

A Cloud solution for large image storage and processing

Dissertation presented by
David HAVEN

for obtaining the Master's degree in
Computer Science

Supervisor(s)
Benoît MACQ, Sebastien LUGAN

Reader(s)
Damien LEROY, Christopher DE VLEESCHOUWER

Academic year 2016-2017

The interesting thing about Cloud Computing is that we've redefined Cloud Computing to include everything that we already do. I can't think of anything that isn't Cloud Computing with all of these announcements. The computer industry is the only industry that is more fashion-driven than women's fashion. Maybe I'm an idiot, but I have no idea what anyone is talking about. What is it? It's complete gibberish. It's insane. When is this idiocy going to stop?

Larry Ellison, chairman, Oracle, 2008

Contents

Introduction	1
1 Theoretical foundations	3
1.1 In image compression	3
1.1.1 Techniques for compression	3
1.1.2 Pros and cons of lossy versus lossless	7
1.2 Cloud Computing	8
1.2.1 What is Cloud Computing ?	8
1.2.2 An example of Cloud IaaS service	10
1.2.3 Virtualization	11
1.2.4 Assembling the building blocks	11
1.3 Security	13
1.3.1 Security threats	13
1.3.2 Cryptographic algorithms	14
1.3.3 Security in the Cloud	17
2 Implementation	26
2.1 Solution 1	26
2.1.1 Architecture overview	26
2.1.2 subnets and routing	27
2.1.3 Client side	28
2.1.4 Server side	29
2.1.5 Security	33
2.2 Solution 2	35
2.2.1 Backend Architecture	35
2.2.2 Authentication system	36
2.2.3 frontend-backend interaction	38
3 Benchmark	40
3.1 Compression algorithms	40
3.2 Performance benchmark	43
3.2.1 Solution 1	43
3.2.2 Solution 2	46
3.3 Cost estimation	49
3.3.1 Solution 1	49
3.3.2 Solution 2	52
Conclusion	54

Introduction

The context

Telemedicine is the remote delivery of health care services and clinical information using telecommunications technology. This includes a wide array of clinical services using internet, wireless, satellite and telephone media. Within this domain, telepathology can be viewed as the analysis of tissue, cell, and body fluid samples from a distance. A pathologist can make diagnostics by observing via a communication link the medical slides captured by a remote microscope. This particular form of telemedicine has several applications such as providing a primary diagnosis to a site with no pathologist or request a second opinion from a distant colleague for a complex or ambiguous case. In addition, telepathology can serve research or education purposes. Over the years, telepathology systems have continuously advanced and their use are more and more widespread.

The first generation of these systems consists of acquiring the microscope image and sending it to the pathologist. The acquisition can be done either in dynamic mode where live video images are transmitted and viewed dynamically in real-time or in store-and-forward mode where images (typically only a few) are captured and transmitted for sequential viewing. The pathologist can remotely manipulate the microscope to select the slide region to visualize, requiring a new acquisition whenever this region changes. In these systems the operator is always bound to the microscope, so any session requires using this device in an exclusive way.

The second generation of telepathology systems introduces the concept of Whole Slide Imaging WSI and virtual slide. WSI is a technique with 2 components : the creation of digital images (also known as virtual slides) of the entire area of a glass slide, and the viewing of such a large virtual slide using a virtual slide viewer (also known as virtual microscope). Virtual slides are digitizations of complete slides, that are stored in a server for later inspection. A client/server architecture allows pathologists to navigate remotely through the virtual slides, simulating the working of a microscope. Different regions of the virtual slide can be observed, and the zoom can be increased or decreased. Pathologists have in this way the sensation of working with a virtual remote microscope, with the difference that the digitization is made in a previous step on the whole slide. This technique provides the advantage of being fully automated and the resulting file can be easily shared among pathologists. In addition, virtual slides do not degrade over time unlike the real slides.

The process of slide virtualization in pathology involves a series of operations, each contributing to the quality of the final image that is displayed on the computer screen. The operations include pre-imaging steps including sample preparation and staining by a histology laboratory; optical image formation by a virtual slide scanner, digital image sampling by the camera, postprocessing of the digital information, image compression, transmission of the digital image file across a telecommunication network, and finally display of the digital image file on the pathologist's monitor. [1, 4]

The need

To obtain a digitization of the entire slide of a quality which permits a correct diagnosis, it must have, among other features, the appropriate resolution. Complying with this premise is not always possible with the available acquisition device. In many cases its resolution is not enough to cover the entire slide in order to digitize without losing detail. In these cases the slide is divided in contiguous regions such that each region is digitized with the maximum resolution of the device. All the resulting images are subsequently joined in a stitching process, generating one big image with the necessary detail. As could be expected, this results in file sizes of several gigabytes.

The question that needs to be answered is how to store and transmit such large files. Storage at scale on a local computer is unfeasible due to limited storage capacity. The focus of this thesis will be to develop a solution for remote storage of such images using Cloud Computing resources and find a trade-off between local and remote processing of the images. Given the large file sizes, image compression is necessary and we will study the algorithms available to handle such a use case. It will also be important to take into account the security aspect of the solution given the confidentiality that is required for such data. The features we will be focusing on implementing are the following :

1. The software should have an authentication system to control who has access to the core functionalities;
2. An authenticated user should be able to view a list of all images he has access to;
3. An authenticated user should be able to upload an image to the Cloud;
4. An authenticated user should be able to download an image from the Cloud.

Structure of the thesis

This thesis is structured in the following way : the first chapter will be focused on reviewing some theoretical concepts that are necessary for the comprehension of what we implemented for this thesis. These theoretical concepts can be grouped into three categories : image compression concepts, security concepts and Cloud Computing concepts. The next chapter will detail two architectures and we will see how they answer the need that was formulated previously. We will then further complement this presentation by in-depth performance benchmarks of both solutions to identify advantages and disadvantages of both solutions, potential improvements and their cost.

Chapter 1

Theoretical foundations

1.1 In image compression

There currently exists a vast amount of algorithms for compressing images. Instead of giving an in-depth explanation of just one or two of them, we will focus on the different approaches and techniques used in image compression that justify the existence of such a wide range of algorithms. We will then compare the advantages and disadvantages of lossless compression versus lossy compression.

1.1.1 Techniques for compression

Color space transformation

Compression algorithms, either lossless or lossy often begin with some sort of color space transformation and downsampling. The color space transformation aims at transforming the initial RGB pixel values to another format more suited for compression. For instance, JPEG transforms from RGB format to $YCbCr$ format. This allows isolating in separate channels the luminance component ("brightness") from the chrominance components ("color components"). This is relevant because the human eye is more sensitive to the brightness information than the other components. This means that chrominance can be downsampled without any significant effect on the human perception of the image while the luminance remains intact. Figure 1.1 illustrates the different types of downsampling that can be applied to the chrominance channels.

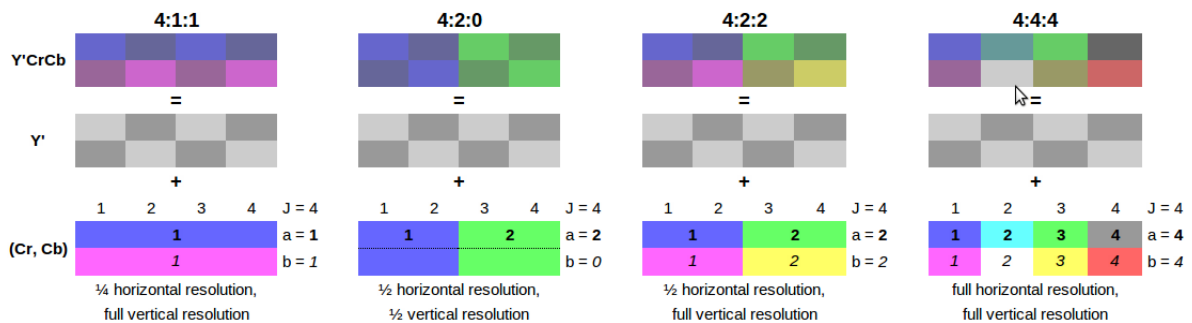


Figure 1.1: Chrominance subsampling

Input space transformation

All compression algorithms contain at some point some sort of input space transformation. The purpose of this operation is to transform the input matrix of pixels into a new matrix that is

more appropriate for compression. We will explain what we mean by more appropriate with the following examples of such input space transformation :

- **DCT** : A Discrete Cosine Transform is a specific type of Fourier Transform, i.e. a representation of discrete values as a sum of cosine functions with different frequencies. Graphically, any image can be seen as a contribution of each of the DCT coefficients in figure 1.2. Most of the information following such a transformation will be contained in the top left corner and adjacent coefficients. This means that the information close to the bottom right corner can be safely discarded without any major loss of information as it does not significantly change our perception of the image.

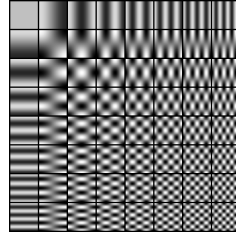


Figure 1.2: DCT coefficients

- **DWT** : This algorithm decomposes a two-dimensional array of pixels into multiple levels of resolution using a wavelet transform. The most commonly used set of discrete wavelet transforms are the Daubechies wavelets (used in the implementation of the JPEG2000 specification [5]). These decomposition levels are made up of subbands of coefficients that describe the spatial frequency characteristics of local areas of the image both horizontally and vertically. The DWT is performed by applying a low-pass filter and high-pass filter as many times as the amount of decomposition levels we want. The main difference between the wavelet transform and the cosine tranform used in the DCT is that the wavelet transform captures time and frequency information whereas the cosine transform captures only frequency information.

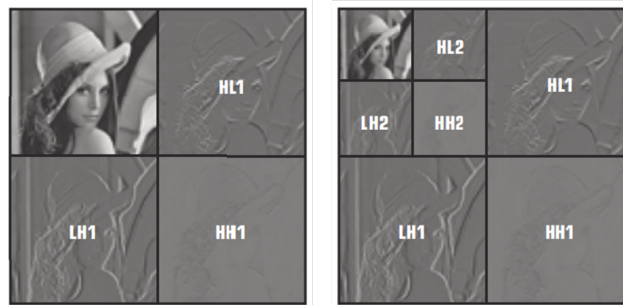


Figure 1.3: DWT decomposition [29]

encoding

The encoding step generally refers to the transformation of the matrix of values representing an image to an output compressed bit stream by associating a code word to each symbol. It is optimized to be as short as possible to improve the compression performances. It is generally one of the last steps of most image compression algorithms. The following is a list of some of those encoding algorithms :

- **Huffman encoding** : The Huffman encoding aims at encoding each symbol separately as a sequence of bits such that the code of one symbol isn't the prefix of any other symbol

and the length of the codes of the symbols that have a higher frequency of occurrence is shorter than the others. It is used in many compression algorithms such as JPEG. In figure 1.4 we can see that words with higher frequency such as "a1" are encoded in less bits.

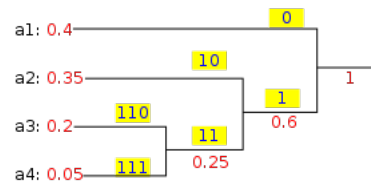


Figure 1.4: Huffman encoding

- **Arithmetic encoding** : As opposed to the Huffman encoding where each symbol is encoded separately, the arithmetic encoding finds a code for the whole input message. The algorithm starts with a range $[0,1]$ divided into N blocks (each corresponding to a symbol) of different size where the size of the block is the probability corresponding to that symbol. The algorithm then iteratively expands a certain block corresponding to the symbol that needs to be encoded and divides that block in the same way as it did for the interval $[0,1]$. Once there is no more input to be read, the last interval to be expanded can be encoded because it unambiguously identifies the sequence of symbols that were read. More precisely, it is only necessary to encode as many bits as required so that all values in that interval have the same prefix. In terms of performance, the arithmetic encoding is much closer to the optimal entropy encoding than Huffman encoding. Arithmetic encoding is used in most recent compression algorithm such as JPEG2000 or WebP.

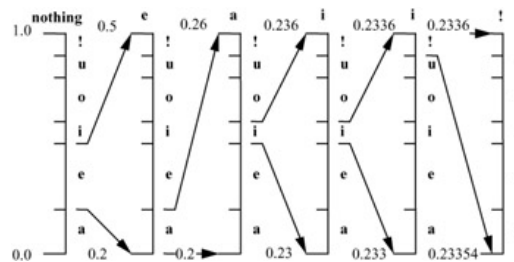


Figure 1.5: Arithmetic encoding

- **run-length encoding** : The run-length encoding is often used with other encoding algorithms such as Huffman encoding and aims at shortening the length of the bit-stream by substituting multiple consecutive occurrence of a symbol by one unique occurrence of the symbol followed by the number of times it was repeated. In jpeg, a run-length encoding is applied right before the Huffman encoding.
- **LZ77** : The LZ77 algorithm is similar to the RLE but more complex in the sense that instead of compressing adjacent occurrences of a given symbol, it attempts to compress patterns of different symbols that tend to appear multiple times. The algorithm maintains a sliding window of the stream to be encoded and tries to find the longest match at the beginning of the window with the previous encoded stream. If it finds a match, it encodes a pair (distance, length) which indicates at what distance the pointer needs to be placed from it's current location and how many characters there are in the match. The LZ77 algorithm is used in many compression algorithms such as PNG or GIF.

Intra-frame prediction

This technique is based on the assumption that pixels that lie close to each other often have similar values. This technique is used by Google in its image format called WebP where a certain number of key areas will be fully encoded. For adjacent blocks only the residual, that is the information that is not redundant, will be encoded. The format also needs to encode the prediction mode that was used in order to be able to recover the information when the image will be decompressed. Figure 1.6 shows the different prediction modes that WebP utilizes in order to obtain a block of pixels from the first line and column. [13]

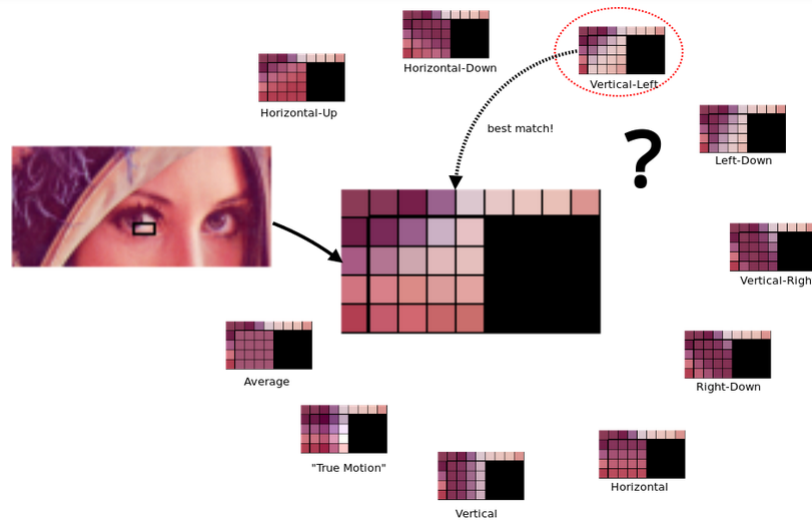


Figure 1.6: webp prediction modes

1.1.2 Pros and cons of lossy versus lossless

Compression artifacts

The reduced file size of lossy compression comes at a price, i.e. the whole information of the initial image cannot be retrieved when decompressing. Usually the impact is unimportant as the loss of information happens in a way that the human eye can not perceive. When too much compression is done or when the compression is not performed in an intelligent fashion, the loss of quality can be seen and/or visual artifacts appear. Some of these artifacts include :

- **Block boundary artifacts** : This type of artifact is a result of the fact that many lossy compression algorithms break up an image into smaller blocks and apply the compression to each individual block. This can lead to discontinuities at the block boundaries between adjacent blocks that is visible in the final image. In the following image, we can clearly see the boundaries of each block.



Figure 1.7: block boundaries artifacts

- **Ringing artifacts** : These artifacts tend to appear in an image where there is a sharp discontinuity in the signal such as shape borders. Ringing artifacts are also particularly visible in text. This is a consequence of the transformation of the input to a frequency domain both in DCT and DWT where the high frequency components are generally discarded given that the human eye is more sensitive to low frequencies modifications (color gradient on a large surface) than high frequencies modifications (sharp brightness variations i.e. contours). Figure 1.8 and 1.9 illustrate this [14].

Lossless compression on the other hand as the name suggests has the advantage of keeping all of the initial image information once it is compressed. However, the drawback is that the compression ratio will be inferior to lossy algorithms.



Figure 1.8: original image



Figure 1.9: compressed image

1.2 Cloud Computing

1.2.1 What is Cloud Computing ?

When talking about the Cloud, people often associate it with the services available through the internet. The well-known "Software as a Service" such as Dropbox (storage service), Gmail (e-mail service), Odoo (ERP service), etc. This is true but is only part of the global picture. To be more precise the SaaS is the "highest level of abstraction" on the scale of the Cloud offerings. By abstraction, we mean that the user has little or no knowledge of what is going on under the hood and about the inner mechanisms of the service. Lower levels include "Platform as a Service" and "Infrastructure as a Service". While the PaaS framework gives users the possibility to develop applications without prior knowledge of the underlying infrastructure (Heroku apps and Google App Engine are good examples of such a service), the IaaS allows users to interact with the "lowest levels" of the infrastructure (quotation marks used since even here there is some level of abstraction as we will see later) such as the network (IP address, routing, etc.) and the hardware (software stack, instance types, etc.).

