

IN4392 Cloud Computing TU Delft

Group 12

Ingimundur Vilhjalmsson

4911326
i.vilhjalmsson@
student.tudelft.nl

Sjoerd van den Bos

4239261
s.r.vandenbos@
student.tudelft.nl

Wouter Morssink

4590643
w.r.morssink@
student.tudelft.nl

D.H.J. Epema

Instructor
d.h.j.epema@
tudelft.nl

S. O. Khorasani

Assistant
s.omraniankhorasani@
tudelft.nl

C. Ileri

Assistant
c.u.ileri@
tudelft.nl

ABSTRACT

Due to the increasing popularity of cloud computing, WantCloud BV wants their next application to be cloud-based. Their idea is to create a conversion service which allows users to convert MP4 formatted videos to AVI videos. This report proposes an automated system which utilizes several different services from Amazon Web Services, and is capable of scaling according to the demand while also being failure tolerant. The created application consists of a master machine, which takes care of the spawning of worker machines, and worker machines, which process the files. The application proved to scale well during the experiments and provided the lowest processing times with unrestricted worker scaling. Furthermore, it was found to be reliable in the event of worker failures.

INTRODUCTION

WantCloud BV is currently looking at creating a video conversion tool online. With them learning about cloud computing it was required that the system would be a cloud-based application. Cloud computing is becoming more and more popular for businesses like WantCloud BV, as it provides a reliable service and does not require the company to invest in high-end hardware in order to run their applications [9]. With some general knowledge of cloud computing, WantCloud BV wanted us to set up the application according to some general requirements.

There are already several different video conversion tools available on the internet, but not all of them are easy to use, or work online. An example of such a service is Freemake's video converter [8], which requires the user to download the program on their computer before they can convert a file. Another example is CloudConvert [7], which provides an online conversion tool. However CloudConvert does not offer infinite conversion, but requires credits, which can only be bought in bulk.

Since we were already somewhat familiar with Amazon Web Services [6] (AWS), we decided to make use of various different services it provides. The general idea behind the application is that users can easily submit MP4 formatted files via a web-server. After the submission the application converts the

file into a video with an AVI extension, which can be accessed and downloaded by the user.

Throughout the remainder of this report, we will go over the background of the application by discussing the requirements set by WantCloud BV in detail. Next, we give a detailed overview of how the system components work and interact with each other. To prove that the created application works according to the requirements, we evaluate the system with individual experiments on each requirement.

BACKGROUND ON APPLICATION

Our application allows users to convert MP4 files to AVI files. The process is all done on the cloud so that users don't need to download and install any software. The users interact with the system through a website where they upload files through and are then provided with their result after the conversion is complete. The process of conversion is done without requiring any action from the user except submitting their files.

Besides providing the conversion service to the users, the application must fit the demands set by WantCloud BV. WantCloud BV is familiar with the possibilities of Cloud Computing, so it set some requirements for the application. The main features necessary for the application are described in the table 1 below.

Requirement	Explanation
Automation	Working without human intervention
Elasticity	Auto-scale the amount of VMs to demand
Performance	Resource allocation
Reliability	Keep working in case of failure
Monitoring	Track the status of the system

Table 1: Requirements of WantCloud BV

Automation means that WantCloud BV wants to deploy the application once and from that moment forward the application should work without requiring any actions to keep it running.

Furthermore, the system should auto-scale to the demand of the users. When the application receives many requests, it should automatically lease more instances to take care of the workload. The same logic applies when the amount of requests drop, then the system should release machines that are idle. Closely linked to this is the performance requirement, as the system should balance the load of the service across the VMs in the current resource pool of the application. Furthermore, WantCloud BV demanded a reliable application, meaning that when a VM malfunctions, the application should be able to continue running and restart the work which was lost by the faulty VM. The last main requirement is that WantCloud BV wanted insight into the application by means of monitoring.

