MASTER IN COMPUTER SCIENCE

A comprehensible pattern-oriented approach to understanding cloud and distributed architecture's challenges

Malcourant, Anthony

Award date:
2016

Awarding institution:
University of Namur

Université de Namur
Faculté d'informatique
Année académique 2015-2016

A COMPREHENSIBLE PATTERN-ORIENTED APPROACH TO UNDERSTANDING AND ACHIEVING CLOUD AND DISTRIBUTED ARCHITECTURE'S CHALLENGES

Anthony Malcourant

Promoteur: Philippe Thiran
(Signature pour approbation du dépôt - REE art. 40)

Mémoire présenté en vue de l'obtention du grade de
Master en Sciences Informatiques

Abstract

Cloud Computing has grown in popularity over the past decade and has become more and more competitive and widespread in large and medium companies. Unfortunately, Cloud is every so often misunderstood, even by IT professionals. This misunderstanding often leads to poor implementation and non-optimal use of the various available techniques and tools.

This thesis aims to present the Cloud and its challenges in a *comprehensible* form and to define ways of tackling and implementing it efficiently. Indeed, thanks to the different available *patterns*, the reader will be guided through an architectural solution to meet all the Cloud characteristics while leveraging the various benefits of Cloud and more generally Distributed Architectures.

This work does not aim to explain how to implement the various Cloud providers solutions but rather give the reader a set of concepts and rules that should be kept in mind during an introduction of Cloud solutions. Concepts and patterns are presented in a narrative and incremental way in order to progressively introduce the complexity of the subject.

Le Cloud Computing a évolué en popularité depuis les dix dernières années et est devenu de plus en plus compétitif et étendu dans les grandes et moyennes entreprises. Malheureusement, le Cloud est encore parfois mal compris, et ce même par des professionnels de l'informatique. Cette imcompréhension mène parfois à une mauvaise implémentation et à une utilisation non-optimale des différentes ressources et techniques disponibles.

Cette thèse a pour objectif de présenter le Cloud et ses défis de manière compréhensible et de définir les moyens pour l'intégrer et l'implémenter efficacement. En effet, au travers des différents patterns disponibles, le lecteur sera guidé au travers d'une solution architecturale qui répond à l'ensemble des prérequis tout en exploitant les différents bénéfices du Cloud et, plus généralement, des systèmes distribués.

Ce travail n'a pas pour objectif d'expliquer comment mettre en place les solutions des différents fournisseurs de solution Cloud mais plutôt de donner au lecteur un ensemble de concepts et de règles qui sont importants à garder à l'esprit lors d'une introduction aux solutions Cloud. Les concepts et les patterns sont présentés de manière narrative et incrémentale de sorte à introduire la complexité du sujet de manière progressive.

Keywords: C sloud, challenges, pattern-oriented, comprehensible, narrative.

Foreword

I would like to thank the following people, without whose help and support this thesis would not have been possible. Most of all, I would like to show my appreciation and gratitude for the involvement of my supervisor, Mr. Philippe Thiran, who kept an eye on the shaping of the thesis. Moreover, I would like to thank Mr. André Füzfa who gave me the opportunity to finally complete my Master by accepting my submission. I would also like to thank Ms. Benjamine Lurquin for her practical information and help. Finally, I wish to thank my friends, my parents and my girlfriend for their constant support and advices.

Table of contents

1 Introduction

The Cloud sounds familiar to almost anybody with notions of Internet Technologies. Unfortunately, there is a big confusion around the Cloud and its concepts. Sometimes defined as an infrastructure provider, as a marketing technic [67] or as a bunch of on-demand applications, it is quite complex to have a complete view of what the Cloud really is, what it provides and particularly of what it implies. Often considered as being the new "Big Thing" [73], the Cloud has gained in popularity and is regularly implemented and used in companies whatever their size. However, adopting such a paradigm contains risks. Jumping into the Cloud adventure without a deep comprehension and an exhaustive architectural vision and all its challenges often leads to failure.

Based on observations from the existing literature, one can easily notice that articles fail to provide readers with a practical and comprehensible approach. This thesis is a first attempt to provide information about the Cloud and more precisely, to explain how to efficiently tackle it in an architectural point of view. The thesis is set up in multiple parts which are organised incrementally to guide the reader progressively through the *Cloud Computing* complexity. This thesis also undertakes to offer an alternative to the style broadly used in classical pattern-oriented books by using a narrative approach. The reader consequently travels from global to more fine-grained issues like in a story book. However, a classical and structured form is provided as an appendix at the end of this thesis.

