation

In this section we will describe the software-stack we have used to compose our solution and explain why we have chosen to use this software. We will also explain what we have done to ensure the quality of the software created.

6.1 Stack

We will discuss some of the off-the-shelf solutions we used for our project, their key characteristics and why they were chosen over their competitors.

6.1.1 Django

Django is a very popular web and Object Relational Mapping (ORM) framework for Python. We were both already familiar with this framework and it offers all the required functionalities needed for this project. Alternatives like Ruby on Rails would also have worked, but would have involved learning a new programming language. Besides that, the existing application built by Veneficus was built in Django, which will make embedding the project within Veneficus significantly easier.

6.1.2 PostgreSQL

PostgreSQL is one of two recommended database backends for Django, together with MySQL. PostgreSQL was chosen for its support of JSON fields and overall superior performance compared to MySQL.

6.1.3 Celery

Celery is a distributed task queue for Python. We have chosen Celery over alternatives, because the community around Celery (11,740 GitHub stars) is bigger than that of alternatives like Dramatiq (1342 GitHub stars) and rq (5347 GitHub stars). Celery has been around longer and has been heavily tested and used in production.

6.1.4 Redis

We have used Redis as the broker for Celery, because we would like to use this for caching as well. Alternatives such as RabbitMQ do not support being used as a cache backend. Because Redis can be used for this as well, we can limit the number of dependencies we have in our project. This makes it easier for maintainers to know all dependencies in depth, which decreases the number of potential bugs.

6.1.5 Swagger

We have used Swagger to generate and serve interactive API documentation. We have chosen it over alternatives like Doxygen because it also features generating code, such as clients to connect with the API and mock servers. We have used the client generation feature to generate an R client for our API.

6.1.6 Angular

We have used Angular as our frontend application framework. We have chosen it over React because of its complete MVC and the dependency injection. This is particularly useful in a small application like ours, in which we only have several components and services. We do not have to adopt extra dependencies to get a complete MVC.
6.1.7 Google Cloud Platform

Google Cloud Platform (GCP) offers both regular and preemptible virtual machines. The main competitor is Amazon Web Services (AWS), who offers the same functionality (preemptible machines are called spot instances on AWS). Both offer per-second billing after a minimum charge of one minute. The reason GCP was chosen was based on preventing ingress and egress charges. The dashboarding and data interface to the client are already hosted on GCP by Veneficus and moving the data from AWS to GCP would incur $0.15/GB of additional charges.

6.1.8 Kubernetes

Kubernetes is realistically the only stable orchestration framework available. It offers vastly more functionalities with regards to scheduling and allocation than alternatives like Docker Swarm. Furthermore, since it was designed by Google, its integration with the Google Cloud Platform is excellent.

6.2 Quality Assurance

The software is being used internally by Veneficus. In order to make sure the software we build is the software the data scientists (the users of the software) want, we held regular meetings with them to discuss with them their working process and possible solutions. During these meetings we formed the requirements of the software.

Because the product is being used internally, potential bugs in the software do not impact Veneficus’ clients directly. However, they can take some time to resolve and block the data-scientists in their work. Therefore, we have taken some measures to assure quality of the software.

As time was limited, we have chosen to focus on components in which bugs can go unnoticed, such as when some jobs are not being run on preemptible VM’s, while they should. As no errors are returned to the user and results come in as expected, these kind of bugs can easily go unnoticed and can take a while to be discovered.

Logic that is more likely to be noticed, such as bugs in validation of names that are passed to Kubernetes are not covered. For example, when our product does not correctly validate a name that will be used as the Kubernetes Pods name, and such a name is being picked, it will incorrectly pass the validation of our product, but will not pass Kubernetes’ validation. Therefore, Kubernetes will return an error. This will be noticed, as results are not being delivered when Kubernetes does not allocate the job. To streamline detection of this kind of bugs, monitoring should be set up, so that these errors are reported when they occur.