For this thesis, our main use of the Cloud will be through IaaS. Let us now look at a definition of Cloud Computing given by the NIST (National Institute of Standards and Technology) and try to extract from it some key characteristics :

"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."

From this definition, we can extract three key characteristics :

- on-demand access to computing resources;
- rapidly provisioned and released;
- minimal service provider interaction.

In addition to this we can also note the following characteristics [8] :

- the illusion of infinite computing resources;
- no upfront commitment (for IaaS);
- a pay-as-you-go pricing model.

Together, these characteristics provide flexibility to the cloud user when building a solution and the possibility to start small and grow your infrastructure steadily as the business grows. Let us look at an example. When you work with your own infrastructure and you plan on delivering a web application, you need to decide on the amount of physical servers you will need to sustain the requests from the users. As shown in figure 1.10, overprovisioning means having too much hardware that are not fully utilized which results in loss of profit. On the other hand, underprovisioning (figure 1.11) means not having enough hardware which results in dissatisfied customers because your service will not function properly when all servers are saturated.

Following the demand is quite a complicated task if you use your own infrastructure given that purchasing, installing and deploying a server takes time. The cloud IaaS solves this by "the

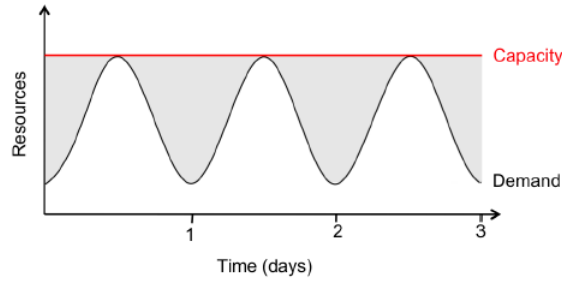


Figure 1.10: Overprovisioning example [8]

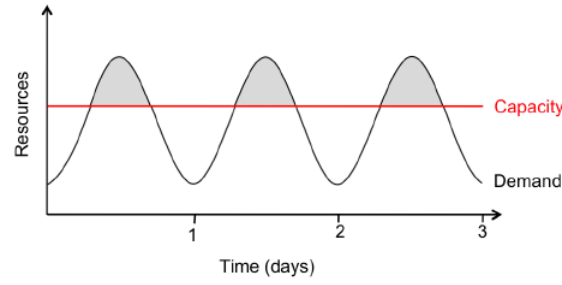


Figure 1.11: underprovisioning example [8]

rapid on-demand provisioning and release of computing resources" which allows to "follow the curve" (figure 1.12) and always have just the amount of servers that are needed.

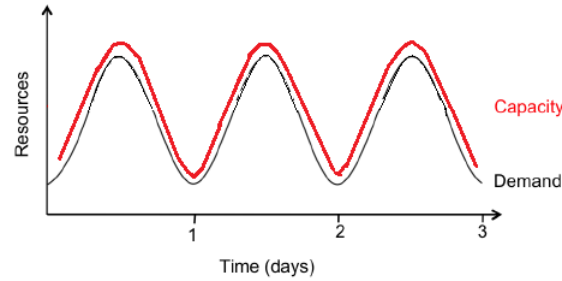


Figure 1.12: flexible provisioning example

This model is also interesting from a cost viewpoint. Investing in servers can be a big upfront commitment that might be inefficient in case of underutilization or saturation. Cloud infrastructure on the other hand provides a model where you pay only for what you use and for as long as you use it (usually by units of hour).

In summary, just like the power industry went from local and limited production using infrastructure such as watermills and windmills to larger centralized power plants where the production was distributed to users and billing was based on consumption, we can observe a similar shift in the computing industry where Cloud Computing offloads "computational tasks" such as provisioning servers, to a very large datacentre that provides metered usage for computing. This allows the Cloud users to focus on providing the best service for their customers. The following sections will illustrate the various advantages of Cloud Computing by providing concrete examples.

1.2.2 An example of Cloud IaaS service

Let's take a closer look at one specific service offered by the Amazon Cloud [15] (further denoted as Amazon Web Services or AWS for short) Elastic Cloud Compute (EC2). Amazon EC2's are virtual computers that can be used to run any type of software. There are 5 different categories of EC2 instances [16] :

- **On-demand instances** : Those are the "standard" instances. You pay for compute capacity by the hour with no long-term commitments or upfront payments. You can rapidly launch or stop instances, add them to a Cloud network and install any software you want.
- **Spot instances** : These are instances on which you can bid a specific price. The price for those instances changes in real time and is generally lower than on-demand instances. When the price of the instance is lower than your bidding price, the spot instances will be launched and run for as long the spot price equals or exceeds your bidding price.
- **Reserved instances** : Similar to on-demand instances except that you must reserve the instance for a longer period (usually 12 months). The advantage is that the price will be lower than on-demand instances for the same period by up to 75%.
- **Dedicated hosts** : this is a physical EC2 server dedicated for your use (it is not virtualized and shared among multiple users). Thus, you have full control over the hardware and can import your own licenses (such as the operating system licenses).
- **Scheduled instances** : These are a special case of reserved instances where you purchase the instance for a long period of time (generally 12 months). However, the instance doesn't run 24/7 but only during specific periods of the day and you are billed only for those periods.

We will focus on the on-demand instances since they will be used for this thesis. The following table shows the pricing for different size of EC2 on-demand linux instances (non-exhaustive list).

Name	vCPU	ECU	Memory (GiB)	Instance Storage (GB)	Linux/UNIX usage
t2.micro	1	variable	1	EBS-only	\$0.013 per Hour
t2.medium	2	variable	4	EBS-only	\$0.05 per Hour
t2.xlarge	4	variable	16	EBS-only	\$0.202 per Hour
m4.large	2	6.5	8	EBS-only	\$0.119 per Hour
m4.10xlarge	40	124.5	160	EBS-only	\$2.377 per Hour

These are only a few of the EC2 instances available. Nonetheless they give a good idea of the wide range of computational power that is available. The column "Instance storage" with value "EBS-only" means that a hard drive (Elastic Block Storage is the Amazon terminology) must be attached to the instance in order to have persistency. The ECU column stands for EC2 compute Unit which is an abstraction of the CPU capacity offered by the instance. This is necessary to give a consistent view of the underlying hardware which might not always be the same for a given EC2 instance type due to virtualization. This process is explained in the following section.

1.2.3 Virtualization

As we can see from the previous presentation even though EC2 has the lowest level of abstraction on the scale of Cloud services (or the highest level of control is another way to phrase it), there is still some level of abstraction (cpu abstraction, network abstraction, etc.) due to the virtualization Amazon uses in order to optimize its hardware utilization. The virtualization process is the following : the first layer above the hardware that is in charge of the virtualization is a highly modified version of the Xen hypervisor. This hypervisor manages the CPU scheduling and memory management between the different instances. Amazon instances (referred to as Amazon Machine Image or AMI for short) can be launched in one of two virtualization mode [12] :

1. **Hardware Virtual Machine** : "HVM guests are fully virtualized. It means that the VMs running on top of their hypervisors are not aware that they are sharing processing time with other clients on the same hardware. The host should have the capability to emulate underlying hardware for each of its guest machines. This virtualization type provides the ability to run an operating system directly on top of a virtual machine without any modification — as if it were run on the bare-metal hardware. The advantage of this is that HVMs can use hardware extensions which provide very fast access to underlying hardware on the host system."
2. **Paravirtual** : "Paravirtualization, on the other hand, is a lighter form of virtualization. This technique is fast, and provides near native speed in comparison to full virtualization. With Paravirtualization, the guest operating system requires some modification before everything can work. These modifications allow the hypervisor to export a modified version of the underlying hardware to the VMs, allowing them near-native performance. All PV machines running on a hypervisor are basically modified operating systems like Solaris or various Linux distributions"

Historically, paravirtualization provided the best performance and HVM was slow due to its full virtualization process. In recent years, HVM has closed the gap on PV and Amazon now recommends using HVM.

1.2.4 Assembling the building blocks

An Amazon EC2 instance can be viewed as the building block in the process of creating a Cloud infrastructure. The last two sections illustrated how EC2 fits our view of Cloud Computing by allowing flexibility by rapid provisioning and release (launching a new instance takes less than 2 minutes) and "pay as you go" billing system to minimize the monetary investment needed in order to get started. We will now add a few concepts that will allow EC2 instances to be combined in order to form a network that can scale given some criterion :

- **Auto-Scaling** : This is the process of defining an auto-scaling policy which is a criterion based on which an instance will be automatically launched or stopped. For example you might deploy a web application that is managed by 2 servers and decide that the policy for launching a new instance is whenever the CPU load goes above 80 % and the policy for removing an instance is whenever the CPU load goes below 40 %. You can also decide on a minimum and maximum number of instances that must be running [17].
- **Elastic load balancer (ELB)** : Having multiple instance running is only useful if they can be reached. The ELB manages this task by distributing the inbound traffic evenly to the different instances. The following image summarizes this. ELB also achieves fault-tolerance by monitoring the health of the instances and routing traffic to healthy instances [18].

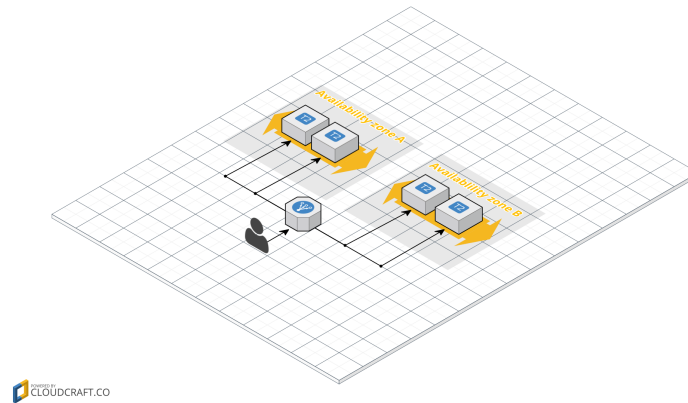


Figure 1.13: basic cloud architecture : ELB + auto-scaling

- **Virtual Private Cloud (VPC)** : The different pieces of software and hardware described previously can be grouped into an isolated network called a VPC. The User has complete control over the virtual networking environment, including selection of the IP address range, creation of subnets (groups of instances that share the same configurations), and configuration of route tables and network gateways [19].

Using these tools allows us to build a robust and flexible infrastructure. This presentation of Cloud Computing is a brief overview of the core concepts. Many more tools and services offered by the Cloud will be presented during this thesis as they become relevant.

1.3 Security

In this section we will take a look at some well-known security threats that appear when building a network. We will then see how we can address these threats, both by using security protocols and algorithms and by applying best practices when it comes to securing the Cloud.

The basic aspects of information security of concern are :

1. **Confidentiality** : Who can read the data either stored or in transit.
2. **Integrity** : Who can manipulate the data while it is transmitted over a channel or while it is stored. Compromising the integrity of data is known as tampering.
3. **Authentication** : Who can access our data.

1.3.1 Security threats

Low-level attacks :

- **Packet sniffing/ eavesdropping** : Intercept packets at the physical/link-layer level.
- **Spoofing** : Sending a packet with a modified source address over a network. It can be combined with eavesdropping in order to replicate a specific packet structure. Spoofing can be used to communicate at the IP-level between a forged source and a destination. Different types of spoofing exist such as IP-spoofing, ARP-spoofing or DNS-spoofing
- **TCP disruption** : If an attacker knows the ports and the sequence numbers used in the TCP connection, he can disrupt the TCP connection by sending a packet with the RST flag set which force the connection to end. This technique can also be used to inject data (session hijacking).

These attacks can affect all 3 security threats. For instance, eavesdropping (confidentiality) can be used to construct a spoofing attack (integrity) which can be used to perform TCP disruption (availability).

Some other common attacks :

- **Man-in-the-middle attack** : A man-in-the-middle attack is an attack where the attacker relays and possibly modifies messages sent between 2 parties. Both parties think they are communicating with each other while in reality they are communicating with the attacker. This type of attack can be prevented by means of authentication such as certificate validation
- **Birthday attack** : The name comes from the analogy that in a room of 23 persons, the likelihood of two of them having the same birthday is approximately 50%. This attack tries to find a collision with a specific message digest by generating a specific number of random messages such that the probability of having a collision with the target digest is $1/2$. If no collision is found, a new set of random messages is generated and the algorithm starts over again. It is possible to render this technique computationally infeasible by having a long digest length. In the case of SHA256 there are 2^{256} different digests. The relation that gives us the number of messages to generate in order to expect a collision with probability $1/2$ is given by : $n = 1/2 \times B^{1/2}$ where n is the number of messages to generate and B is the number of messages digest that exist. The solution to this for $B = 2^{256}$ is $n = 1.7 \times 10^{38}$ which isn't feasible from a time complexity or space complexity view point.

- **Length-extension attack** : The length extension attack is another type of attack on hash functions which, as opposed to the birthday attack, doesn't try to find a collision. Instead, this attack is used in the context of the signing process of a message by trying to include some additional information in the message while still having a valid signature. Basic signing techniques susceptible to this attack include functions such as $H(secret || message)$. Other more refined constructions such as HMAC are not susceptible to this attack. This attack is done by taking into account the fact that the hash value is the internal state of the hash function at a given point in time. By appending some information to the message and "resuming" the hash function, a new valid hash value will be computed even though the secret message isn't known.
- **Replay-attack** : This attack attempts to intercept a message and later retransmit it. For example while logging in to a website account, an eavesdropper might capture the session and try to replay it in order to gain access to the account. Replay attacks can be prevented by using session ids that identify a session between 2 actors as being unique or by using time stamp where a server will reject a message that arrives within a delay that is too long.

1.3.2 Cryptographic algorithms

We will now present a few cryptographic algorithms that can be used to guarantee the security of private information with respect to the three main security concerns mentioned above : "authentication, confidentiality, integrity".

Certificates

A certificate is a signed statement about someone's public key. It is a basic component in the process of establishing a secure communication channel between a client and a server. The idea is that the signing process of a public key provides a guarantee that the public key belongs to a trusted party. For this to be valid we need to trust the person signing the key. This is the role of the Certification Authority. CA's are companies that you trust to exercise prudence when signing people's keys. The standard for certificates is X.509. Such certificates contain 3 required fields [20] :

- **TBS certificate** : The useful payload of the certificate
- **CA signature algorithm** : Identifier for the crypto algorithm used by the CA to sign this certificate.
- **CA signature value** : Signature of the certificate by the CA. This is most commonly done by using a hash function such as SHA2 on the preceding fields and then encrypting the result using the CA's private key.

When a client wants to communicate with a server in a secure way, it can validate the server's certificate in order to make sure that the server is who it says it is. In order to do this, the client first checks if the domain name in the certificate matches the URL. The browser then accesses another certificate, referred to as the root certificate that is stored within the browser in order to get the CA public key. Then the client applies the CA's public key to retrieve the hash of the certificate. It then compares it with a hash the client has computed over the certificate. If there is a match, the server can be trusted.

SHA-256

The SHA-256 algorithm is one of many algorithms from the family of Secure Hash Algorithms. Its role is to take as input a message of length $< 2^{64}$ and return an output called a message

digest of length 256 bits. Secure hash algorithms are typically used with other cryptographic algorithms, such as digital signature algorithms or in the generation of random numbers. SHA algorithms must respect the following properties :

1. It should be impossible to deduce the message given a message digest.
2. It should not occur that the hash function returns the same digest for two different messages.
3. The digest should be indistinguishable from a random n-bit value.

The SHA256 algorithm executes the following preliminary steps before executing the computations [6]:

1. The message is padded in order that its total length is a multiple of 512. The hash function will then process the message as blocks of 512 bits. Each of these blocks can itself be decomposed in 16 32-bit words $M(i)0, M(i)1, \dots, M(i)15$ where i is the block index.
2. SHA256 uses a set of 6 logical functions where each function operates on 32-bit words. The following are examples of such functions : (insert math function).
3. SHA256 uses 64 constant 32-bit words. These words represent the first thirty-two bits of the fractional parts of the cube roots of the first sixty-four prime numbers. They are denoted as $K0, K1, \dots, K63$

The algorithm then starts its core processing. It is composed of an outer loop of N iterations where N is the number of 512 bit blocks in the initial message. Each of these outer loop aims at updating the hash value which is composed of 8 32-bit words $H(i)0, H(i)1, \dots, H(i)7$. And an inner loop of 64 iterations. The following picture illustrates one of those inner loops. Each of the working variables $a, b, c, \dots, h$ at iteration " i " is initialized to the hash value $h(i-1)0, h(i-1)1, \dots, h(i-1)7$. The algorithm applies multiple transformation to those working variables by using the logical functions (blue boxes), modulo 2^{32} additions (red boxes), the constants $K0, \dots, K63$ and the messages $W0, \dots, W63$ which are transformations of the initial message $M(i)0, M(i)1, \dots, M(i)15$.