SYSTEM DESIGN

In order to give an intuitive idea of how the application works, we will first explain the process view of the system. This means that we will explain in order of execution what happens in the system after a customer has made a request for a video conversion.

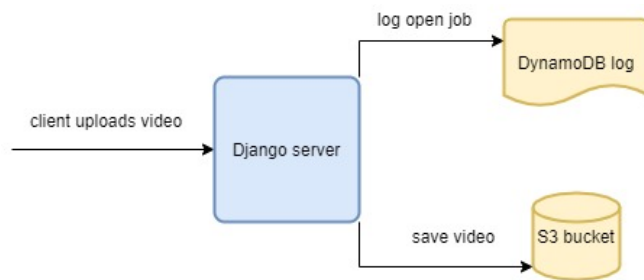


Figure 1: Flow diagram of the job acceptance process

Process View

In order to enable the customer to make requests, we have set up a web server that, when requested, returns a form where a customer can upload the video file they wish to be converted. After the client has selected their file and clicked on the 'submit' button two things happen. As depicted in figure 1, the video-file, which is checked to be in MP4 format, is uploaded to an Amazon S3 bucket [5], then a log is inserted into a key-value store. The log is timestamped with the time of upload, is identified by the name of the video file that was uploaded to the bucket, and due to the absence of other attributes such as 'assigned worker id', it can be identified that the video-file to which is referred to has yet to be assigned to a worker.

Independently from the aforementioned process, there is a master process running on an instance in the cloud which periodically polls the key-value store. Whenever it finds new logs, the master process will use a heuristic algorithm that decides how many workers to spawn and accordingly spawns a number of worker instances. The worker instances, depicted in figure 2, will scan the key-value store for unassigned jobs. They will pick the oldest unassigned job making this system adhere to a first-in-first-out (FIFO) policy. The workers 'pick' such a job by updating the log with an 'assigned worker id' and a timestamp. In order, a worker instance then downloads the corresponding video-file from the S3 bucket, converts the

file to AVI format, uploads this file to the S3 bucket, updates the log one last time with a timestamp and CPU metrics of its own machine during conversion, and finally deletes both versions of the video-file from disk. The worker then proceeds to scan the log again for more unassigned jobs, in the case that it finds more, it repeats the process described. If it finds no more unassigned jobs it will shut itself down. The last mentioned forms the down-scaling policy of this system.

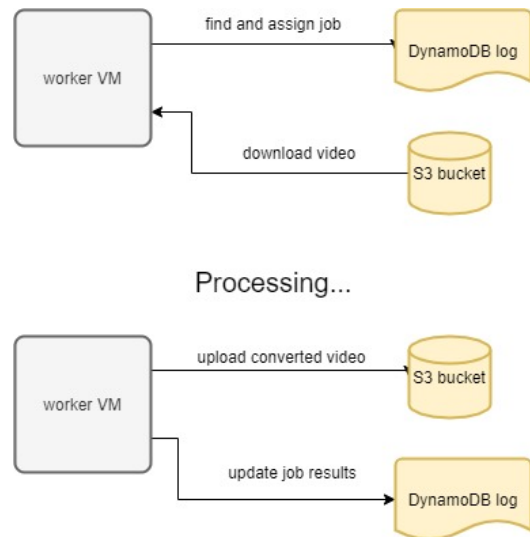


Figure 2: Flow diagram of video conversion

Libraries

We chose to implement our system in Python in the context of AWS [6]. This is done because Python is a compact programming language that allows rapid development and expressive scripting. This enabled us to make use of the Boto3 [1] library. This is Amazon's Python cloud SDK. For VMs we use EC2 [3] instances and for our logging we make use of the key-value store DynamoDB [2]. Boto3 has a number of convenience methods that make it easy to perform CRUD operations on DynamoDB and the S3 buckets. Boto3 also contains convenience methods for the launching, reading and terminating of EC2 instances. Finally we made use of Boto3's CloudWatch libraries to record CPU metrics of the worker instances. For the web server, we used the open-source Django framework. The video conversion itself will be done using the "FFmpeg" command line tool [4].