In section 6.1 we covered the stack we are using. This stack exists of established open-source software. Because the software is used a lot, bugs are found quickly. As the source-code is openly shared, anyone can fix bugs that are found. Therefore, bugs are fixed relatively fast after being found. Because of the big communities behind the software we are using, the software has been heavily audited.

6.3 Code quality assessment by SIG

The Software Improvement Group has analyzed and studied our code from a testing and maintainability perspective. They considered the code well above average, and marked it at over 4 out of 5 stars. Particularly the Python code has some functions that are at risk of growing too large and complex as the code base increases in size, but they were overall impressed with the code quality.
7 Performance Analysis

In the beginning of January the cluster was ready for a first run of all 458 product subcategories. The cluster was configured to automatically scale to a maximum of 500 pre-emptible virtual machines in the n1-highmem-pe pool. This was not a synthetic 'benchmarking' run, this was production-ready code running on production-ready data. For that reason, several manual interventions had to be performed to ensure the batch made it all the way through. This section explores this initial run, as well as a second run later in January.

7.1 First run

Initially, we attempted to 'schedule' all jobs at the same time, essentially setting $c = 0$. Within the first minutes, the first problem emerged, showing that this was not feasible. All the virtual machines simultaneously attempted to pull an 800MB Docker image from the GitLab registry. This resulted in quite a lot of ImagePullBackoff errors, since GitLab was not able to keep up. 5 jobs were able to start, but 453 gave this error. As Kubernetes attempted to pull these images several times, we expected the jobs to eventually launch and essentially creating a completely random scheduler based on which pod was able to pull the image. This proved unsuccessful, as the pods did not recover and after the initial 5, no more pods were activated. We suspect one or both of two causes for this: Authentication with GitLab was done in the very beginning and a pod that has to pull the image more than 30 minutes after this authentication, will find its authenticated session to have expired. Furthermore, the backoff suggested by the exception does not appear to contain any randomness, causing the failed 453 jobs to try again at the exact same time. This obviously does not solve the congestion issue.

A second issue we encountered was not related to our implementation, but many of the categories had no relevant sales data at all over the past two years. This naturally meant that no meaningful forecast could be made. This was probably caused by the client re-labelling certain products causing empty categories.

The batch was canceled and a second batch was started, now with a rather conservative $c$ of 30 seconds. This solved the ImagePullBackoff issues, but obviously not the problems caused by missing data. Since we had no estimates for run time of the jobs, they were scheduled alphabetically by the name of the subcategory to be processed. Because the name of a subcategory is not correlated to the data input size, for all intelligent scheduling purposes, this can be considered a random schedule. Several jobs ran out of memory on the n1-highmem-pe nodes, and were manually restarted manually on nodes in the n1-ultramem pool.

7.2 Results

Figure 3 shows the distribution of all run times that ran to completion. The median runtime was 52.4 and the 95-th percentile runtime was still under 10 hours (576.62 minutes).

As expected, the run times of categories varied significantly. For future runs, we would like to be able to predict how long a subcategory will take beforehand. We compare the observed run times to both the total sales volume (the number of items sold within each subcategory) and the number of SKU’s (the number of distinct products sold at least once) within each subcategory.

Comparing these directly, as shown in figures 4a and 4b, does not directly suggest much predictive value. It was discovered that certain jobs started over from the beginning one or several times throughout the process. The cause of this has not yet been discovered. The jobs that ran on different hardware, performed more than one run, or both, represent a significant portion of the outliers, as can be seen in figures 4c and 4d.

Since these jobs do not represent a reliable measure of run time, the decision was made to exclude them from the dataset. Looking at figures 4e and 4f we can say that the number of distinct products is a much more significant indicator for runtime.

7.3 Run time consistency

The analysis uses Hamiltonian Monte Carlo to move between so-called states, in each step drawing with a certain probability of that draw being accepted or not. The analysis is finished when a given number of accepted draws has been reached. It is possible for the model to become trapped in a state with a low probability of
Figure 3: Histogram of run times
Figure 4: Breakdown of run times
acceptance and spend a lot of time there before moving on. This makes run times of identical models inconsistent and makes it hard to predict run times. To observe this, we have taken two subcategories and ran each of them ten additional times.