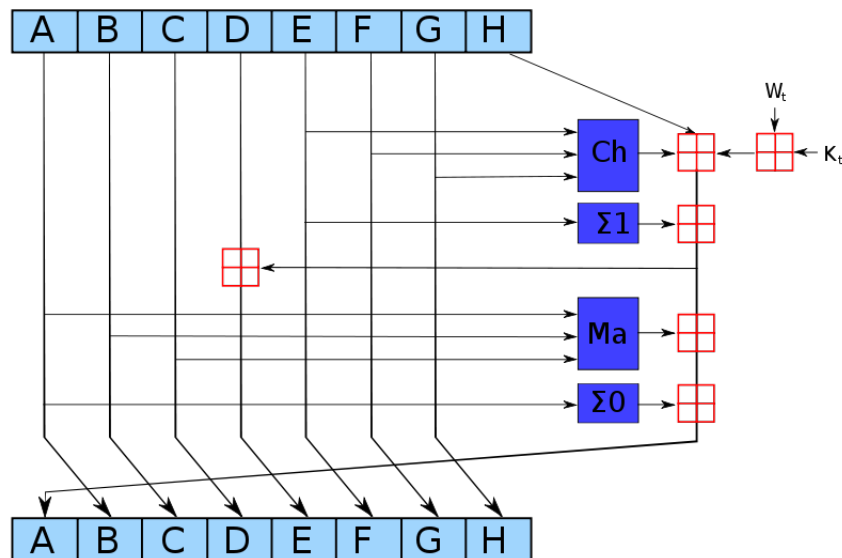


Figure 1.14: 1 step of the SHA256 algorithm

HMAC

HMAC is a method for computing a keyed MAC (message authentication code) using a cryptographic hash function. The cryptographic strength of HMAC depends on the properties of the underlying hash function. The purpose of the HMAC algorithm is to simultaneously verify the integrity and authentication of a message. The HMAC algorithm does the following computation [7] :

$$H((K \oplus (opad) \parallel H(K \oplus (ipad) \parallel text)))$$

H is the hash function,

K is the authentication key,

ipad is the byte 0x36 repeated B times,

opad is the byte 0x5C repeated B times,

B is the block length of the hash algorithm

$\oplus$ is the logical XOR function

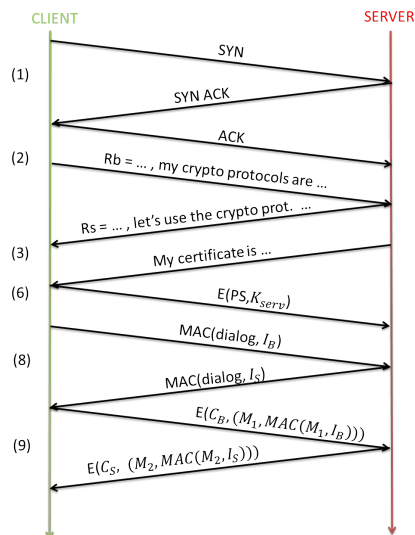
$\parallel$ appends it's right-hand side to it's left-hand side

Once the MAC is computed by the sender, it is sent along with the message to the receiver who recomputes MAC on the message. If it matches the one sent by the sender, then we can be sure that the message has not been altered and that it has been sent by a valid user (because the user shares the same key as the receiver). The choice of a random (or cryptographically pseudorandom) keys, a secure key exchange mechanism, frequent key refreshments, and good secrecy protection of keys are all essential ingredients for the security of the integrity verification mechanism provided by HMAC.

The HMAC provides additional security over other MAC computations such as $MAC = H(key \parallel message)$ which are more sensitive to length-extension attacks.

HTTPS

HTTP(S) is an application layer protocol for secure communication over the internet. It is based on an underlying encrypted connection provided by TLS or it's predecessor SSL. HTTPS provides security against eavesdropping, tampering, man-in-the-middle attacks as well as authenticating the remote server to which a client is connecting. The establishment of a secure communication channel with SSL/TLS follows the following steps [28] :



1. The browser connects the HTTPS server via TCP
2. The client picks a 256-bit random number R_B and sends it over to the server with a list of cryptographic protocols it supports.
3. The server picks a 256-bit random number R_S and agrees on the crypto protocols that will be used during the session. It sends it over to the client with it's certificate.
4. The client validates the certificate
5. Depending on the encryption algorithms chosen at step 3, the following steps can vary. If we assume that the RSA algorithm was chosen the following applies.

6. The browser constructs a long (368 bits) "Premaster Secret" PS and sends it to the server using the server's public RSA key K_{Server}
7. Using PS, R_B , and R_S , browser and server derive symmetric cipher keys (C_B, C_S) and MAC integrity keys (I_B, I_S) , one pair to use in each direction.
8. Browser and server exchange MACs computed over entire dialog so far.
9. If the MACs are validated, the browser displays a lock. All further communication are encrypted using the symmetric keys and the MAC integrity keys.

1.3.3 Security in the Cloud

Shared responsibility model

When it comes to security and data privacy in the Cloud, people often view it as being highly unreliable and untrustworthy. Two things are worth mentioning here.

Firstly using the Cloud infrastructure to build a solution doesn't imply that you lose all privacy over the data that might be processed. We have seen earlier that the Cloud can be viewed as a spectrum where different cloud services are offered at different levels of abstraction (ranging from IaaS to SaaS). When it comes to security, the SaaS provides the least control to the user about how his data is processed and secured and the user must trust the SaaS provider that his data will be correctly managed. On the other end of the spectrum, IaaS which is the focus of this thesis offers the most flexibility and control over how secure you want your data to be.

This leads us to our second point. The security responsibility is shared between the Cloud provider and the Cloud user. The Cloud provider provides secure low-level infrastructure and security features and it is up to the user to take advantage of those features in order to build a solution where the security meets the standards that are expected for the particular task at hand. Figure 1.15 shows how Amazon Web services are responsible for the low level security (the global infrastructure that runs all of the services offered in the AWS Cloud) and the Cloud user is responsible for the higher level security (softwares installed on the instances, firewall, authentication, etc).

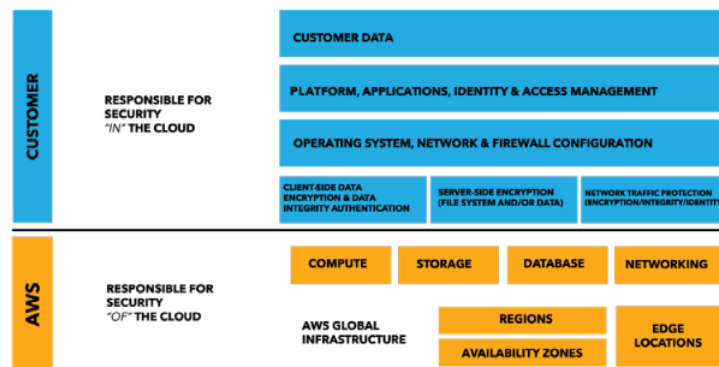


Figure 1.15: AWS Shared Security Responsibility Model [10]

AWS Global Infrastructure security

The AWS Global Infrastructure is composed of regions, availability zones and endpoints. Regions are geographical locations where Amazon has built a datacentre. The choice of the region where a Cloud user decides to deploy a Cloud solution affects latency and regulatory compliance. Regions can be further broken down into availability zones which are fault isolated. Availability zones are connected to multiple Internet Service Providers and power grids and can communicate with each other within a region through high speed LAN links. It is a best practice for fault tolerance to use multiple availability zones when building a solution on the Cloud. This infrastructure meets several security compliance standards. The following is a non-exhaustive list of some of those standards [21] :

- **ISO27001** : Security management standard that specifies security management best practices and comprehensive security controls. The basis of this certification is the development and implementation of a rigorous security program, which includes the development and implementation of an Information Security Management System (ISMS)
- **ISO27018** : It is a code of practice that focuses on protection of personal data in the cloud.
- **PCI DSS level 1** : It is a proprietary information security standard that applies to all entities that store, process or transmit cardholder data (CHD) and/or sensitive authentication data (SAD).

AWS is also aligned with specific laws and regulations when it comes to private data protection. The most relevant for the EU is probably the EU data protection directive also known as Directive 95/46/EC. In addition to those standards, Amazon follows strict controls when it comes to physical access of the datacentre, electrical power management, fire prevention and temperature control [9].



Figure 1.16: AWS regions and availability zones

Account security features

The most important security measure that can be taken by the user point of view is controlling who has access to the AWS account and therefore to the Cloud resources. As with most web applications, accessing the AWS control panel require authentication by means of a username and password. In addition, the user can also enable Multi-Factor Authentication (MFA).

Two types of users are allowed to login, the root account which belongs to whoever created the account and the IAM users (Identity and Access Management). IAM users are created by the root account and allow for having additional users to have access to the same account without having to share the root account password. In addition, when creating the IAM users, you can decide on which services they will have access to.

Another way to access the Cloud resources is through the API's. Making API requests such as "launch new instance", "terminate given instance", "create VPC", etc, cannot be performed without access keys. Access keys are another form of credentials composed of an access key id and secret key. These access keys can be downloaded once when the root account is created or whenever a new IAM user is created. These access keys are then used to digitally sign the requests. The digital signing process is done by means of a cryptographic hash function. The input to the hash function is the text of the request and the secret access key. The most recent version of the digital signature calculation process is Signature Version 4 which calculates the signature using the HMAC-SHA256 protocol. Version 4 provides additional security over previous versions by not using the secret key directly in the hash function but using a key that is derived from the secret key.

In addition, sometimes you might want one of AWS services to perform API requests on your behalf. For example you might want your instances to be able to perform API requests to Amazon S3 in order to store or retrieve some data. In this case, Amazon provides the concept of "role" that can be associated to a particular cloud resource and allows it to retrieve temporary access keys in order to perform the API requests [10].

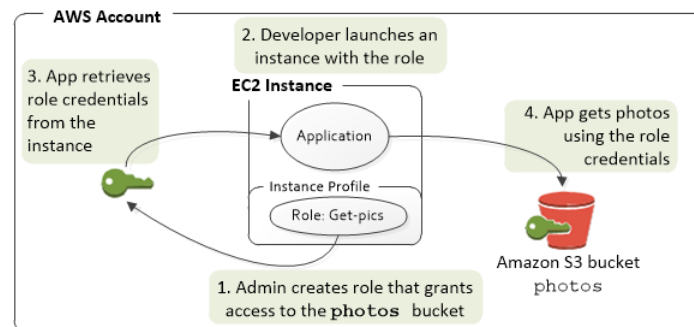


Figure 1.17: Process of retrieving temporary credentials

EC2 and EBS security

Access to an EC2 instance is controlled by means of a firewall (referred to as Security Groups in the Amazon documentation). By default, the firewall blocks all inbound connections and it is up to the user to open specific ports. For example, for an EC2 instance that runs a server you might want to open port 443 for inbound HTTPS connections, port 80 for HTTP connections and port 22 for SSH access when you want to debug the server or install additional softwares. You can also decide on what instances can be reached with which security groups.

As discussed previously EC2 instances are virtualized which means 2 users might have their instances running on the same hardware. Given this premise, it is important that EC2 instances are correctly isolated from each other. This is one of the roles of the hypervisor. The AWS firewall is located in the hypervisor layer which means that all packets that reach any instance must go through this layer. This implies that other instances running on the same hardware have no privilege compared to any other hardware located on the internet. In addition, AWS customers don't have access to raw disks which are virtualized. Every block of storage is automatically

reset when deallocated by the user. The same goes for the memory allocated to the user. Given that the instance, once it is running, is completely controlled by the customer, it is up to him to add additional security layers such as a certificate based SSHv2 to access the instance.

Type ⓘ	Protocol ⓘ	Port Range ⓘ	Source ⓘ
HTTP	TCP	80	0.0.0.0/0
SSH	TCP	22	0.0.0.0/0
HTTPS	TCP	443	0.0.0.0/0

Figure 1.18: Example of inbound security rules

Network security

As we have seen in the previous section, a common pattern is to put the EC2 instances inside a VPC in order to isolate them from the rest of the cloud. This allows for additional security measures and finer grain control. Figure 1.19 illustrates this. Firstly, instances can be grouped into a subnet within an availability zone where all instances within a subnet share common security patterns. In addition, a routing table is used to reach those instances. Subnets and routing tables provide by default protection against layer 2 security attacks such as MAC spoofing and ARP spoofing. Network ACL's add an additional layer of security where one can define a set of rules evaluated both for inbound and outbound traffic. These rules can be used as filters to allow or deny traffic based upon IP protocol, service port or source/destination IP address.

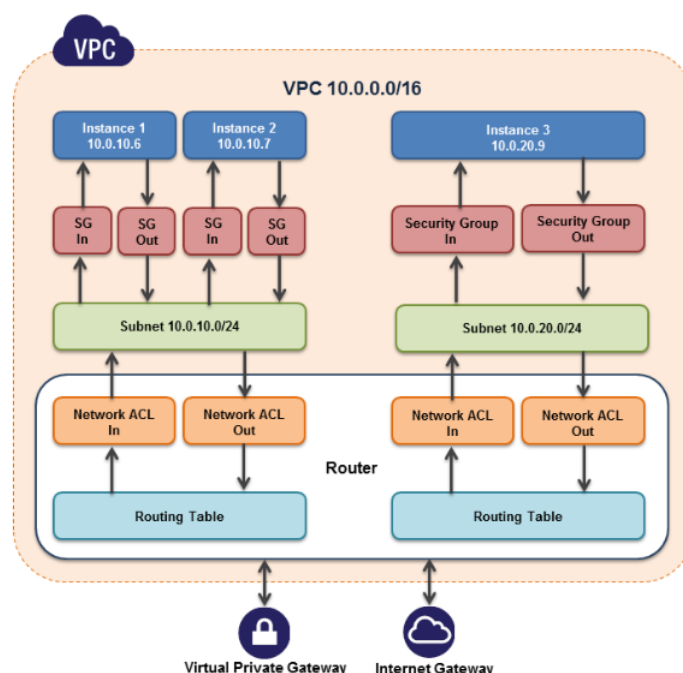


Figure 1.19: Flexible network topology [9]

Protecting data in transit

When talking about data in transit we mean the data when it is sent from a client to servers running in the cloud and data when it is sent from the server to storage locations such as S3 (Simple Storage Service) or RDS (Relational Database Service). The best way to protect data in this case

is by using the HTTPS protocol. As presented earlier, this prevents eavesdropping, tampering and forgery. When using the AWS Management Console (web interface), for example for sending some data such as an image to S3, HTTPS encryption happens by default. Alternatively, data can be sent to the Cloud using the AWS API's which can be accessed via the various SDK's or via the command line. These web services are also provided over HTTPS which means that SSL/TLS sessions are established between the client and the specific AWS service endpoint. This means that once our data has been sent to the servers on the cloud, we can further guarantee that the data can also be sent securely to persistent storage locations such as S3.

Protecting data at rest

For EBS and S3. AWS provides a selection of security features for guaranteeing the security of the data once it is stored on S3 or on EBS. Some of them are implemented by default by AWS and others are given as additional security features that may or may not be used by the user or not. The following table provides an overview of the various security features that are available when storing data in S3 [10] :

Amazon S3 feature	description
Permissions	Access to content stored on S3 can be managed through the permissions functionality. For example if storing a static web site on S3 you might want to make the content available to everyone. For other use cases you might want to make the content available to specific AWS users or to yourself only. The different types of permissions are "create", "delete", "list". By default, access is restricted to the AWS account owner of the bucket that was created
Versioning	AWS S3 supports object versioning. By enabling this functionality, a new version of an object is stored for every modified or deleted object which you can restore if necessary.
Replication	Amazon S3 Standard redundantly stores your objects on multiple devices across multiple facilities in an Amazon S3 Region. The service is designed to sustain concurrent device failures by quickly detecting and repairing any lost redundancy. When processing a request to store data, the service will redundantly store your object across multiple facilities before returning SUCCESS. This does not prevent from accidental deletions because changes are replicated across all availability zones where it stores copies. Amazon S3 also regularly verifies the integrity of data stored using checksums. If corruption is detected, it is repaired using redundant data. In addition, Amazon S3 calculates checksums on all network traffic to detect corruption of data packets when storing or retrieving data.
Encryption-server side	Amazon S3 supports server-side encryption of user data. Server-side encryption is transparent to the end user. In this case, data is encrypted remotely before it is stored on S3 and decrypted when it is accessed.
Encryption-client side	With client-side encryption you create and manage your own encryption keys. Keys you create are not exported to AWS in clear text. Your applications encrypt data before submitting it to Amazon S3, and decrypt data after receiving it from Amazon S3. Data is stored in an encrypted form, with keys and algorithms only known to you.

Let's go into more detail of the client-side encryption (CSE) and server-side encryption (SSE) features. Encryption requires 3 main components : data to encrypt, an encryption process, a Key Management Infrastructure (KMI) for storing the encryption keys. A KMI is typically composed of two subcomponents : the storage layer where the keys are in plain text and a management

layer to control who has access to the keys. A KMI usually takes the form of a Hardware Security Module (HSM) which is an independent piece of hardware designed to be a safeguard against tampering. AWS provides 3 models [12] when it comes to managing these 3 components. Each model presents a different tradeoff between "doing things ourselves" or "delegating to Amazon". The following picture illustrates this.