Although Boto3 allows for fast and easy development of applications on AWS, the disadvantage is that the developer is now dependent on AWS as a cloud-provider and will need to rewrite a significant part of their application when switching or porting to different cloud-providers.

Resource Allocation and Provisioning

Resource allocation in computing refers to the act of reserving resources such as memory, CPU capacity and network capacity. Provisioning, generally, is the act of uploading application code and other resources to machines that are intended to run on those machines.

Resource Allocation

One of the pros of deploying applications on the cloud is that one can make use of other cloud-native services as constituents of one's applications. The advantage here is that for some parts of the system, resource allocation is abstracted away. In our case the S3 buckets and the DynamoDB can hold a practically unlimited amount of data without the developer needing to program or perform allocation.

Provisioning

Since our S3 and logging are cloud-based services, we do not have to install or in any other way provision machines to be able to make use of these non-volatile data storage methods. For the VMs used as workers, VM used as master and our web server, we do however need to take care of provisioning. To handle this, we created our own Amazon Machine Image (AMI). In this image we specify that machines spawned from this image will have all our business logic installed. We can make calls to spawn VMs from the master using Boto3. The master process uses an algorithm that spawns a number of worker VMs proportional to the workload on the system, described in equation 1. These VMs are programmed to terminate in the absence of work, making sure the system is not reserving more resources than it needs. The provisioning of the master VM, which also serves the Django content, is done manually by using the Linux utility of secure copy and then using PIP to install all dependencies. In future work we could include a separate AMI for the master to speed up recovery after failure of the master.

Monitoring and Reliability

The master continuously checks the log, it compares this to a list held in main memory containing the worker VMs and whether they are running or terminated. From these two sources it deduces the state of the system. This means the master knows how many videos are being converted at every moment, how many conversions are done in total, how many worker VMs are running and terminated, how long the VMs took to convert videos and even the average CPU utilization for every worker VM and every conversion. This last feature is achieved by wrapping the CloudWatch functionality from Boto3 inside our own business logic.

Since the log contains the worker id of every conversion job that has started and an empty timestamp entry for every job not finished, the master can compare the worker ids of jobs not finished with the list of terminated workers and use this to detect crashed jobs. In order to restart the job, the master simply alters the log to allow other workers to recognize this as an open job and consequently assign themselves to it.

EXPERIMENTAL RESULTS

Experimental setup

The EC2 instances used for both the master and the workers were of type `t2.micro` running on Ubuntu 18.04 LTS, they each had 1vCPU, 1GB of memory, and 8GB of storage. In our experiments we used the Django web server to upload the files to S3, since that perfectly reflects the real-world usage of our application. Analysis of the operation and performance

of our application was done from the logs generated by the DynamoDB table.

In order to thoroughly test the system, we created several different workloads to run:

- 1GB of small files (varying from 1.3MB to 15MB each).
- 1GB, 2GB, and 4GB of only big files (varying from 22.5MB to 500MB each).
- 1GB, 2GB, and 4GB of small and big files mixed (varying from 1.3MB to 500MB each).

Our tests were conducted in a manner where 50% of the workloads were loaded into the queue prior to starting the processing, and the remaining 50% of the workload was uploaded sequentially through the web server after processing began. The reason why not everything was loaded into the queue prior to processing or by uploading everything sequentially is because that would likely not showcase the scaling capabilities as well. In addition, our testing method gives an insight closer to what would be observed in real-world use cases.

Automation

Our system provides video conversion with minimal human interaction. First, the master instance needs to be started once by the system owner. Second, when the master is running, the Django web server is accessible through a simple front end that allows users select an MP4 file to upload. Once the file has been uploaded, the master automatically launches a worker which will process and deliver the results automatically.

The scaling of worker instances is also done automatically by the master. Every 10 seconds the master computes the necessary amount of worker instances depending on the size and amount of files that are queued. To enable the automation of newly launched workers we made use of `crontab`, such that once the instance boots up it executes the worker program.