Subcategory A was chosen because it was extensively used during development and is considered the most stable subcategory. Subcategory B was a suspected outlier in the initial run and can be easily identified in figures 4c and 4d as having roughly 11 million items sold across 153 distinct products, but with a run time of 1.171 minutes. This is well beyond the time a job of this size is expected to take.

Figure 5a shows that the 1.171 minutes for subcategory B can be considered an outlier, as this same analysis completes in considerably less time in later iterations. It is however clear to see that run times still vary greatly, even when not considering outliers (figure 5b). As mentioned earlier in 5.6, the benefits of intelligent scheduling remain present even when run time is highly stochastic.

7.4 Scheduling

Figure 6 shows the 132 jobs that were left after the unreliable runs from figures 4c and 4d were removed. They are shown in the order they were started. We expect the time spent in the queue to gradually increase over time, however the ‘jumps’ in the time spent in queue represent removed outliers. The blocks on the right hand side with a significantly higher time in the queue show jobs that had to be manually restarted at a considerably later time, but on the same hardware as the rest of the batch. The TTC for this run was 2,430 minutes, or 40 hours and 30 minutes.
If we correct for the manual interventions and the issues with the data, we can get a clearer picture. That is to say, if we had known that exactly these 132 jobs would need to be run and no others and no manual restarts would have been required, we could have started many of the jobs sooner than we actually did. Figure 7 shows what that timeline would have been, and we can see that the TTC would have dropped to 1,459 minutes, or 24 hours and 19 minutes.
At this point, it becomes clear that intelligent scheduling can have some benefits. The long-running jobs that started very late in the batch ultimately dominate the TTC. Had these been scheduled earlier, TTC would have been lower. If we consider the number of distinct products in a subcategory to be an estimator for run time, and had scheduled with a Greedy approach with regards to this estimator, we would have observed the timeline shown in figure 8. The TTC then would have been 23 hours and 50 minutes. This is 29 minutes or 2.0% better than the observed approach.
Figure 8: Start and end times of jobs had they been scheduled by number of distinct products

Obviously, knowing the actual run times, we can also determine what the ideal schedule would have been. By starting the jobs in decreasing order of run time, we observe the timeline shown in figure 9. This schedule would have completed in 23 hours and 36 minutes, only a 14 minute (0.98%) improvement over the scheduled approach.
Comparing the four timelines, we see that intelligent scheduling can reduce TTC, and that using the number of distinct products comes quite close to an optimal solution. Figure 10 shows this comparison. Please note that the y-axis for figure 10 starts at 1,300 minutes for improved readability.

Figure 9: Start and end times of jobs in an optimal schedule

Figure 10: Comparison of TTC
7.5 Second Iteration

We ran a second batch of jobs based on the results of the January run. Since the results of this run were not going to be delivered to the client, we could exclude any jobs that failed due to data errors, or any jobs that ran on the n1-ultramem cluster, in order to get a clear result of the performance of the scheduler. Furthermore, we decreased \( c \), the time between two jobs being sent to the cluster, to 12 seconds. This decreased the influence of the time spent in the queue even more (the last job started less than 30 minutes later than the first job). It is worth noting that decreasing \( c \) also makes the effect of intelligent scheduling smaller. This can also be seen in equation \( 4 \).

In this run the jobs were scheduled in order of their expected run time (as explained in more detail in section 5.6). The estimated run time was given by the distinct number of products in the subcategory. The fact that this is a number of products rather than a unit of time does not matter in this case, as we are only interested in the order of execution, not the actual expected run time, as long as the expected run time is lower than 24 hours (which was the case for all subcategories).

We did experience a problem during this run, as the NFS server used for reading data files and writing results crashed due to lack of resources in the default pool. Most, if not all, jobs had by that time reached the stage of the analysis where the disk was not needed, and could continue working. However, when they finished that stage and attempted to write their results to the NFS server, they got no reply. 160 minutes into the run, we were able to solve the resource issue in the default pool and recover the NFS server. At that time, most of the jobs completed their analysis and were reported as finished. The consequences of this event were a considerably higher TMT than the previous run, and can be clearly seen in figure 11.