Chapter 2 intends to define Cloud Computing and explain in details the definitions broadly used in the literature. Some of the main benefits of adopting the Cloud will be listed as well as its five essential characteristics. As the goal is to provide a comprehensible approach, we will add some widely established concepts of distributed architectures which develop and explain these characteristics. We will then define the Cloud delivery models and deployment models to understand what the Cloud really offers. Finally, we will go through Cloud challenges and see how they can affect its characteristics.

Chapter 3 is a short introduction about patterns. It gives the reader a complete definition of what a pattern is and what it stands for. It also defines the pattern canvas which is used in the technical pattern section. This canvas aims to describe the pattern of Chapter 4 in a generic way, allowing the reader to deal with each pattern with the same approach and methodology.

Chapter 4 introduces the patterns in a narrative way. Indeed, the reader is invited to ask himself questions about the practical manner of implementing the Cloud in order to maintain and improve its characteristics. Each pattern is introduced by a question that initiates the interrogation of the reader. We then try as much as possible to answer this question by providing solution(s) and discussing them. Then, we will see how such solutions could impact positively or negatively the Cloud challenges introduced in Chapter 2 and really understand why they are so ambitious. The reader is therefore guided narratively through a series of questions-answers that indirectly give him insights of the Cloud and its concepts.

We will finally end at Chapter 5 with the conclusion of this thesis. We will summarize what readers have learned and how it could help them for further studies about *Cloud Computing*. We will also present the limitations of this thesis as well as the lines of thought to go a step further.

1 Cloud computing

1.1 Definition

One can easily gather a hundred of *Cloud Computing* definitions over the Internet. However, the National Institute of Standards and Technology (NIST) definition has received industry-wide acceptance and is therefore often used as a reference in many Computer Sciences books. The NIST published its original definition back in 2009, which was followed by multiple revised versions after further reviews and industry inputs. In September 2011, they finally defined *Cloud Computing* as follows [2]:

> "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. This cloud model is composed of five essential characteristics, three service models, and four deployment models."

As a more concise definition example, we can quote Thomas Erl's definition [3] :

> "Cloud computing is a specialized form of distributed computing that introduces utilization models for remotely provisioning scalable and measured resources."

These definitions may still appear as a bit complex and therefore not clearly understood at this point. Indeed, it introduces new concepts which will further be covered.

1.2 Goals and benefits

Cloud advantages are numerous and well recognized by IT professionals. The financial benefits of the Cloud are often highlighted [70] which tends to make it THE solution to acquire. However, *Cloud Computing* is neither the best solution in every case nor the cheapest one. Joe Weinman believes that most people use simplistic models to evaluate the economics of the Cloud. The standard argument is that considerable providers (such as Amazon, Microsoft and

Google with their proprietary solutions [69]), achieve large economies of scale and thus will be cheaper than a "do-it-yourself" approach [4]. However, he argues, this is neither a necessary nor a sufficient condition for Cloud computing to be valuable for companies. His example is quite meaningful: "After all, people rent cars all the time, at a unit cost per day much higher than that of owning". Similarly, he argues that the true cost reduction value of Cloud infrastructure has nothing to do with lower unit cost, but with a no commit, *pay-per-use* model. "In effect, it doesn't matter that much what you pay when you use Cloud services, the key cost reduction driver is what you pay when you don't use them: zero." [5].

The Weinman economic vision is probably the more realistic one. However, here are the main global benefits a company can get by switching to *Cloud Computing*.

1.2.1 Reduced investments and proportional costs

The first benefit of the Cloud is, according to what was previously said, its ability to reduce the costs. This benefit is the most famous one because it is often used in businesses to promote its use and to convince Cloud consumers. A Cloud consumer is considered as an entity (an organization or a human) which has an arrangement with a Cloud provider and therefore who uses resources and services it offers (*Figure 2.1*). Consumers and Providers are considered as the two main actors of the *Cloud Computing*.