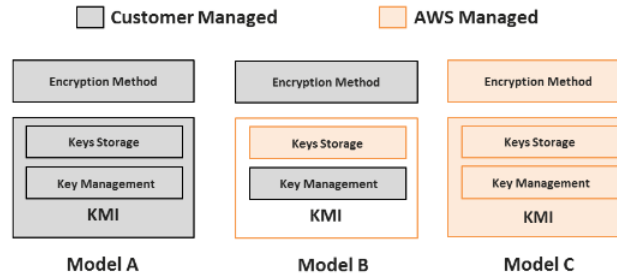


Figure 1.20: Encryption models in AWS [12]

model A : client-managed

The important security property of this model is that the user has full control over the encryption keys and the encryption process which means Amazon cannot perform encryption or decryption on your behalf. The encryption can make use of opensource softwares such as Bouncy Castle or OpenSSL, third-party softwares or AWS SDK's. It is also important to note that the encryption process doesn't have to take place "on premise" but can also take place in the cloud within an AWS EC2 instance for example. The following picture summarizes these different options in the case of storing data on S3.

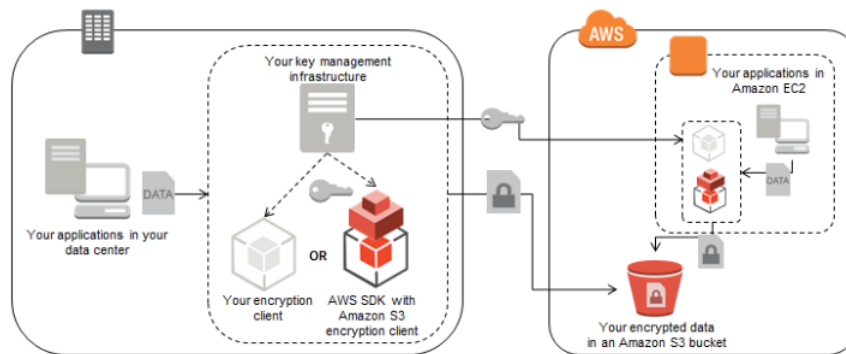


Figure 1.21: Amazon S3 client-side encryption from on-premises system or from within your Amazon EC2 application [12]

model B : shared model

This model is similar to the previous one except that the key storage is managed by Amazon and takes the form of a "CloudHSM". This is a service offered by Amazon where the hardware layer is managed by Amazon and the management layer is managed by the user. This means that it is up to the user to decide who has access to the keys and even though they are stored in the AWS environment, they are inaccessible to an AWS employee. It is important to note that the CloudHSM does not perform key lifecycle management functions such as access control policy or key rotations. This means that an additional piece of software needs to be implemented in order to communicate with the CloudHSM. Because the AWS CloudHSM service uses SafeNet Luna appliances, a key management server that supports the SafeNet Luna platform should be

used with AWS CloudHSM.

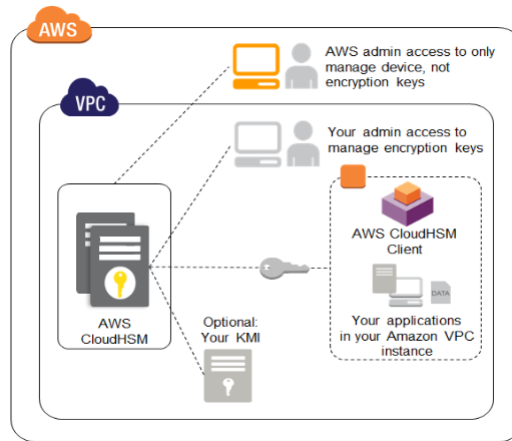


Figure 1.22: AWS CloudHSM deployed in Amazon VPC [12]

model C : AWS managed

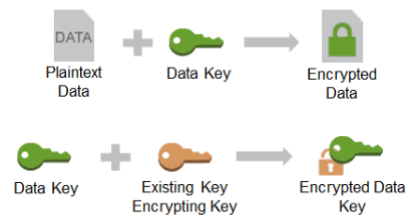
In this model the encryption process and the keys are managed by AWS. For encryption of data stored in S3 we can identify three levels of granularity offered when it comes to the level of control that the user can have over server-side encryption.

- **no control** : You simply ask Amazon to encrypt your data before it is stored (by setting a flag from the command line instructions or by checking a box in the AWS management console). In this case, each object stored will be encrypted with a unique key which will itself be encrypted by a master key, managed by AWS which is regularly rotated. The encryption process uses 256-bit Advanced Encryption Standard (AES) keys for both object and master keys.
- **providing the keys** : When uploading data to S3 you can provide your own encryption key that will be used to encrypt the data before it is stored. The key is not kept by Amazon and needs to be provided again when the data is retrieved.
- **using KMS** : This is similar to the previous one the difference being that this solution is more integrated given that the keys used for encryption are managed by the user through an Amazon web service known as KMS [22] (Key Management Service). Using this service gives the user control over the master key from which data keys are derived for encryption. KMS will interact seamlessly with S3 in order to manage the encryption and decryption process. KMS behaves in a way similar to HSM when it comes to managing the master keys because they are designed in such a way that the master keys never leave the service but data can be sent to it for encryption and decryption. AWS KMS uses an encryption procedure that follows this pattern :

1. A data key is generated by the AWS service at the time you request your data to be encrypted.



2. The data key is used to encrypt your data
3. The data key is then encrypted with a key encrypting key unique to the service storing your data.



4. The encrypted data key and the encrypted data are then stored by the AWS storage service on your behalf.



Chapter 2

Implementation

2.1 Solution 1

In this section we will present what was implemented in order to satisfy the requirements of finding a trade-off between local processing and remote processing for storing large images in the cloud while also concentrating on image compression and security aspects. We will first describe the global architecture of the solution with its different components and then go into more detail of each of those components in order to see what functionalities they offer and how they satisfy our requirements.

2.1.1 Architecture overview

Figure 2.1 shows that there are two main components in the solution. The client-side is the software that runs locally on the user's computer and contains the functionalities the user has access to. The server-side is the whole Cloud implementation and its role is to serve the client's requests. This implementation follows a 3-tier architecture where the presentation tier is the client app and the logic and data tier are implemented in the Cloud. The servers are located in an isolated portion of the Cloud called a VPC. We will first detail the main components of the Cloud architecture and then go into greater detail of some aspects of the implementation. The main components of this architecture are the following :

- **Route 53** : This is a DNS service offered by AWS. It also offers Domain Name Registration services. Using this service, the domain name *cloud-storage-ucl.com* was registered and mapped to the ELB so that all traffic directed to *cloud-storage-ucl.com* will be sent to the ELB;
- **Elastic Load Balancer** : This is the entry point to our VPC where our instances are running. The ELB plays multiple roles. Firstly it distributes the traffic to the instances it is connected to. Secondly, the ELB can detect unhealthy instances, for example if a server crashed, and redirects the traffic from that instance to another healthy instance. This is good for fault tolerance. The ELB can also be used as a SLL termination point as we will see in the security section;
- **EC2 instances** : This is where the logic of our app is implemented. These servers receive the request from the client through the ELB as illustrated in the diagram. Section 2.1.4 details the implementation that was done on the servers. For this implementation we used t2.micro instances;
- **auto-scaling** : In order to take full advantage of the cloud features, an auto-scaling group and a launch configuration were defined. An auto-scaling group defines how a group of instances will scale. The basic configurations are the minimum and maximum number of

instances that should be running at any time and the scaling policy which is the condition that needs to be met in order to either launch a new instance or stop an instance. For a start we chose that a new instance would be launched whenever the CPU load of an instance goes above 80% and an instance should be stopped whenever the CPU load of an instance goes below 30%. Finally, whenever a new instance is launched, we want it to be directly capable of serving requests from the client. This is the role of the launch-configuration which defines what softwares will be installed by default when a new instance is launched.

- **S3 storage** : Because we are interested in storing images, the S3 service is the right solution for this. It follows a bucket/object structure where the bucket is a container for file objects. Furthermore a bucket can be organized in folders using prefixes for the objects (for instance object "myImage.png" can be stored in folder "folder1" using the storage key "folder1/myImage.png") . It is highly scalable and reliable. Amazon S3 uses a combination of Content-MD5 checksums and cyclic redundancy checks (CRC) to detect data corruption. Amazon S3 performs these checksums on data at rest and repairs any corruption using redundant data. In addition, the service calculates checksums on all network traffic to detect corruption of data packets when storing or retrieving data. Amazon S3 Standard redundantly stores objects on multiple devices across multiple facilities in an Amazon S3 Region. The service is designed to sustain concurrent device failures by quickly detecting and repairing any lost redundancy.
- **MySQL server** : The EC2 instances communicate with a MySQL server that is launched through the Amazon RDS service (Relational Database Service). This is useful in a 3-tier architecture where users need to be authenticated in order to access functionalities.

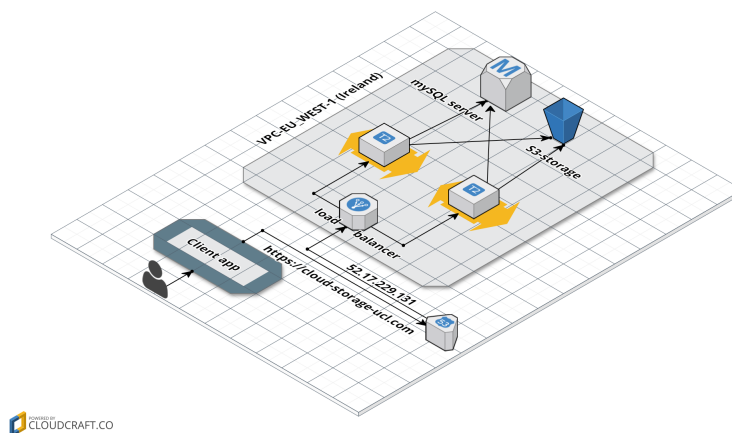


Figure 2.1: Components architecture

2.1.2 subnets and routing

The figure 2.2 shows another view of the cloud architecture focusing on the subnets, routing and addressing. First of all the implementation was done in eu-west-1 region which is located in Ireland. As we can see in the diagram, this region is divided into three availability zones. The VPC we have created spans the three availability zones and is assigned a set of IP addresses for private communication between the instances launched in the VPC. The VPC can be further broken down into subnets. A subnet can be used to add additional security as well as group together instances that play a similar role. In our case, subnet 1 and 2 are called public subnets because they are connected to the internet through the Internet Gateway. These subnets will hold our EC2 instances which

need to be publicly available. Subnet 3 is a private subnet which means it is not connected to the internet. Its purpose is to hold the database server which only needs to be accessed by EC2 instances located in either the subnet 1 or the subnet 2.

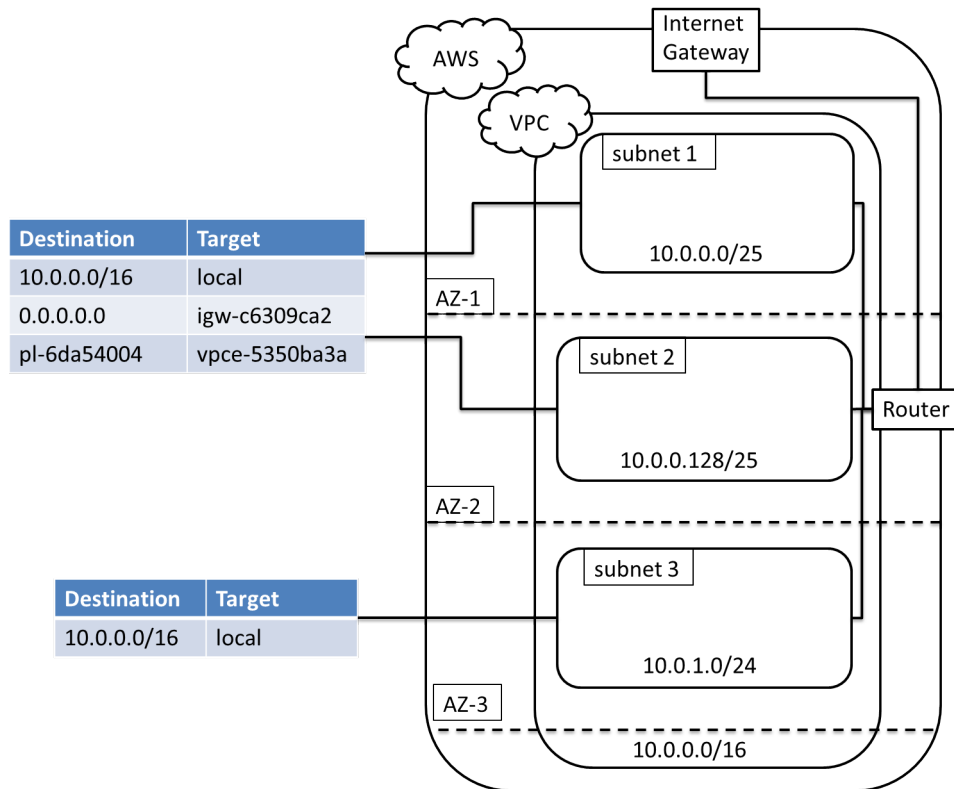


Figure 2.2: Vpc architecture

2.1.3 Client side

Before starting to develop the client application many decisions needed to be taken. Firstly I had to decide whether to develop a web app or a desktop app. The main advantage of a web app is flexibility because it can be accessed from any computer with internet access. On the other hand desktop apps can better leverage the computer's processing capabilities. Given that our solution is only aimed at a limited amount of users, the flexibility advantage isn't very relevant. In addition we need to be able to access the hardware and external libraries in order to apply some compression to the images so the desktop app seems to be the better choice.

Next I had to decide which framework to use to develop the app. My main concerns in order to make this choice were : How quickly can I get started with development ? Do I know the programming language that it uses ? Is it lightweight ? How mature is the framework ? Is it cross-platform ? My research led me to two frameworks : The Qt framework [23] and the Kivy framework [24]. On the one hand, Qt is a very popular C++ cross-platform framework for software development. It is also very mature given that the initial release was in 1995. Kivy on the other hand is a more recent Python framework for cross-platform applications. I chose to develop the client software using the Kivy framework for 2 reasons : Kivy is much more lightweight and easier to get started with. Also, Kivy is free to use which makes things easier.

The main features for the client app are the following :

- **User authentication** : The user authentication is done by means of a user name and

password as shown in the following picture. We will see how this is handled by the server in the next section;

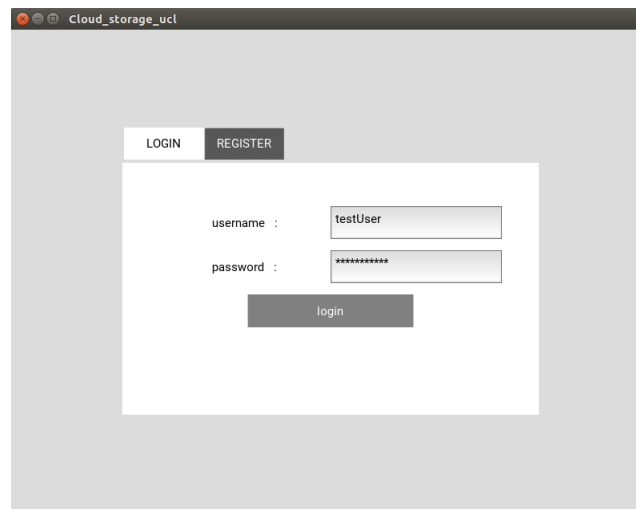


Figure 2.3: Log-in screen

- **Listing of all the images a user has access to :** A user can also see the filenames of all the pictures he has access to by clicking on the refresh button. These pictures are all located in folders. A user always has access to his own private folder but can also have access to shared folders;

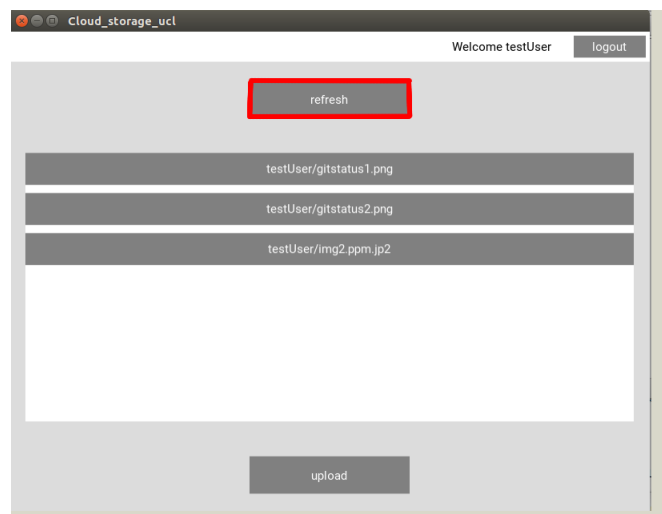


Figure 2.4: View all filenames

- **Downloading an image :** A user can download any image by clicking on the filename of the image he wishes to download and choosing a local filename and location for the image;
- **uploading an image :** Finally, a user can upload an image by clicking on the upload button, choosing an image to upload from the ORIGIN column and choosing a destination folder from the DESTINATION column.