The last job to finish did so after running for 21 hours and 27 minutes. Based on the number of SKU’s it was
scheduled as the 27th job and started 5:12 minutes after the first job. The run time for this job in the first batch was 23 hours and 36 minutes the first time.
8 Results

Two batches have been executed, one scheduled alphabetically and the other scheduled using Greedy scheduling based on number of SKU’s. In this section, we will look at the results and compare these to the goals we set in section 3.

8.1 Time to Completion

The total TTC for both batches is shown in table 2. In the second batch, had the longest running job been scheduled first, this could have been decreased by 5 minutes and 12 seconds. Our estimation and scheduling therefore resulted in a TTC that was 0.402% higher than the optimal solution.

Furthermore, we have shown that our start interval $c$ of 12 seconds, as described in section 3 and equation 6 has no significant influence on TTC.

8.2 Scalability

As discussed above, we can see that TTC is dominated by the execution time of a few large jobs. Since the compute cluster can scale effortlessly, adding more jobs to the system is very unlikely to cause a significant increase in TTC. The effort of data scientist in triggering jobs is constant in the number of jobs, as this can be automated by writing scripts that use the client to create batches. Monitoring can also be done without extra work when the number of jobs increases, as statistics are provided that are aggregates of all jobs in a batch.

8.3 Cost effectiveness

All of the jobs in section 7.5 were run on preemptible virtual machines, causing the total analysis of 322 total computing hours to be completed for $4.50. It is difficult to say much about the costs if this were done through the local execution method of section 4.1 but the multiple Docker execution method would have required 33 of the mentioned four core machines, costing $5.000 per month when leaving them always on. Manually turning these machines on when needed and off when not needed would surely be cheaper, but we must not overlook the overhead costs of managing this infrastructure by hand.

The constant costs of running the cluster are $50 per month. These costs are made to have a Kubernetes master node and the application running at all times, so that requests can be processed. Although these costs appear to be high in comparison to the variable costs of running one batch, these costs are quite low compared to having the computing machines always ready.

<table>
<thead>
<tr>
<th></th>
<th>Run 1</th>
<th>Run 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of jobs started</td>
<td>458</td>
<td>132</td>
</tr>
<tr>
<td>Number of jobs finished</td>
<td>138</td>
<td>132</td>
</tr>
<tr>
<td>Total machine time (TMT)</td>
<td>305:47:41</td>
<td>321:53:55</td>
</tr>
<tr>
<td>Time to Completion (TTC)</td>
<td>40:30</td>
<td>21:27</td>
</tr>
<tr>
<td>Cost</td>
<td>$14.44</td>
<td>$4.50</td>
</tr>
</tbody>
</table>

Table 2: Overview of results from both runs
9 Conclusion

By running the jobs on an auto-scaling Kubernetes cluster, the number of jobs run can scale infinitely. Because the analyses are parallelized and a Greedy scheduler is used, the Time to Completion is minimized. A decrease from 322 hours to 21.5 hours is achieved, a reduction of 93.3%. Furthermore, costs of operation are reduced by running the jobs on cheaper machines when possible. Idle machines are terminated, which further reduces the costs of infrastructure. The total reduction of infrastructure costs is $4,945.50, a reduction of 98.9% from the otherwise spent $5,000. Orchestration of the necessary infrastructure is automated, eliminating the costs of maintaining this infrastructure, which was estimated to be around eight working hours per batch run of 500 jobs.
10 Ethical considerations

The data being analyzed is sales data. It is however aggregated by product and by store, noting how many items of each product have been sold in each store. This means the data is in no way traceable to individual purchases and privacy considerations or GDPR regulations are not applicable to this project.
11 Discussion

While this project has shown significant benefits over existing solutions, further work is still possible. In this section, we will cover areas that require or deserve more attention at a later time.