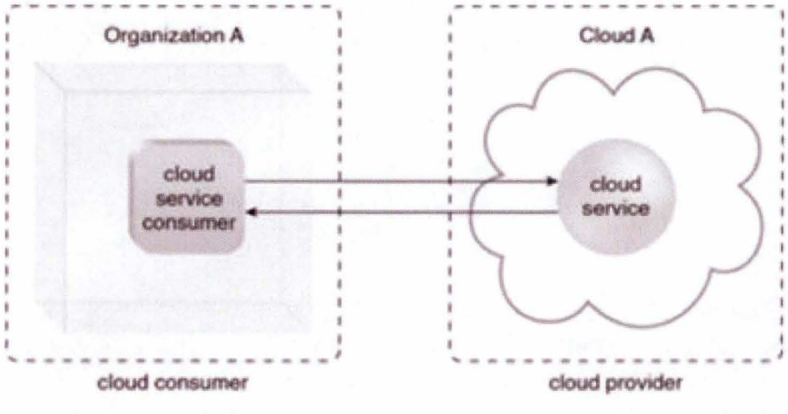


Figure 2.1- Cloud consumer and Cloud provider.

Cloud providers and Cloud consumers both benefit from this cost reduction ability. Cloud providers lower their prices by the mass-acquisition of cheap computer resources. An interesting quote of Thomas Erl [6] defines it perfectly : "Similar to a product wholesaler that purchases goods in bulk for a lower price points, public Cloud providers base their business model on the mass-acquisition of IT resources that are then made available to Cloud consumers via attractively priced leasing package. This opens the door for organization to access powerful infrastructure without having to purchase it themselves." Moreover, Cloud Computing introduces the concept of *commodity computing*. The purpose of commodity computing is to

utilize large numbers of readily available computing components for parallel computing to obtain the greatest amount of useful computations for the least price [7]. Such systems are said to be based on *commodity* components, since the standardization process promotes lower costs and less differentiation between products [38]. They are based on standards and often outdated components. *Scalability*, which is defined below, is therefore possible at a lower price.

1.2.2 Increased Scalability

Scalability is the ability for a system to meet the increasing or decreasing workloads by increasing or decreasing its capabilities[1]. There are two kinds of *scalability*: *vertical* and *horizontal*. *Figure 2.2* illustrates the difference between them.

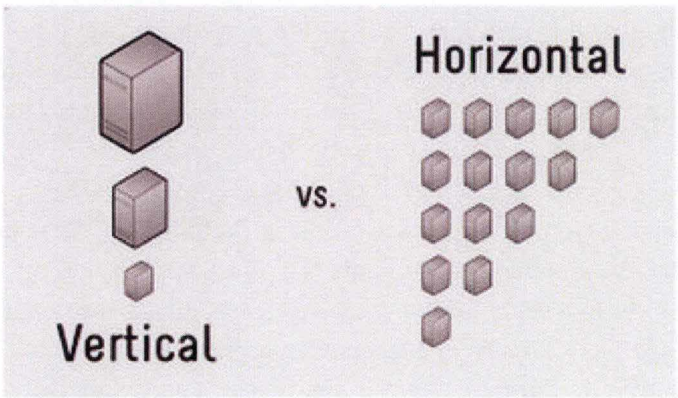


Figure 2.2 - Vertical versus Horizontal Scaling.

On the one hand, *Horizontal Scaling* is the ability to connect multiple resources so that they can work together as a single logical unit called a *cluster*. On the other hand, *Vertical Scaling* increases the capacity of the existing hardware by adding more capacity (processing, storage, bandwidth, etc.). *Horizontal scaling* is more complex and often requires more resulted architectural concepts. It is this kind of *scalability* which is discussed across this thesis.

By providing pools of resources, along with tools and technologies designed to leverage them collectively, Cloud providers can instantly and dynamically allocate IT resources to Cloud consumers, on-demand or via manual configuration. This allows Cloud consumers to scale their cloud-based IT infrastructure to accommodate their needs and face peaks [39].

Vertical scaling	Horizontal scaling
More expensive	Less expensive
IT Resources normally instantly available	IT Resources instantly available
Limited hardware capacity	Not limited by hardware capacity

[1] Capacities and abilties

No Additional IT Resources needed	Additional IT Resources needed

Table 2.1 - Vertical versus Horizontal Scaling.