2.1.4 Server side

As with the client, I had to decide what tools to use to develop the server that would communicate with the client. I decided to go with Apache [26] for the base layer that would handle

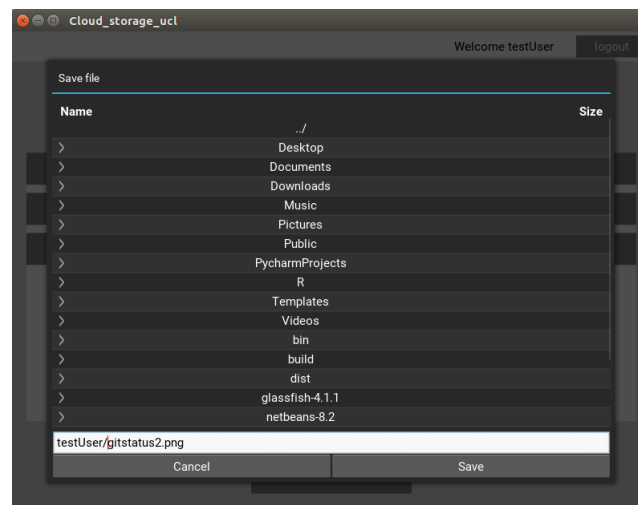


Figure 2.5: Download an image

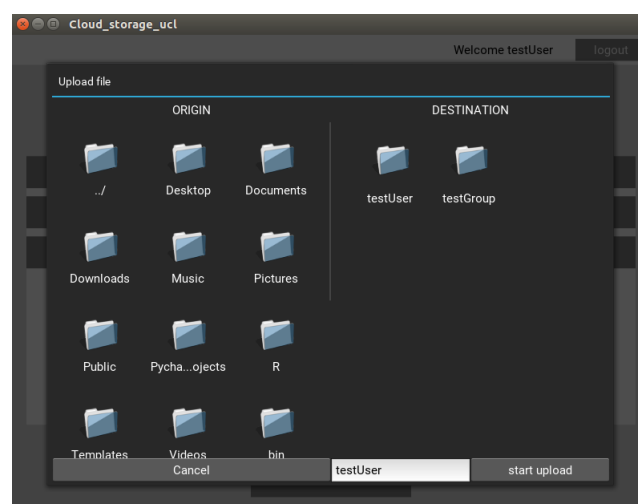


Figure 2.6: Upload an image

the connections with the users and a Django web app [25] for handling all the logic of storing images or serving them to the clients. Apache is the most commonly used HTTP server and is very reliable so it makes sense to use it. Django is a modern python web framework that is easy to use and allows for rapid development. Communication between the web server and the web app is made possible thanks to a module called WSGI (Web Server Gateway Interface).

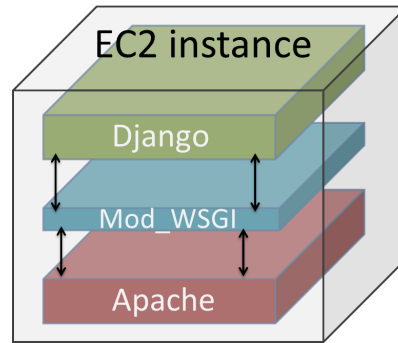


Figure 2.7: Apache, mod_wsgi and django interaction

Different paths with different behaviour have been implemented in order to serve the client's requests. These paths correspond to the different functionalities implemented on the client and are accessed either using HTTP GET requests or HTTP POST requests.

- **/login** : This path is accessed using a HTTP POST request from the client when he submits his username and password to the server. The authentication process is done by the built-in Django authentication system. By default Django generates .sqlite files which hold information corresponding to the users. Figure 2.8 shows the structure of some of those tables. Given our architecture which can contain multiple django servers with each of them their own sqlite file containing the tables, maintaining consistency between them can be difficult. For that reason we chose to work with AWS Relational Database service which stores the tables on a mySQL server. Using this database server there is no need to worry about consistency. The username and password are stored in the **Auth_user** table as shown in figure 2.8. The built-in Django authentication system handles password's hashing and salting. If the username and password are validated by the server, it will return a session token that the user application can include in further communications as mean of identification. All the following paths will verify the session token before serving the requests.
- **/list** : This GET request allows the user to view all the filenames of the images he has access to. A user has access to his own folder and group folders. This organisation of users and groups is presented in figure 2.8. The server can query this database in order to retrieve the groups a user is part of. Figure 2.9 shows the bucket structure in S3 for storing images. Each user or group corresponds to a folder in S3. This means that once the server has retrieved the groups a user is part of, it can query S3 to retrieve the content of the folders that have the same name. For instance if User1 is part of Group1, the list function will retrieve the filenames "Image1", "Image2", "Image4", "Image5", that is, his own images and those of Group1.

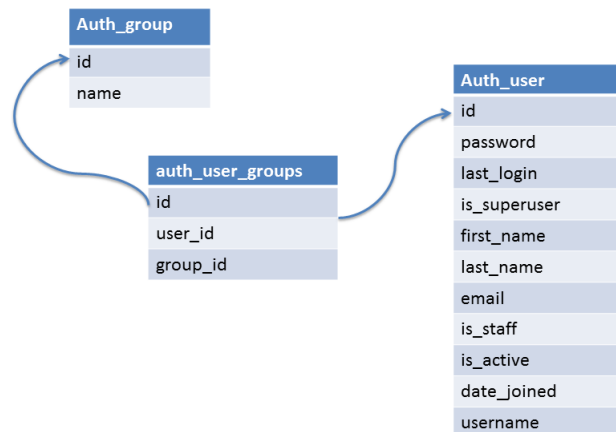


Figure 2.8: Django database

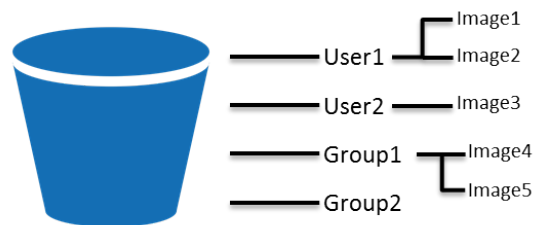


Figure 2.9: Image storage structure

- `/upload/{image}` : This is a POST request for uploading a local image to the remote storage location. Figure 2.10

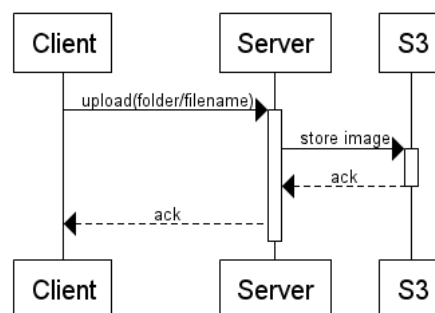


Figure 2.10: Upload sequence diagram

- `/download/{image}` : This is a GET request for downloading an image.

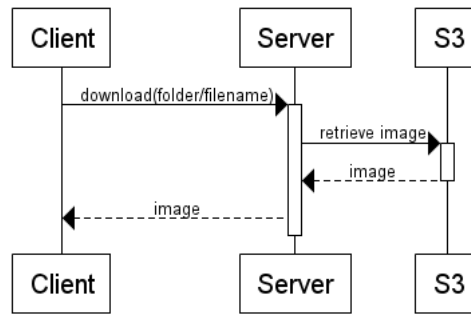
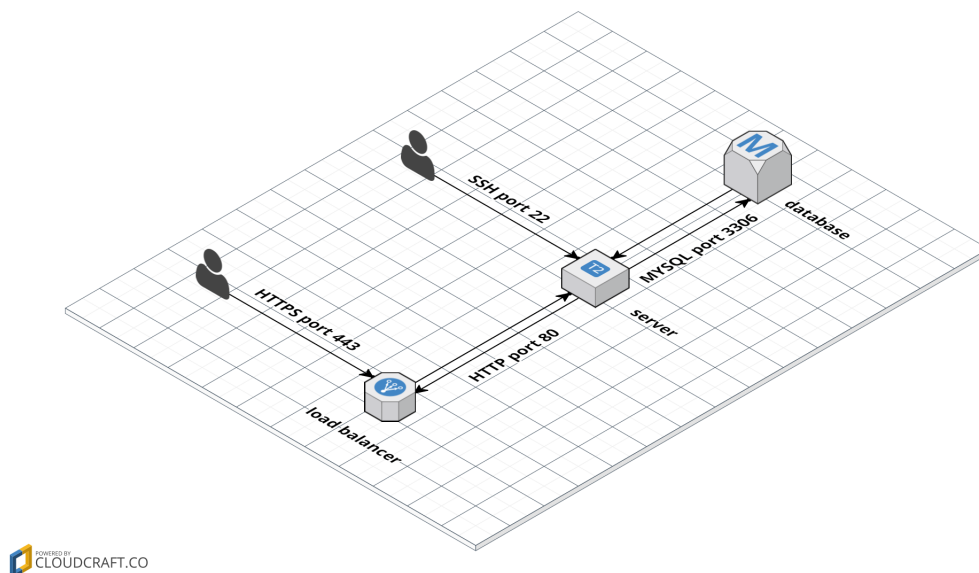


Figure 2.11: Download sequence diagram

2.1.5 Security

In this section we will review the different security implementations that were done in order to guarantee the confidentiality of the images that are stored in the cloud. Security takes place at different levels as we have seen in the theoretical foundations. We need to protect data in transit as well as control access to the data.

Access is regulated by the various security groups. The following figure shows the ports that are open. We can see that the server is only accessible by SSH from any source or from the load balancer through HTTP. The load balancer, in addition to distributing the traffic to the different servers, plays the role of SSL termination point. This means that the load balancer decrypts the data coming from the client before forwarding it to the instance. The main advantage of using the load balancer as an SSL termination point is that it offloads the computational intensive task of decryption from the servers to allow them to focus on the business logic.



POWERED BY
CLOUDCRAFT.CO

Figure 2.12: Overview of security groups

In order to establish a secure connexion between the client and the load balancer 2 certificates were generated using the OpenSSL library. One is called the root certificate. It contains a

public key that is self-signed which means it has been encrypted using the corresponding private key. This root certificate plays the same role as certificates that are installed by default on the browsers which is to validate the server certificate. The difference is that our certificate will not be installed on a browser but will be contained within the client software. The second certificate is the one installed on the load balancer. It contains a public key signed by the private key of the certification authority (ourselves) that signed the root certificate. Using this, a secure communication channel can be established between the client app and the Virtual Private Cloud.

Access is also controlled by the user signing in on the client-side to make sure only registered users can access the functionalities and on the server side by verifying that users are signed in before serving appropriate content to them.

2.2 Solution 2

This second solution is an attempt to solve the initial problem using an approach that takes more into account the fact that the images can be very large. The goal here is to implement a different architecture that makes use of new tools and techniques and then compare it with our first solution. The main technique used here will be to tile the image. That is to break it down into smaller images. Each sub-image will then be treated individually both for the upload and for the download.

2.2.1 Backend Architecture

In order to implement this different approach of tiling the initial image and then sending each sub-image individually, a different design for the backend was conceived which better fits this new mechanism. This new design makes use of two new AWS services known as API Gateway and Lambda Functions.

- **API Gateway** : This service allows developers to expose functionalities and services through HTTP requests. The API consists of a set of resources and methods. A resource is a logical entity that can be accessed through a resource path using the API. A resource can have one or more operations that are defined by appropriate HTTP verbs such as GET, POST, and DELETE. A combination of a resource path and an operation identify a method in the API. This API acts as a “front door” for applications to access data, business logic, or functionality from back-end services, such as workloads running on Amazon Elastic Compute Cloud (Amazon EC2), code running on AWS Lambda, or any Web application.
- **Lambda** : AWS Lambda is a compute service that allows the user to run code without provisioning or managing servers. The code is executed only when necessary based on a triggering mechanism. This type of service is known as "Function as a Service" (FaaS).

Figure 2.13 depicts this combination of API Gateway and Lambda. This architecture fits a more general design pattern known as **serverless microservices**. The serverless aspect comes from the fact that there isn't a server that is constantly running and waiting for requests as was the case with the first solution. There still is some amount of server logic that is written but it is stateless and runs in ephemeral containers (i.e., servers that are provisioned for the duration of the code execution). The microservices aspect comes from the fact that the functionalities offered by a given service are divided into independent components. This design style is the opposite of the monolithic application like the one developed in the first solution. With a monolithic application, all the functionalities are bundled together and delivered through a single server. As shown in the following picture, 3 services are exposed through 3 API methods, each of them integrated with it's own lambda function :

- **GET /list** : This method allows a user to retrieve all the file names of the images he has uploaded.
- **GET /download** : This method can be called in order to download a specific image. Additional information necessary for this request such as the file name is past in the header of the query.
- **POST /upload** : The method is called to upload an image to the Cloud. As with the download method, additional information such as the file name or the name of the user uploading the image is passed in the header of the query.

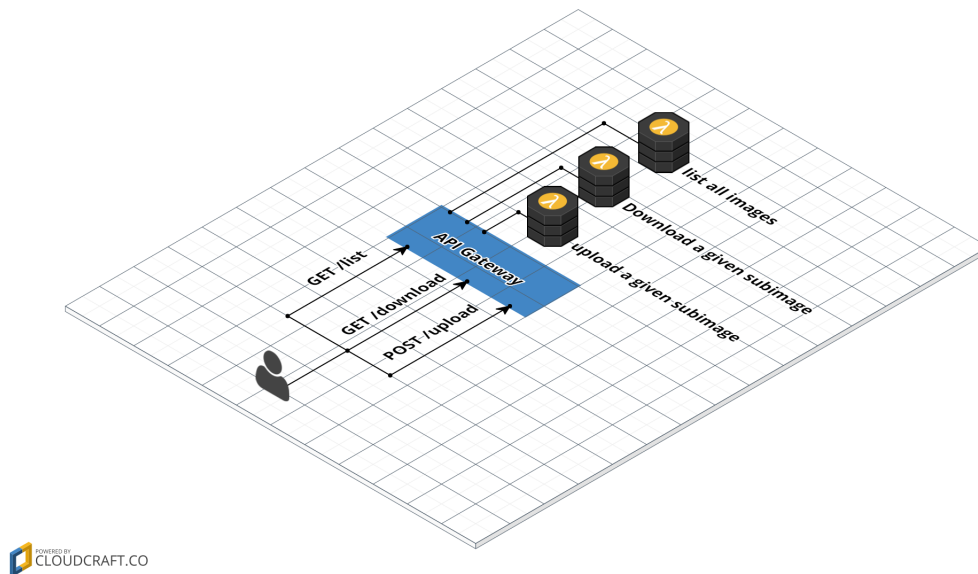


Figure 2.13: API architecture

2.2.2 Authentication system

As with our first solution, we want our backend solution to be accessible only to authenticated users. In our first solution, the client sends the user's credentials to the server which would either establish a session with the client if the credentials are validated or deny the connection. In a microservice architecture, such a pattern is inefficient and cumbersome because it would require the client to authenticate with each service that is accessed. Furthermore because the backend is stateless, once a request has been served the client has to authenticate once again for the same service. Many approaches can solve this issue. The solution that was chosen here is the use of token-based authentication.

The initial concept with token based authentication is the distinction between the authentication server and the resource server. In a monolithic application, the server plays both of those roles but in a microservice architecture these role are managed by distinct entities. Figure 2.14 shows the interaction between the client app, the authentication server and the resource server.

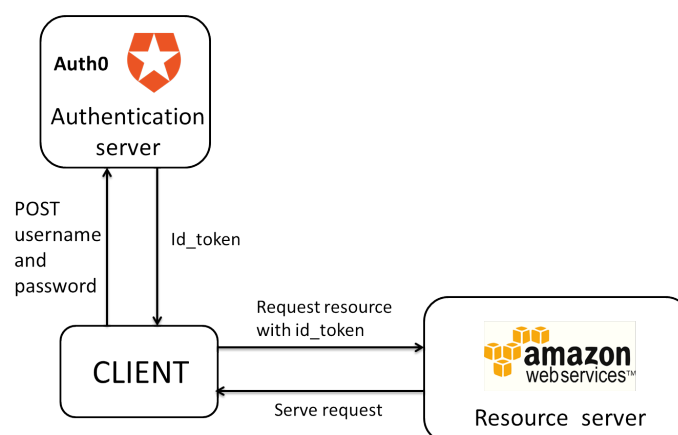


Figure 2.14: Authentication flow

The authentication server role is played by Auth0. Auth0 is a platform for managing identity.

It provides a wide range of tools for managing user authentication and authorization through various flows. In order to use this service, we need to register an application within Auth0 which identifies our client app by creating a unique client id and client secret. We also create a user profile that we will use for authentication. The resource server is managed by AWS and accessed through the API as discussed previously.

When the user launches the desktop app, he is shown an authentication widget as illustrated in figure 2.15. The user must fill in his email address and password. These two items are sent to Auth0 for authentication. If they are valid, Auth0 will return an id token and the user will be able to access the client functionalities.