11.1 Embedding

This project was built as a standalone application. In order to maximize its usefulness to Veneficus, it should be integrated into the existing set of tools that Veneficus uses. It should also be embedded as a standard way of working within the organization. It can then benefit the company, both from a financial point of view and an ease-of-use perspective for the data scientists. Once the integration is completed, the data scientists, or a subset of them, can be trained in Dockerizing their projects and then a user manual should be created for triggering these jobs.

11.2 Predictability of run times

We have already discussed the stochastic nature of the analysis and the effect this has on run time. While it is conceivable that an analysis might run 50% longer because of this, or even twice as long, we have observed fluctuations in run time of over 1000%. Further research and development, in cooperation with the involved data scientists is needed to find the cause of this. Since these long running jobs dominate the total TTC, eliminating these outliers from our analyses will substantially lower TTC.

11.3 Memory requirements

We expect the end user, likely a data scientist, to provide an estimate of the RAM requirements of a job. In this project, we have studied meta data to produce estimates for run time to provide the end users some assistance in feeding this to the API. The same could be applied to RAM requirements. The allocation mechanics in our solution and Kubernetes already allow for control over which node pool a job is executed in.

11.4 Scalability

When scaling to the extremes (millions of jobs), some parts of the system still require some changes. On the scale of the project for which we have created the product these changes are not needed. When the number of jobs is extremely high, the small delay c between consecutive jobs might become a significant factor in TTC. When this is the case, one might want to scale the Docker image registry or use Google Instance Templates to spin up machines that have already fetched the image. By doing so, the time between consecutive jobs can be decreased or even removed. Taking this even further, multiple schedulers could be used, so that scheduling goes even faster. This can be easily done, as our scheduler supports multiple workers (see section 5.6).
References


A  Original project description
Bachelor Project proposal
Jorick Spitzen en Simon Barendse

The problem
Data scientists at Veneficus often need to run extensive calculations that take a long time to complete. These calculations are either performed on the Data Scientist’s own working computer, hindering their capability to perform other tasks and taking longer than necessary, or sometimes on dedicated machines. These machines, though completing the calculations faster, require significant set-up costs and maintenance. Also, they often sit idle for the majority of the time, costing the company money.

The project
The proposed project is to design and implement a platform through which Data Scientists can execute their models in a quick and easy way. The platform will allow the data scientist to make decisions on computational power (speed vs price), but also about the nature of the computing hardware. For example, for a Monte Carlo simulation a data scientist will need many processing cores, while for the analysis of a large existing dataset, more memory is needed.

The platform should be simple to use by Data Scientists who are used to working with either R or Python code.

The company
Veneficus specializes in (big) data analytics and Software Development. A team of 30 developers and econometricians work together. The econometricians perform data analytics using R on cloud-based systems, Developers work in a variety of systems, Python (Django) is the primary used framework. The Project sits at the heart of the organization and will fall under guidance of one of the partners of Veneficus.
B Info sheet

Title of the project: Job containerization and orchestration to reduce TTC and operational costs
Name of the client organization: Veneficus B.V.
Date of the final presentation: February 14th 2019

Project description This project shows the process of setting up a autoscaling compute cluster, as well as an interface that data scientists with little to no knowledge of Kubernetes or other orchestration frameworks can use. We set out to achieve a scalable, cost-effective and easy to use solution for an analytics project by Veneficus. We researched existing (partial) solutions and scheduling algorithms. We worked together with the Data Scientists at Veneficus to continually test and improve our solution.

Members of the project team

Name: Simon Barendse
Interests: Security
Contribution and role: User Interface, Scheduler, API

Name: Jorick Spitzen
Interests: DevOps, Data Science
Contribution and role: Cloud Infrastructure and analytics

Client

Name: Robbert Bos
Affiliation: CEO, Veneficus B.V.

Coach

Name: Lydia Chen
Affiliation: TU Delft

Contact

Name: Jorick Spitzen
Email: jorick.spitzen@veneficus.nl

The final report for this project can be found at: http://repository.tudelft.nl