1.2.3 Increased Availability and Reliability

In addition to the *Scalability* and financial benefits of the Cloud, Nucleus's benchmarking effort shows that migrating existing workloads from on-premise ("do-it-yourself" in your building) environments to AWS or any other Cloud providers offers Cloud consumers significant infrastructure *reliability* and application *availability* benefits [8].

Availability (or High Availability) is one of the biggest challenge and, ultimately, the holy grail of the Cloud. It embodies the idea that services are available at anytime, from anywhere. *Availability* is also linked to *Reliability.* Indeed, can a service constantly available but failing still be considered as available? Of course not, a service is considered highly available when it is constantly up and running. *Availability* is often defined by Cloud providers SLAs[2] and indicated as a percentage of availability over a period of time. For example, in its SLA, Amazon declares availability with a Monthly Uptime Percentage of at least 99.95% [25]. More precisely, services cannot fail more than 30 minutes in a month. To illustrate the current Cloud providers availabilities, we can refer to a study [45] about downtime statistics that gives us a cumulative downtime from 2007 to 2013 for the 39 biggest Cloud providers and solutions. The total of downtime for these providers during 6 years is significant: approximatively 2200 hours, that being less than 0.001%.

1.2.4 Ease Team Tasks

Cloud providers offer a series of tools to support the entire software lifecycle [46,47] which profits for a certain type of Cloud consumers: the technical teams. For example, Operations team can benefit from monitoring and metrics tools and developers can easily multiply environments.

In a traditional way, developers usually develop their code locally on their own machine, have access to a shared database and so forth. This behaviour multiplies the environments and it becomes difficult to enforce the *Consistency*: each developer has it own version of the core language, its own OS system with its own update version. This kind of issue could be solved by always installing the same environment (virtual image) and it is possible through Cloud technologies such as Vagrant, Chef, Puppet or Docker [10]. Of course there are a lot of other benefits: testing, deployments, repositories, demonstrations, queuing but it will not be approached in this thesis.

---

[2] *Service Level Agrement: SLA*

1.3 Characteristics

In its final definition of Cloud Computing, The NIST defines the Cloud as a model composed of *five essential characteristics* [22]. The following sub-chapters aim to explain them in a more detailed way by incorporating additional explanations. Indeed, the NIST usually uses complex concepts without explaining them.

Individually, these Cloud characteristics may already be known and are often available in different well-established products and services, for example, server hosting solutions or public Web applications. It is the combination of these concepts and techniques in addition to a significant improvement of Internet connectivity and data transfer speed that distinguishes *Cloud Computing* from existing products and services and which justifies calling it the "new big thing" [23].

1.3.1 On-demand self-service

Cloud providers enable Cloud consumers to provision computing resources (storage, calculation power, etc.) when they need it. This is made by self-service or automation. Self-service means that Cloud consumers are able to perform all the actions needed to acquire these resources themselves, instead of going to the IT department or calling the Customer Service. As shown in *Figure 2.3,* self-service capabilities are usually provided through online interfaces which abstract all the complexity and make it quite simple for non-technical profiles. This acquiring is also possible in a automated manner but we will discuss it later when we will introduce the concept of *Elasticity.*