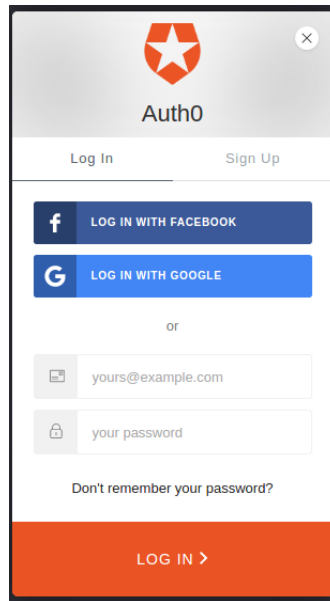


Figure 2.15: Authentication widget

The id token is known as a JSON Web Token. JWT is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the HMAC algorithm) or a public/private key pair using RSA. The JWT is composed of a header a payload and a signature. The header has two parts : the type of the token and the algorithm being used for signing. The payload is composed of claims which are statements about the user that has been authenticated and additional metadata. The signature is the result of base64 encoding the header and the payload and inputting them with a secret to a hashing algorithm. The secret that Auth0 uses in this process is the client secret of our application.

Once the client is in control of the JWT he can embed it in the header of any API request in order to access protected resources.

Figure 2.16 illustrates the authorization process that takes place in the backend in order to decide if the user that is making the request for a specific resource is allowed access to the resource or not.

1. The client makes an API call for a specific resource (for instance GET /list) and puts the JWT in the header of the request.
2. The API Gateway service receives this request and forwards it to a lambda authorization function.

3. The lambda function validates the JWT signature by creating its own signature in the same way Auth0 created the signatures contained in the JWT and compares the result with the signature contained in the JWT. If both signatures are identical, then we can be sure the token represents a user that has been correctly authenticated and we can authorize him to access the resource. This authorization is done by means of a policy which is a JSON document which contains statements about what AWS services can be accessed and at what level.
4. The policy returned by the authorization function is cached which allows subsequent request using the same token to not have to be validated once again.
5. the initial call is forwarded to the lambda function responsible for the given API resource.

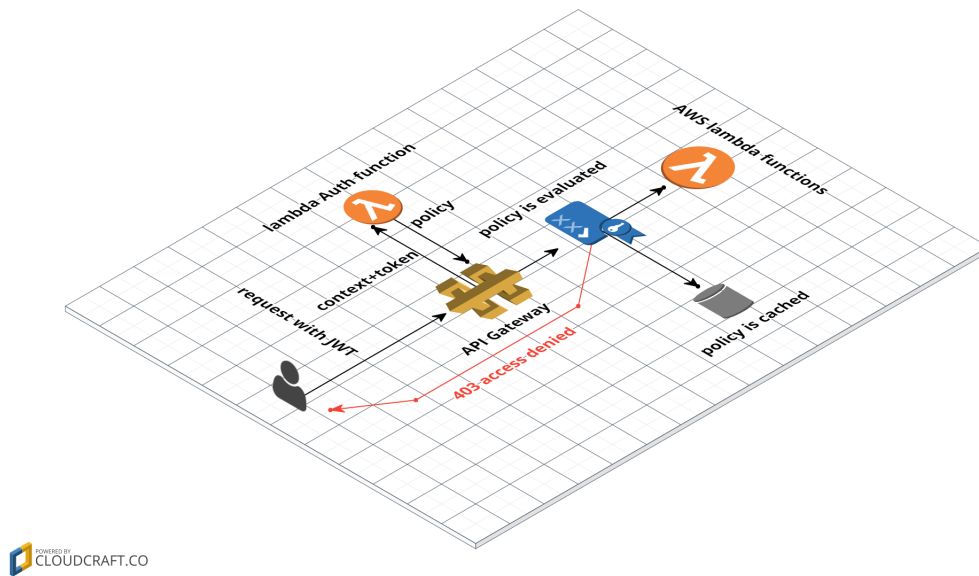


Figure 2.16: API authorization system

2.2.3 frontend-backend interaction

Let us now take a look at how the client application interacts with the API in order to store or retrieve images. Let's detail the different functionalities the user has access to :

- **upload** : Figure 2.17 depicts how the frontend interacts with the backend for uploading images. The frontend end which is the desktop application receives an image that the user wishes to upload to the cloud. The frontend's role is to tile the image into a given number of sub-images. The number of sub-images depends on the size of the initial image. Once the image is tiled, each sub-image is treated separately and sent to the backend. Filenames of all sub-images are made distinct so as to be sure that once they are stored we can distinguish them from one another. Metadata that contains the position of the sub-image within the initial image is also passed to the backend. This tiling technique allows to highly parallelize the processing both on the client and on the server side. While the client parallel capabilities are still limited by the number of cores, the benefits of having the serverless architecture is becoming clear here because any number of lambda functions can be executed simultaneously. This means that if the initial image is divided into 100 sub-images, then we can have 100 lambda functions that handles its own sub-image and stores it in S3.

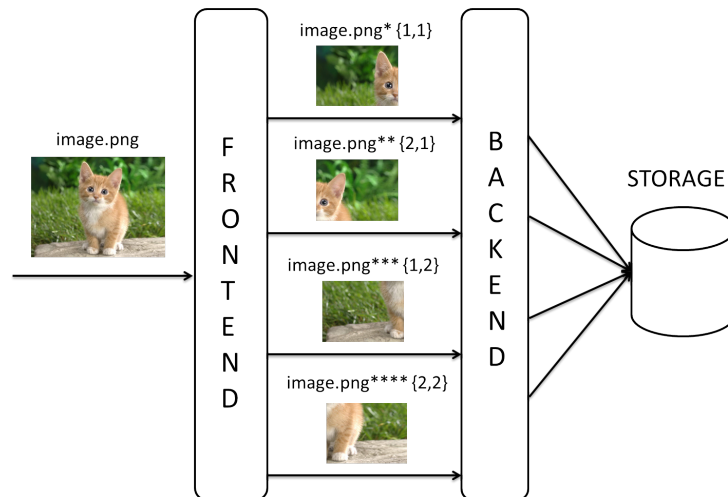


Figure 2.17: Client-side tiling and sending to backend

- **Download** : The Download function is the reversed operation of the upload function. Given a filename for an image and a number of sub-images, the client launches as many API requests as there are sub-images. Each of these request will fetch a specific image with its metadata. Once the client has received all of the sub-images, it can stitch back the sub-images together using the positions contained in the metadata of each image and store the final image on the user's computer.
- **Listing** : The list function retrieves all the filenames of the images a user has previously uploaded. More precisely it will retrieve the filenames of the sub-images of all images and then aggregate the result per image in order to return to the client a pair (image name, number of sub-images).

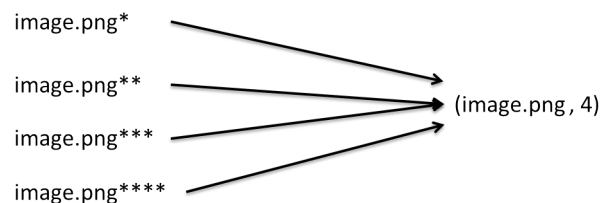


Figure 2.18: Listing done by aggregating sub-images

Chapter 3

Benchmark

3.1 Compression algorithms

In order to reduce the bandwidth cost of uploading large images, as well as reducing the cost of storing large images in the Cloud, we will study in this section and the following the usefulness of compressing the images. A few questions need to be answered :

1. Where should we apply the compression? On the Client-side? On the server-side? On both?
2. Should we choose lossy or lossless compression?
3. How well does lossless compression perform? What algorithm should we use ?

Given the nature of the images that are used for medical diagnostic it is necessary to use lossless compression algorithms. The user should be able to zoom into the image without having any artifacts appearing if he wants to perform an accurate diagnosis.

Question 1 will be answered in the following section by doing a performance benchmark on the different cases (client-side or server-side).

In order to answer the third question we will benchmark different compression algorithms that have a lossless mode. We will report both the compression time in seconds and the compression ratio computed as :

$$\frac{\text{Initial size}}{\text{Final size}}$$

The following compression algorithms will be tested :

1. **webP** : WebP is an image format employing both lossy and lossless compression. It is currently developed by Google, based on technology acquired with the purchase of On2 Technologies.
2. **Bzip2** : bzip2 is a free and open-source file compression program that uses the Burrows–Wheeler algorithm. It only compresses single files and is not a file archiver.
3. **JPEG XR** : The JPEG XR standard provides a practical coding technology for a broad range of applications with excellent compression capability and important additional functionalities. JPEG XR can be run in lossless mode.
4. **JPEG LS** : JPEG LS was defined to address the need for effective lossless and near-lossless compression of continuous-tone still images. JPEG LS is especially suited for low-complexity

hardware implementations of very moderate complexity, while at the same time providing state-of-the-art lossless compression performance.

5. **JPEG 2000** : JPEG 2000 is an image coding system that uses state-of-the-art compression techniques based on wavelet technology. Its architecture lends itself to a wide range of uses from portable digital cameras through to advanced pre-press, medical imaging and other key sectors.

These algorithms were tested on 9 images of increasing size. It is important to mention here that some of the libraries we used did not accept the initial file format. For the algorithms JPEG XR and JPEG LS we had to convert the initial images to the `.ppm`. This conversion is not taken into account for the runtimes and the compression ratio is computed as

$$\frac{\text{Initial size before conversion}}{\text{Final size after compression}}$$

file name	weight	dimensions (pixels)
img1.png	6.1 MB	6300x3000
img2.tif	14.5 MB	4000x2461
img3.tif	16.4 MB	4000x2380
img4.png	25.1 MB	6300x3000
img5.tif	30.2 MB	6127x3979
img6.png	41.6 MB	9725x4862
img7.tif	59.8 MB	3557x5606
img8.tif	99.5 MB	7857x7462
imgs9.tif	138.7 MB	6800x6800

file name	weight	dimensions (pixels)
img1.ppm	54 MB	6300x3000
img2.ppm	28 MB	4000x2461
img3.ppm	27 MB	4000x2380
img4.ppm	54 MB	6300x3000
img5.ppm	70 MB	6127x3979
img6.ppm	135 MB	9725x4862
img7.ppm	57 MB	3557x5606
img8.ppm	168 MB	7857x7462
imgs9.ppm	132 MB	6800x6800

Figure 3.1 and 3.2 report the execution time as well as the compression ratio. The computer used for these benchmarks has the following characteristics : Intel Core i3 at 2.3 GHz, 4 GB memory.

From the first graph, we can see that JPEG LS is by far the quickest algorithm to compress the images.

From the second graph we can see that three algorithms have very similar compression ratios of around 20-25%. These algorithms are WebP, JPEG 2000 and JPEG LS. Unfortunately there

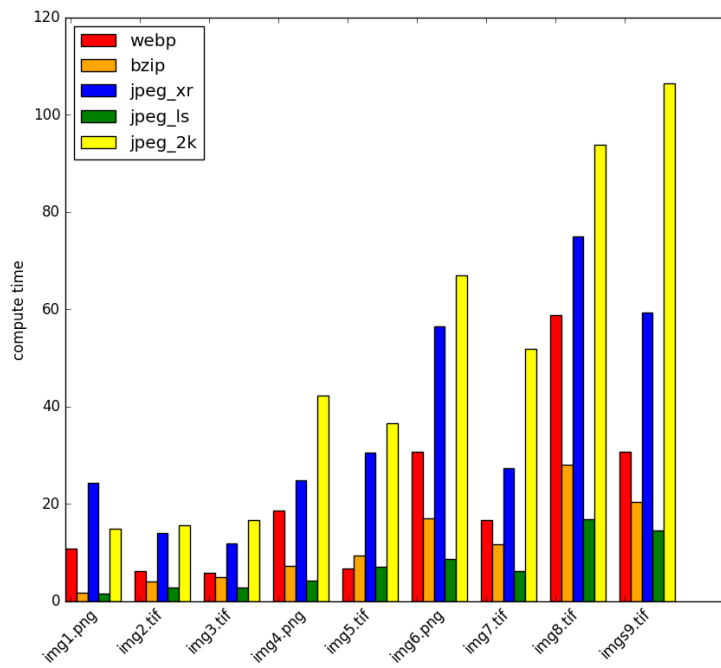


Figure 3.1: Execution time per image

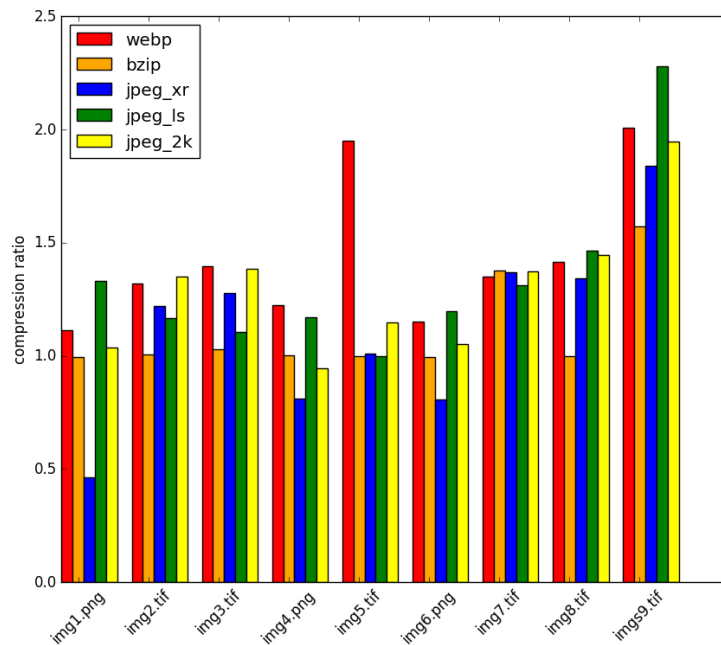


Figure 3.2: Compression ratios per image

is no one algorithm that clearly surpasses the other ones. Given that JPEG LS is by far the quickest one, we will use this algorithm for our following performance benchmark. We will add a call to the JPEG-LS library when an image is uploaded or downloaded to respectively compress or decompress the image. This call will be done on the server side to avoid utilizing too much of the client's CPU.

3.2 Performance benchmark

3.2.1 Solution 1

We will now assess the performance of our solution by submitting it to various workloads. First of all, we need to make assumptions about how our solution will be used in practice and what type of workload we can expect it to have. Given the solution will not be widely distributed we don't expect to have too many concurrent users. Furthermore we expect the users to upload or download some images on an occasional basis. This means that the server will be mostly idle while occasionally having to bear a medium to large workload. We will start by a basic test case :

1. **test case 1 (sequential)** : For this first test we will upload all images in a sequential manner, i.e., the server will never have to process more than one image at the time. The following graph shows the round trip time (RTT) of our different test images. The RTT is defined as the time needed for the client to read an image to memory, send it to the server and receive an acknowledgement that the image was correctly compressed and stored in the Cloud.

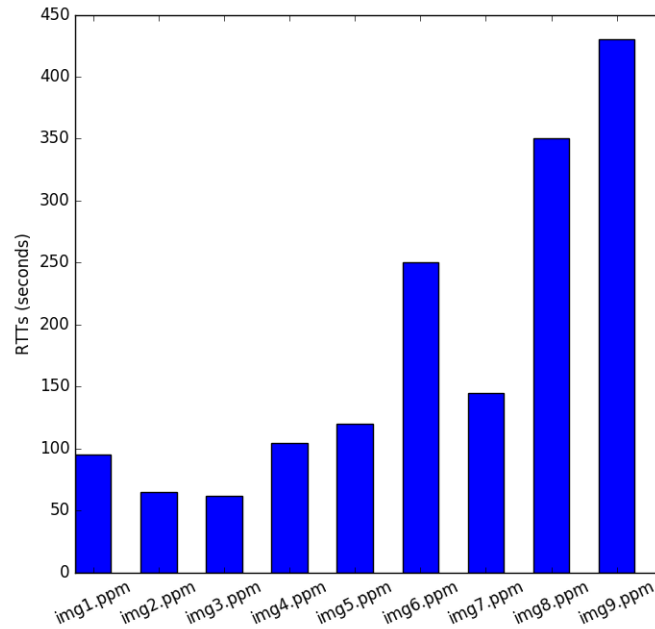


Figure 3.3: RTTs for the sequential test case

While looking at figure 3.3 we can see that the RTT are very long. Overall sending all 9 images took a little over 20 minutes. Figure 3.4 shows the CPU usage of the server during the time the images were being uploaded. As we can see, the CPU utilization never reaches 40 %. From a user experience point of view, these RTTs are too long. If the user decides to quit the application before all images are sent, which means during the time lapse of 20 minutes, all images not yet processed by the client will not be sent and the user will have to try to upload them once again when he restarts the application.

2. **test case 2 (all at once)** : We will now try to put the server under more pressure by sending all of test images simultaneously. Figure 3.5 reports the RTT needed for each image.

This second benchmark gives us some interesting results. Firstly by looking at the RTT from figure 3.5 we can see that the RTT of each image is approximately multiplied by a factor 4.