From all of our experiments, we can conclude that the automation worked as intended after only needing to start the master.

Elasticity

As mentioned in the previous subsection, the master instance periodically computes the appropriate amount of worker instances. The algorithm we devised for this is rather simple:

$$M = B + \text{int}\left(\frac{T - B}{5}\right) + (T - B) \% 5 \quad (1)$$

Where M is the amount of needed machines, B is the amount of big files (larger than 20MB) queued, and T is the total amount of queued files. This formula ensures that there are always more worker instances than amount of big files should smaller files also be queued. That means smaller jobs do not get blocked behind big jobs. The master compares the computed M with the previous value so that it can scale according to the queue, up or down.

Figure 3 demonstrates the elasticity that our system has when working with 4GB of only big files. As can be inferred from equation (1) and seen in figure 3, 10 files were queued when

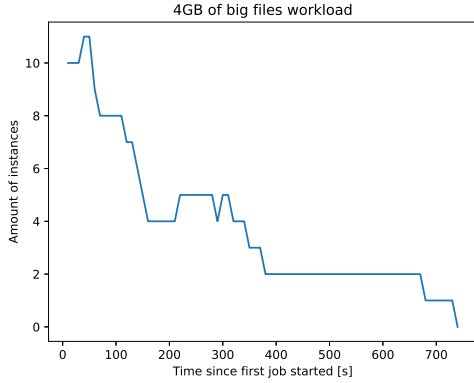


Figure 3: Amount of instances for a 4GB big workload, measured with 30 second windows

the processing began. Although the upload rate of the subsequent files was not recorded, it is evident that the system is perfectly capable of scaling both up and down according to our formula. Similar behaviors were observed in our experiments with the other workloads.

Performance

To test the performance of our system we looked at two metrics: the waiting time of files across the different workload types, and the charged-costs of workloads.

Waiting Times

We define waiting time as the time it takes for a file to get started on after arriving in the DynamoDB table. This metric provides valuable information on whether, for example, small files are getting blocked behind big jobs or if the scaling mechanism is working well. To test this we examined the waiting times of the small files in the workload containing only small files and compared it to the mixed workloads.

The average wait time of small files in the 1GB mixed workload was 40.31s, while for 2GB mixed workload it was 35.13s, and 45.26s for the 4GB mixed workload. Note that the time to launch a worker has an effect on these numbers. Now, the average wait time with the workload of 1GB of only small files was discovered to be 93.64s. These results suggest that the scaling algorithm is not as effective for workloads consisting of only small files because the scaling algorithm favors big files 5x more than small ones. Although we did not manage to create realistic workloads for 2 and 4GB of small files, we expect the wait times to remain similar to the 1GB workload. Meanwhile the wait times of big files are slightly shorter for the mixed workloads compared to the big files only workloads. This is expected, since there will be less waiting for worker launches because in the mixed workloads the small files contribute to more workers initially launching.

Charged-costs

We also explored the effects of limiting the amount of worker instances. This was mainly done to examine whether there would be a difference in charged-costs, but also to gain better

insight into the quality of service difference. For this experiment we only considered the 4GB mixed workload and capped the worker instance amount to five.

With regards to charged-cost of the EC2 worker instances for this workload we measured it by summing the total run-times of each worker instance and multiplied it with the lease-time cost of 10 cents/hr. Our experiments yielded interesting results: The charged-cost of running the workload with maximum 5 instances was 10.06 cents, but without the worker amount limit it was 11.04 cents.