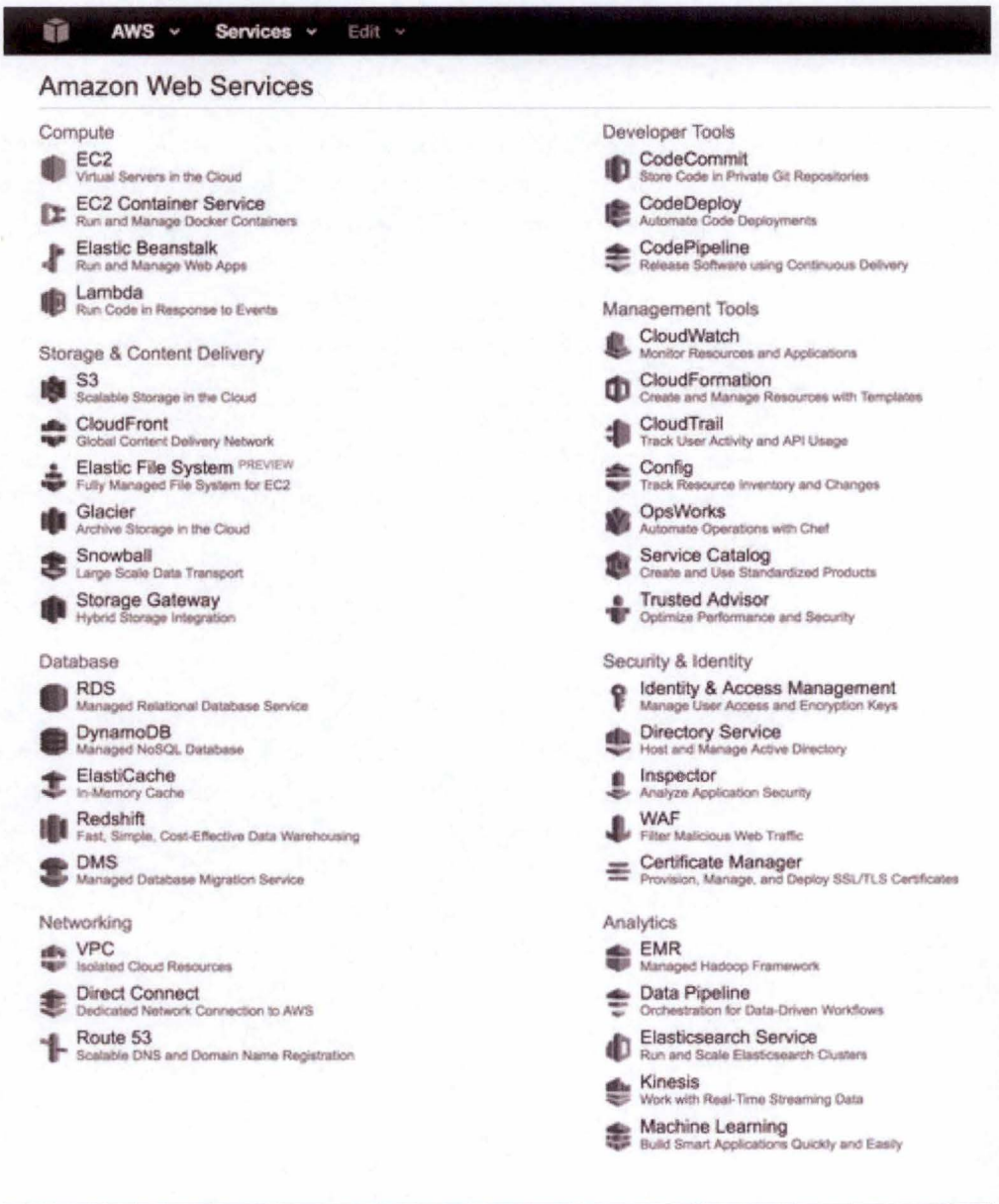


Figure 2.3 - Amazon EC2 remote admin panel

1.3.2 Broad network access

"Capabilities are available over the network and accessed through standard mechanisms that promote use by heterogeneous thin or thick client platforms" [2]. Services are now accessed from anywhere, from any kind of devices and with any kind of protocols. This characteristic looks quite generic but the idea behind is quite simple. Cloud is accessed from a lot of devices which connect from any place across the world as illustrated in *Figure 2.4*.

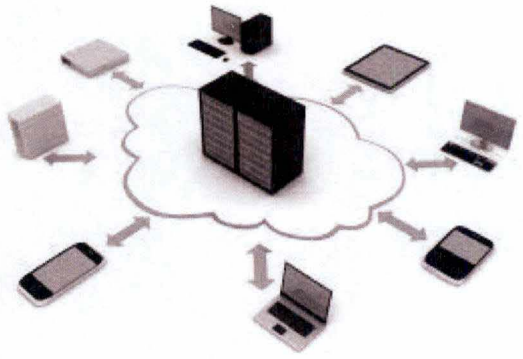


Figure 2.4 - Broad Network access illustrated

1.3.3 Measured service

Cloud providers control and optimize resource use by leveraging a metering capability at different levels of abstraction according to the type of services. Cloud provider offers are discussed later in the delivery models. The metric can be, for example, the number of requests made on a specific service, the response time the service takes, and so forth. Resource usage can be managed, controlled and reported providing transparency for both the Cloud provider and the Cloud consumer. The *pay-per-use* concept is only possible because of this measurable capacity.