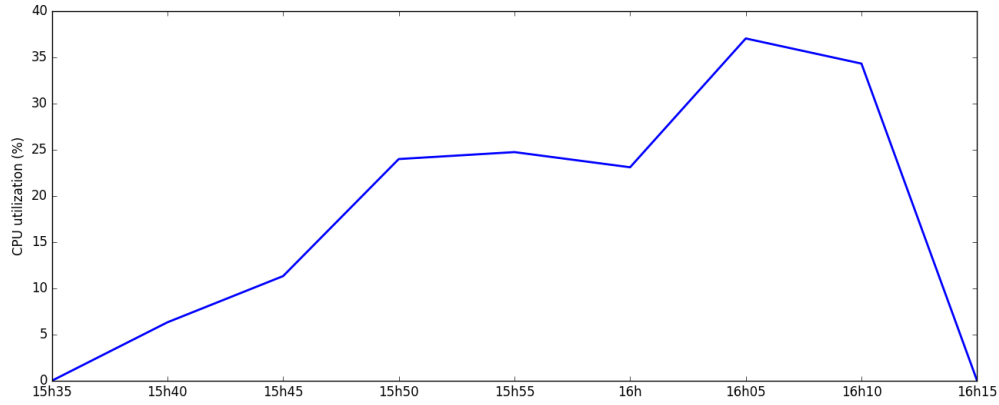


Figure 3.4: CPU usage of the server (%)

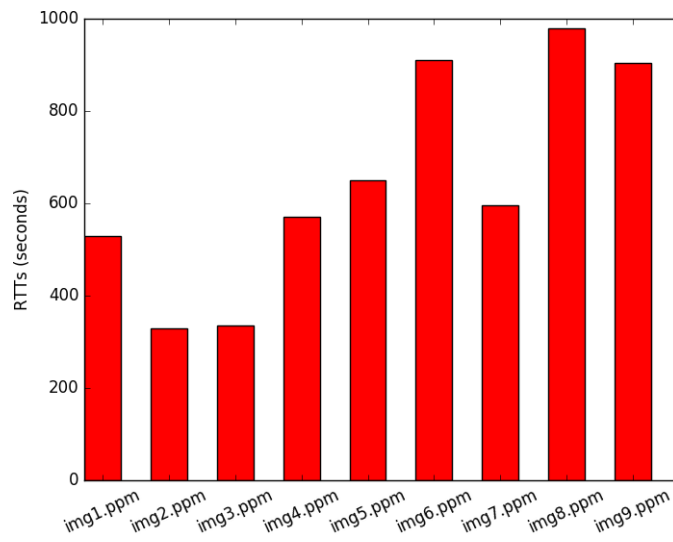


Figure 3.5: RTTs when uploading images all at once

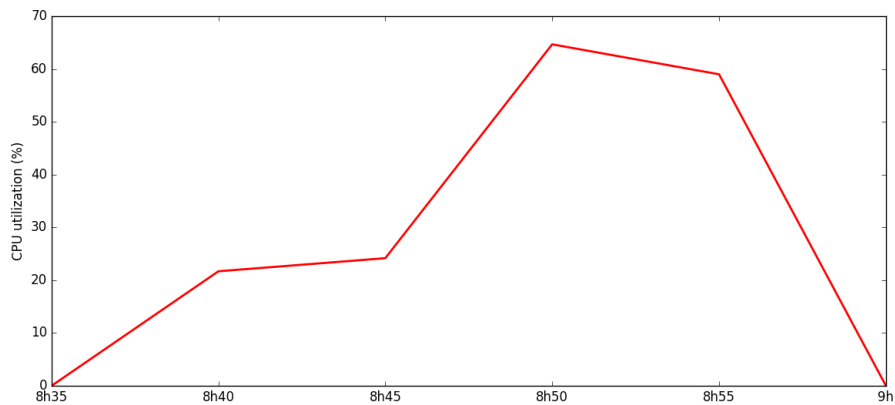


Figure 3.6: CPU usage of the server (%)

Nonetheless, the overall time needed to send all the images is approximately 15 minutes which is a

slight improvement over the sequential test case. Figure 3.6 shows that there is a higher utilization of the server with a peak at 65 %. From a user experience point of view this seems to be a better way of uploading images. All images are sent concurrently to the server within a short time interval which would allow the user to terminate the application without risking of stopping the sending of images as in the first test case. Let us now break down the RTT to see where the bottleneck lies.

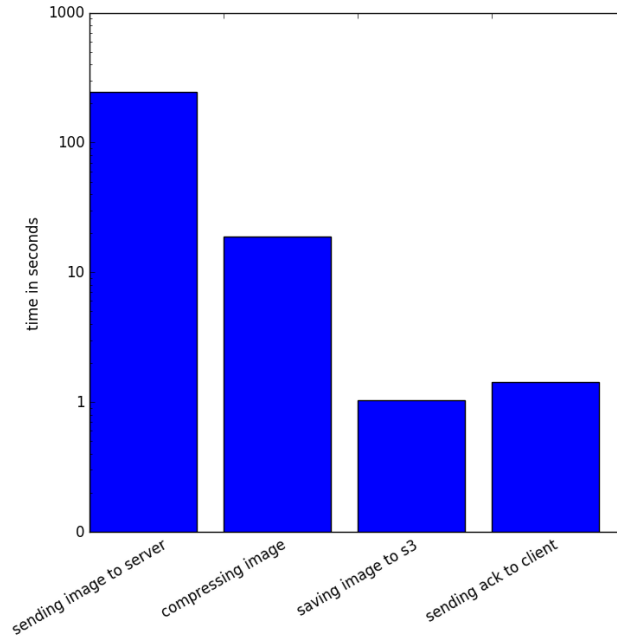


Figure 3.7: Breakdown of the RTT for sending img6.ppm

From figure 3.7 we can see that the most time consuming task is sending the images to the server. From the time when the client sends its POST request containing the image file it takes 246 seconds before the server application receives it and can start manipulating it. Let's move the compression algorithm to the client to see if it results in an improvement of the RTT.

With client-side compression, we can see a clear improvement of the RRT which drops to 71 seconds. With client-side compression the sending of the image takes only 60 seconds (figure 3.8) as compared to 246 seconds with server-side compression. This represents a trade-off given that some of the workload is now passed to the client. Sending too many images at once could significantly slow down the user's computer.

Figure 3.9 compares the RTTs of sending all images at once with client-side compression and server-side compression.

As we can see in figure 3.10 listing all the images a user has access to takes place quickly.

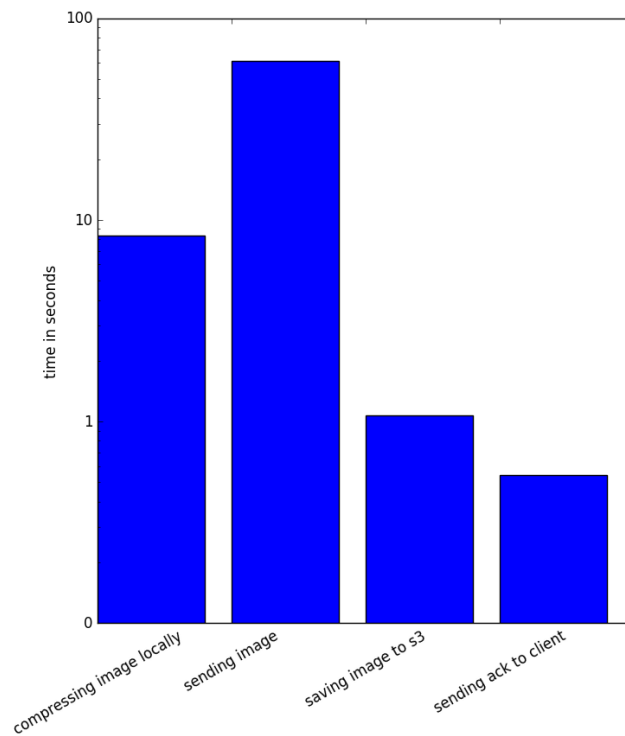


Figure 3.8: RTT breakdown with client-side compression

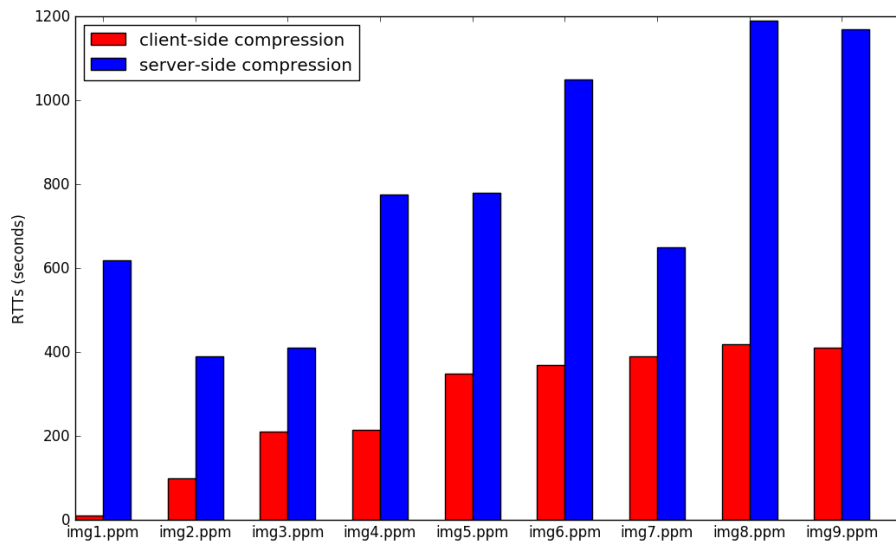


Figure 3.9: RTTs for all-at-once image sending

3.2.2 Solution 2

We will now study the performance of the second solution and compare our results with our first solution. The first step is to decide on the tiling factor to apply, i.e. how much we want to break down our initial image before sending it. Having a low tiling factor means less local processing given that we don't need to split the image into too many pieces but also means that the HTTP bodies will be larger and thus slower to send. A high tiling factor means more sub-images to

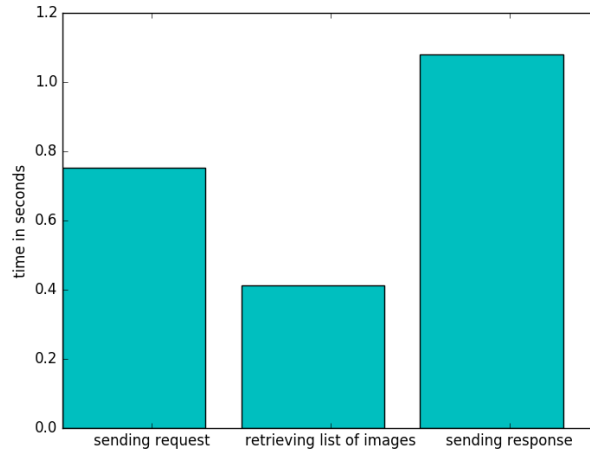


Figure 3.10: RTT for the list function

send but each of them can be sent quickly as they have a smaller size. Figure 3.11 shows some results. The x-axis is the number of sub-images and the y-axis is the RTT for the given number of sub-images specified on the x-axis. The RTT is computed as the time spent to load the image into memory, tile the image, send each sub-image to the backend, store each image in S3 and send an acknowledgement to the client. We can see that the time to send all images tends to decrease when the tiling factor increases. However too much tiling is not useful as it results in too much client-side processing and not enough gain from having smaller HTTP requests. We can see from figure 3.11 that a tiling factor of 24 seems to be optimal for img3.tif. Given that img3.tif's size is 4000x2380. A tiling factor of 24 results in individual tiles of size 630x630 approximately. We will use this tile size for further benchmarks in order to have a dynamic tiling factor.

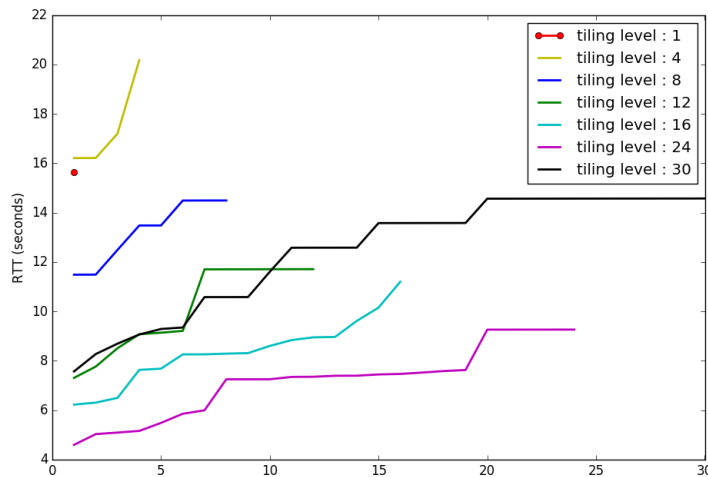


Figure 3.11: Runtimes for different tiling factors on img3.tif

Figure 3.15 shows the RTT's for img8.png which is of size 7857x7462. We see here that the optimal tiling factor lies somewhere between 74 and 84. For a tiling factor higher than this there is no significant improvement.

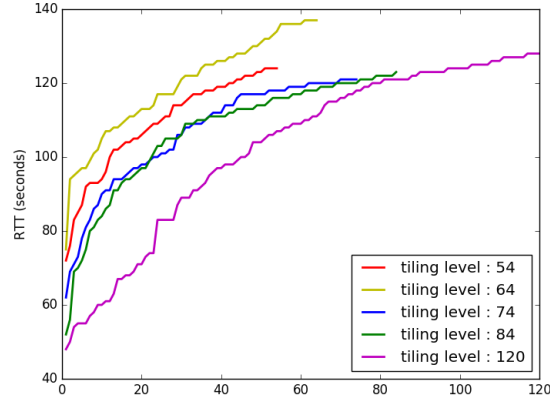


Figure 3.12: Runtimes for different tiling factors on img8.tif

This last figure shows the parallel capabilities of our second solution. The different colours represent the different stages of sending an image and the y-axis represents the amount of time that has elapsed since the first sub-image was sent. As we can see, all sub-images are sent with a slight interval due to the limited parallel capabilities of our computer. Furthermore we can see on the backend that many lambda functions can be executed at the same time. As with our first solution, the bottleneck lies essentially in sending the images which is the most time consuming task. Nonetheless it is significantly reduced thanks to the tiling mechanism.

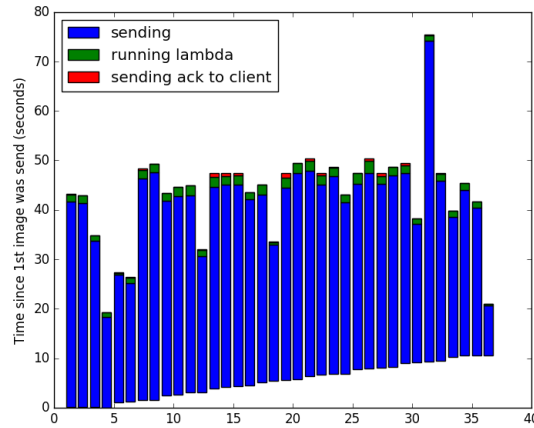


Figure 3.13: Sending sub-images for img4.png

3.3 Cost estimation

The goal here will be to estimate and compare the monthly cost of our 2 Cloud solutions. The cost of Cloud services is highly dependent on the amount of usage that is made of those services. That is why we will define an expected usage of our Cloud solutions based on what we think would be a representative workload. We define the following workload for our 3 functionalities :

1. **Upload** : Given the fact that the images can be very large and that the main use of our tool is to store images we assume a daily upload of 1GB on average. We also assume that potential users will have a tendency to upload large quantities of images at once and then not upload any images during many days. We also assume that the 30 GB of new data uploaded each month represent 70 new images.
2. **Download** : The amount of data downloaded is expected to be lower than the amount of data uploaded. We will suppose an average download per day of 300MB. At the end of the month, the user will have downloaded 9GB of data which represent 20 images.
3. **List** : although this function does not imply downloading or uploading content to S3, it still requires accessing content in S3 which has some cost. If the user stores and average of 100 images using our app, each time he launches the app he will perform at least once a LIST call to retrieve the filenames of all his images.

3.3.1 Solution 1

We will now go over the different AWS services we utilize of and compute their cost for the first solution.

Simple Storage Service

Simple Storage Service [27]		
Service	Cost	Description
Storage	\$ 0.023/GB for the first 50 TB/month	Cost of storing data in S3
Requests	PUT and LIST : \$ 0.005/1000 requests	Cost of making PUT and LIST requests to S3
	GET : \$ 0.004/10000 requests	Cost of making GET requests to S3
Transfer in	Free	All data transfer into S3 is free
Transfer out	First 1GB/month is free	Cost of transferring data out of S3
	up to 10TB/month : \$ 0.09/GB	

For the workload we defined previously and the pricing defined in the table we now compute the cost of S3 for the first month (30 days):

- Storage : $0.023 * 30 = \$0.69$ /month for storing 30GB of data
- Requests : We expect 70 PUT, 20 GET and 30 LIST. So for 120 requests/month the price is \$0.00051/month.

- Transfer out : free because transfer only occurs between EC2 and S3 in the same region which bears no cost

Elastic Cloud Compute

For the EC2 service, we have a t2.micro instance running 24 hours a day during 30 days at a cost of \$0.013/hour. This give us a monthly fee of \$9.36.