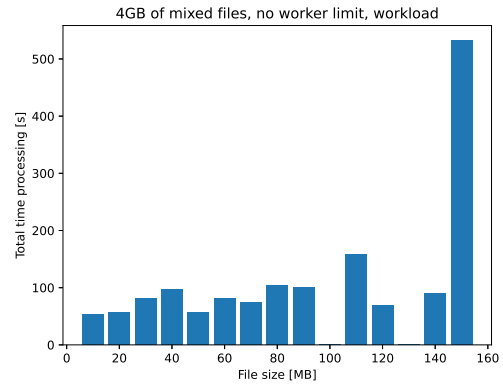


Figure 4: Avg. processing time for file sizes, no worker limit

Total time processing is defined as the time from when a file is registered into the DynamoDB table until it has finished processing. From figures 4 and 5 it is clear that imposing a limit on the scalability degrades the quality of service.

Furthermore, making use of Amazon’s CloudWatch we also looked into the CPU utilization of worker instances. However, as many of the files being processed took less time than the five minute granularity needed with CloudWatch, most workers terminated before being able to obtain data on the CPU utilization. Despite this, from the longer jobs, which took over five minutes, we saw that the CPU utilization during



Figure 5: Avg. processing time for file sizes, maximum 5 workers

video processing was always above 99%. Thus, it is safe to assume this is the case on all machines and that the CPU is a performance bottleneck.

Reliability

The system needs to be reliable as machine failures can occur. Every 10 seconds, the master instance sends out heartbeat requests to every worker. Workers that respond with a state that is neither `pending` nor `running` is considered malfunctioning and gets terminated. Following that action, the job assigned to the terminated worker gets updated in the DynamoDB table so that it is available to other workers.

This functionality was simply tested by stopping a worker instance while it was processing a file. The master instance handled the event as expected, and a replacement worker took care of the file.

CONCLUSION

The video file conversion system presented in this system is shown to satisfy all requirements made by WantCloud BV. Automation is achieved through the use of several AWS building blocks and libraries. Moreover, this also allows for elasticity, which we demonstrated in our experiments. With regards to performance, we show that our scaling algorithm ensures that smaller files do not get blocked behind big files. Additionally, despite it being slightly more economical to impose a limit on scaling it will result in significantly worse processing times of most types of files. Furthermore regarding monitoring, Amazon's CloudWatch is not suitable for very elastic systems like ours, thus we did not explore various metrics deeply. However, from monitoring, via CloudWatch, the CPU utilization was constantly found to be over 99% for bigger files. Lastly, our system's design for reliability manages to detect malfunctions and handle those failures.

REFERENCES

- [1] Boto3, AWS SDK for Python. <https://github.com/boto/boto3>
- [2] DynamoDB. <https://aws.amazon.com/dynamodb/>
- [3] EC2. <https://aws.amazon.com/ec2/>
- [4] FFmpeg command line tool. <https://ffmpeg.org/>
- [5] S3. <https://aws.amazon.com/s3/>
- [6] Amazon. 2006. Amazon Web Services. <https://aws.amazon.com/>
- [7] CloudConvert. 2010. CloudConvert file converter. (2010). <https://cloudconvert.com/>
- [8] Freemake. 2010. Freemake video converter. (2010). https://www.freemake.com/free_video_converter/
- [9] Finn Pierson. 2016. Why Cloud Computing Is So Popular And How It Transforms Business: Articles: Chief Technology Officer. (Sep 2016). <https://channels.theinnovationenterprise.com/articles/why-cloud-computing-is-so-popular-and-how-it-transforms-business>

APPENDIX A

Time spent on project

	Ingimundur	Sjoerd	Wouter	Total
Think-time	12	8	14	34
Dev-time	30	30	28	88
Xp-time	11.75	0	0	11.75
Analysis-time	8	0	0	8
Write-time	5	6	5	16
Wasted-Time	16	18	14	48
Total-time	82.75	62	61	205.75

Table 2: Time spent

Time spent on experiments

total-time	1.5 hours
dev-time	1 hour
setup-time	1.5 hours

Table 3: Time spent on elasticity experiment

total-time	3 hours
dev-time	2 hours
setup-time	2 hours

Table 4: Time spent on performance experiment

total-time	15 minutes
dev-time	15 minutes
setup-time	15 minutes

Table 5: Time spent on reliability experiment