The cost of transferring data out of EC2 is estimated to be $8 * 0.09 = \$0.81$ given that the first GB is free.

Elastic Load Balancer

For the ELB service the overall cost has two components :

1. The amount of time the ELB software is running : \$0.0252/ELB-hour. Given that our ELB is running non-stop that amounts to $0.0252 * 24 * 30 = \$18.14$
2. The amount of data it processes : \$0.008 /GB. Given that our ELB processes approximately 39GB per month we have a final cost of \$0.32/month.

Elastic Block Storage

Given that we have an 8 GB hard drive attached to our EC2 instance and that the monthly fee for our hard drive is \$0.11/GB-month, this means the total cost of our hard drive is \$0.88/month.

Furthermore we need to take into account the cost for EBS snapshots which are S3 backups of the data contained on our hard drive. This data is mainly comprised of the set of software we installed such as Apache or Django. This amounts to a total of 1.5 GB approximately. Given that the unit cost of a snapshot is \$0.05/GB-month, the total cost is \$0.075.

Route 53

Route 53 has a cost of \$0.50 per hosted zone. We have one hosted zone (cloud-storage-ucl.com) sowe must pay \$0.5/month.

There is also a cost of \$0.40 per 1 000 000 queries for the first 1 Billion queries. Given that the amount of requests we expect to have is less than 1000, we can safely discard this cost.

Relational Database Service

The final service that we use is called RDS. As a reminder, we use this service to store the users information such as username, password, address, etc. The price per hour for our db.t2.micro instance is \$0.018. Because it is running non-stop, we can expect our monthly expenses for the RDS service to reach \$12.96.

There is also an additional cost for the storage capacity associated with our database instance. The cost is \$0.127/GB-month. Because we have allocated 5 GB of storage to our instance (we can't go below this value), the monthly cost of the storage is \$0.636.

The following table summarizes all of our expected costs as computed previously. We think that this represents a safe lower bound on the cost of the first solution given the fact that we have

omitted some very low costs. In addition to these low costs, there are sometimes some hidden costs that are not always easy to estimate. By hidden cost we mean some implicit mechanism of AWS services which could increase the cost such as automatic logging for instance.

S3	\$0.69
EC2	\$10.17
ELB	\$18.46
EBS	\$0.995
Route 53	\$0.5
RDS	\$13.595
total	\$44.41

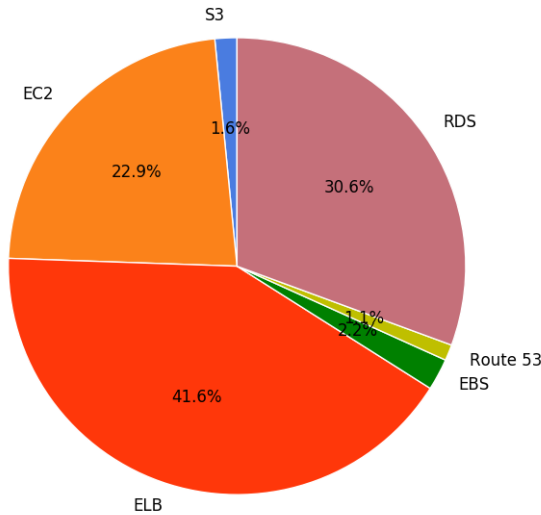


Figure 3.14: Repartition of the cost between the different services

As we can see from this table, the total cost of the first solution is \$44.41/month. This cost will only increase by a few dollars each month because of the additional data that is stored. The cost of the most expensive components will remain stable from one month to the next. The expensive components are the EC2 instances, the ELB and the RDS instance. It is worth noting here that these components will remain idle most of the time because we expect our solution to be used only occasionally. Furthermore, following the benchmark we have made, it seems rather unlikely that a second instance could be launched which implies that the load balancer is not very useful. Let us repeat here that the load balancer in addition to spreading the load to multiple instances, it also plays the role of SSL termination. This means that if we delete the ELB, we will need implement SSL decryption on our Apache server. This represents a decision that should to be carefully evaluated as it would increase the workload on our server. Doing so would decrease the cost to \$11.46/month

An additional benefit of deleting the ELB is that the database instance is no longer needed.

3.3.2 Solution 2

We will now evaluate the cost of the second solution and compare it to the first solution.

S3

The cost of this service is the same as our first solution given that we store the same amount of data.

Lambda

The cost of the Lambda service has two components :

1. The total number of requests made to all of our functions. The unit price is \$0.0000002 per request. To compute the total number of requests made, we should take into account the number of uploads, downloads, calls to the list function as well as the tiling factor. The number of uploads is 70 and the number of downloads is 30. The tiling factor depends on the image size. Following our benchmarks, we expect an average tiling factor of 40. These values add up to 4000 function calls. In addition to this, the list function is called once every time a user wishes to see all the images he has previously uploaded. If we expect 5 of those calls per day, then at the end of the month there were 150 calls made. All of this adds up to a monthly cost of \$0.00083.
2. The total amount of time all the functions were running. The price depends also on the amount of memory allocated to the function. The price is \$0.00001667 for every GB-second used. Because we only allocate 128MB to each Lambda function, the price goes down to \$0.00000208/second. We expect our lambda functions to run no longer than 1 second based on observations made during benchmarking. We make 4150 calls per month so we can expect a monthly expense of \$0.0086.

API Gateway

The API Gateway pricing is based on the following two components :

1. The number of API calls being made. The cost is \$3.5 per million API calls received. Because there is a one-to-one correspondence between API calls and Lambda function calls, the number of API calls is the same as the number of Lambda function calls. The cost amounts to \$0.0145/month.
2. The amount of data transferred out. The first 10 TB cost \$0.09/GB. The amount of data transferred out corresponds to the amount of data downloaded which is 30GB/month. The final cost is \$2.7/month.

The following table summarizes the costs of the different services for the second solution.

S3	\$0.69
Lambda	\$0.00943
API Gateway	\$2.7
total	\$3.4

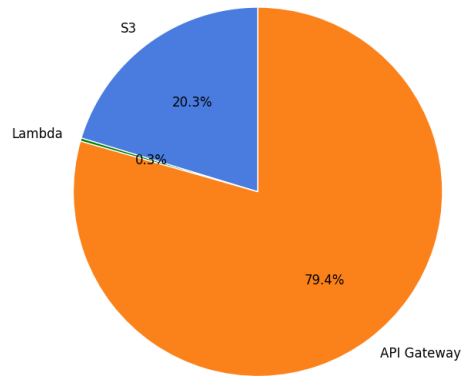


Figure 3.15: Repartition of the cost between the different services

In terms of cost estimations, we can see a clear difference between both solutions. Even if we remove the expensive parts of our solution which would result in less flexibility but also a lower cost, the second solution still remains the best on many aspects.

Firstly it is largely inexpensive costing less than \$5 where our first solution cost more than \$10.

Secondly it is flexible, a large increase in users and requests would not require modifications in our implementation.

Thirdly it is well suited to our use case. The fact that we only expect a small and occasional use of the solution plays an important role in making the first solution too expensive because instances are running all day in a mostly idle state. On the other hand, for the second solution, with lambda functions we pay only for the time they are running which is much better suited to our use case. From a performance point of view, tiling the image as it is done in our second solution is much faster than sending the whole image with no tiling.

Conclusion

We have arrived at the end of this master thesis in Computer Science and it is now time to conclude by reflecting on the work that was done. As was presented in section 2, two solutions were created. The first one follows a 3-tier architecture where the presentation tier is a desktop app and the logic and data tier are implemented in the Cloud using tools such as EC2 to implement a web server and S3 for storing the images. The second solution also has a frontend which is a desktop app but the backend uses a microservice architecture where the different functionalities are independently managed. Furthermore the frontend takes advantage of this architecture by tiling the image before sending it in order to increase parallel processing on the backend side.

One might question the relevance of implementing two distinct solutions. Why not just implement one and make it as good as possible? The idea of doing a new implementation surfaced in our minds a few months into the second semester. At this point in time the first solution already had a solid foundation and was able to handle the functionalities we wanted it to have. However the implementation felt quite generic and lacked innovative concepts. We started thinking about how we could improve on this to come up with something that better fits our use case of having very large images. Eventually after doing some research, a new architecture started taking shape in our minds that would use tiling in order to process small images independently. In addition to the improvement that could be achieved through the implementation of a better solution, having two solutions provided us with the opportunity of performing a comparative analysis of two distinct Cloud implementations.

Let us look back at the goals we formulated at the beginning of the thesis and see if we were able to reach them.

" (...) The focus of this thesis will be to develop a solution for remote storage of such images using Cloud Computing resources and find a trade-off between local and remote processing of the images. Given the large file sizes, image compression is necessary and we will study what algorithms are available to handle such a use case. It will also be important to take into account the security aspect of the solution given the confidentiality that is required for such data. "

In terms of trade-off, during our benchmarks we performed some analysis that helped us to decide where most of the processing should be shifted. For the first solution we found that client-side compression of the images was the best choice as it would significantly decrease the bandwidth required to send the images and thus allow for faster reception by the server. For the second solution we were mostly concerned by deciding what would be the optimal tiling factor. We found that a tile of dimension 630x630 pixels represented a good trade-off between the amount of work required to tile the image and the time it takes to send the sub-images. Not enough tiling means sub-images can be slow to send but too much tiling is not useful as a computer is limited in its capability to send many requests in parallel.

In terms of security both solutions have been focused on implementing good security practices.

The first solution makes use of standard user authentication with the server to establish a user session, HTTPS communication and has an elaborate set of security rules to control access to our Cloud infrastructure and the data it contains. The second solution uses a more modern approach to user authentication through JWT retrieval and validation. The client app of the second solution connects to the API using HTTPS natively.

In terms of the functionalities, our solution should be able to handle, the introduction stated the following :

"The features we will be focusing on implementing are the following :

1. The software should have an authentication system to control who has access to the core functionalities;
2. An authenticated user should be able to view a list of all images he has access to;
3. An authenticated user should be able to upload an image to the Cloud;
4. An authenticated user should be able to download an image from the Cloud."

Both implementations satisfy these requirement but with a different level of success. As shown in the benchmarks the first solution is significantly slower than the second. This is mainly due to the fact that it is too generic and not well suited for the task that was at hands here. The bottleneck was clearly identified as being the sending of the image to the server. The second solution improves on this by tiling the images before sending them. Sending many small images is much better than one big image. Furthermore our backend architecture is well suited for this task because each sub-image can be processed independently from the others thanks to the use of the Lambda service.

In terms of the cost of both solutions, we have seen that the second solution is much cheaper than the first one. Again this is largely due to the fact that the architecture of the second solution is better suited to the way our solution will be used. While for the first solution we must pay for servers that are running non-stop, for the second solution we only pay when a user interacts with our app which is much cheaper if our app is only used occasionally.

Finally, let's take some time to consider what future improvements could be made to the second solution. For a start, it would certainly benefit from a more friendly interface with more visual cues, for instance having a loading widget that shows that images are being uploaded. Making the solution more robust to errors is also something that should be considered. As we have explained earlier, for this thesis we focused on laying out the foundations of the 2 different architectures as we thought this would be the most interesting. It allowed us to get our hands on a variety of tools and frameworks and become familiar with two different ways of designing an architecture. This thesis was also a good opportunity to deepen our understanding of the Cloud infrastructure and realise how useful it can be.

Bibliography

- [1] Julien Meyer, Guy Paré, *Telepathology Impacts and Implementation Challenges, A Scoping Review*. Arch Pathol Lab Med—Vol 139, December 2015.
- [2] Yukako Yagi, John R Gilbertson, *Digital imaging in pathology : the case for standardization*. University of Pittsburgh Medical Center, Pittsburgh, PA, USA. Journal of Telemedicine and Telecare Volume 11 Number 3, 2005.
- [3] J.P. Ortiz, V.G. Ruiz, and I. Garcia, *Efficient Virtual Slide Telepathology Systems with JPEG2000*. University of Almeria, Spain.
- [4] Vilppu J. Tuominen and Jorma Isola, *The Application of JPEG2000 in Virtual Microscopy*. Journal of Digital Imaging, Vol 22, No 3 (June), 2009.
- [5] Charilaos Christopoulos, Athanassios Skodras and Touradj Ebrahimi, *The JPEG2000 Still Image Coding System: An Overview*. IEEE Transactions on Consumer Electronics, Vol. 46, No. 4, November 2000.
- [6] D. Eastlake, T. Hansen, *RFC 6234, US Secure Hash Algorithms*. May 2011. Consulted on 4/17/17.
- [7] H. Krawczyk, M. Bellare, R. Canetti, *RFC 2104, HMAC: Keyed-Hashing for Message Authentication*. February 1997. Consulted on 04/17/17.
- [8] Michael Armbrust, Armando Fox and Al., *Above the Clouds: A Berkeley View of Cloud Computing*. UC Berkeley Reliable Adaptive Distributed Systems Laboratory, February 10, 2009.
- [9] Amazon Web Services, *Overview of Security Processes*. October 2016.
- [10] Amazon Web Services, *AWS Security Best Practices*. August 2016.
- [11] Ken Beer, Ryan Holland (Amazon Web Services), *Encrypting Data at Rest*. November 2014.
- [12] Vineet Badola, *AWS AMI Virtualization types : HVM vs PV*. (15 December 2015), cloudacademy.com. Consulted on 04/17/17
- [13] *A new image format for the Web*, (March 4 2016) <https://developers.google.com/speed/webp/>. Consulted on 04/20/17
- [14] Aleks Jakulin, *Baseline JPEG and JPEG2000 Artifacts Illustrated* <http://www.stat.columbia.edu/~jakulin/jpeg/artifacts.htm>. Consulted on 04/20/17
- [15] Amazon Web Services, <https://aws.amazon.com/>
- [16] Amazon EC2 Instance Types, <https://aws.amazon.com/ec2/instance-types/>
- [17] Auto Scaling, <https://aws.amazon.com/autoscaling/>

- [18] Amazon Elastic Load Balancing, [texttthttps://aws.amazon.com/elasticloadbalancing/](https://aws.amazon.com/elasticloadbalancing/)
- [19] Amazon Virtual Private Cloud, <https://aws.amazon.com/vpc/>
- [20] D. Cooper, S. Santesson and Al. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*, May 2008, Consulted on 05/10/2017
- [21] AWS Cloud Compliance, <https://aws.amazon.com/compliance/>
- [22] AWS Key Management Service, <https://aws.amazon.com/kms/>
- [23] Qt : Software development made smarter, <https://www.qt.io/>
- [24] Kivy : Open source Python Library, <https://kivy.org/>
- [25] Django : The web framework for perfectionists with deadlines, <https://www.djangoproject.com/>
- [26] The Apache Software Foundation, <https://www.apache.org/>
- [27] Amazon S3 Pricing, <https://aws.amazon.com/s3/pricing/>
- [28] Marco Canini, *Certificates, SSL/TLS, BGP*, course slides Spring 2016, INGI2347 : Computer System Security, Université Catholique de Louvain
- [29] IntoPIX s.a., *JPEG2000 handbook*, 2008

Acronyms

ACL	Access Control List. 20
AES	Advanced Encryption Standard. 24
AMI	Amazon Machine Image. 11
API	Application Programming Interface. 19
ARP	Address Resolution Protocol. 13
AWS	Amazon Web services. 10
CA	Certification Authority. 14
CRC	Cyclic Redundancy Check. 27
CSE	Client-Side Encryption. 22
DCT	Discrete Cosine Transform. 4
DNS	Domain Name System. 13
DWT	Discrete Wavelet Transform. 4
EBS	Elastic Block Storage. 10
EC2	Elastic Cloud Compute. 10
ECU	EC2 Compute Unit. 10
ELB	Elastic Load Balancer. 11
FaaS	Function as a Service. 35
GIF	Graphics Interchange Format. 5
HMAC	Hash-based Message Authentication Code. 14
HSM	Hardware Security Module. 23
HTTP(S)	Hyper-Text Transfer Protocol (Secure). 16
HVM	Hardware Virtual Machine. 11
IaaS	Infrastructure as a Service. 8
IAM	Identity and Access Management. 18
JPEG	Joint Photography Expert Group. 3
JSON	JavaScript Object Notation. 37
JWT	JSON Web Token. 37
KMI	Key Management Infrastructure. 22
LAN	Local Area Network. 18
MFA	Multi-Factor Authentication. 18

NIST	National Institute of Standards and Technology. 8
PaaS	Platform as a Service. 8
PNG	Portable Network Graphic. 5
RDS	Relational Database Service. 20
RGB	Red Green Blue. 3
RLE	Run Length Encoding. 5
RTT	Round Trip Time. 43
S3	Simple Storage Service. 19
SaaS	Software as a Service. 8
SHA	Secure Hash Algorithm. 13
SSE	Server-Side Encryption. 22
SSH	Secure SHell. 20
SSL	Secure Socket Layer. 16
TBS	To Be Signed. 14
TCP	Transmission Control Protocol. 13
TLS	Transport Layer Security. 16
URL	Uniform Resource Locator. 15
VPC	Virtual Private Cloud. 12
WSGI	Web Server Gateway Interface. 31
WSI	Whole Slide Imaging. 1