1.3.4 Resource pooling

"The provider's computing resources are pooled *(A)* to serve multiple consumers using a multi-tenant *(B)* model, with different physical and virtual *(C)* resources dynamically assigned and reassigned according to consumer demand. There is a sense of location-independence in that the customer generally has no control or knowledge over the exact location of the provided resources but may be able to specify location at a higher level of abstraction (e.g., country, state, or data center)" [2].

This explanation contains too many information to be clearly understood when not familiar with *Cloud Computing*. The NIST introduces in its sentence three new concepts which are important to clarify in order to fully understand the resource pooling: the notion of clustering, multi-tenancy and virtualization. The following sub-chapters will provide readers with further explanation.

A. Clustering

A *cluster* consists of a set of loosely or tightly connected computers that work together so that they can be considered as a single unit [11]. Each connected computer could be referred to as a *node* or as an *instance* (*Figure 2.5*). There are multiple benefits from working with clusters. According to IBM [12], clustering could offer benefits such as:

- High processing capacity,
- Resource consolidation,
- Optimal use of resources,
- Geographic server consolidation,
- 24 x 7 availability with failover protection,
- Disaster recovery,
- Horizontal and vertical scalability without downtime.

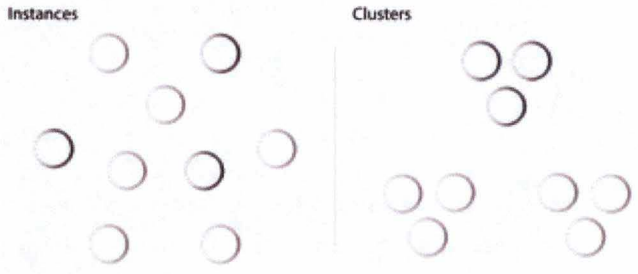


Figure 2.5 - Nodes versus Clusters

B. Multi-tenancy

In *Cloud Computing*, the meaning of *multi-tenancy* architecture has broadened because of new service models that take advantage of *virtualization* and remote access. "A *tenant* is any application that needs its own secure and exclusive virtual computing environment" [67]. The concept of *multi-tenancy* enables to run one instance of an application and provide the access to multiple users. In such a scenario, each *tenant's* data is isolated and remains invisible to others (*Figure 2.6*).

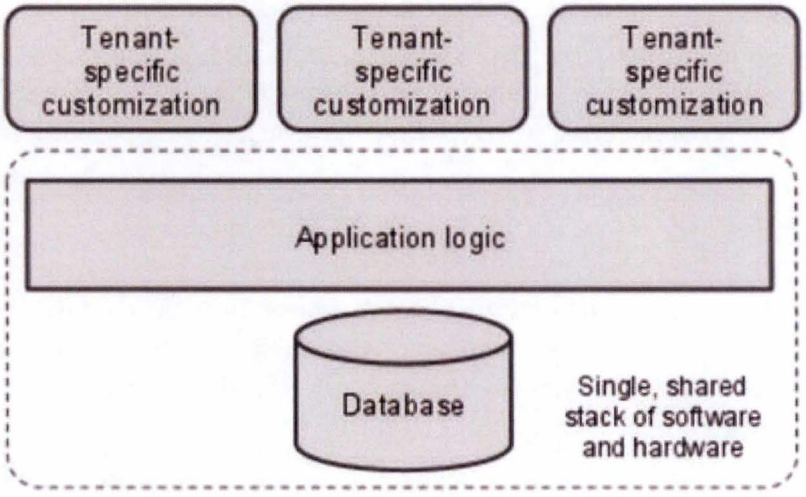


Figure 2.6 - Multiple tenants isolation

C. Virtualization

Virtualization represents a technology platform used for the creation of virtual instances of IT resources. A layer of *virtualization* software allows physical IT resources to provide multiple virtual machines to themselves so that their underlying processing capabilities can be shared by multiple users [3].

At the heart of the *virtualization* stands the virtual machine which can be compared to an isolated container wherein is placed an operating system and a series of applications. The word *isolated* is important because each virtual machine has to be completely separated and independent even if a lot of them are running at the same time on the same computer or more generally speaking on the same physical instance. The *Hypervisor* also called Virtual Machine Monitor (VMM) is responsible for dynamically allocating IT resources and distributing them to virtual machines when they need it [56]. The *Hypervisor* depicted in *Figure 2.7* is named VMware ESX Server [58] and is a VMWare product.

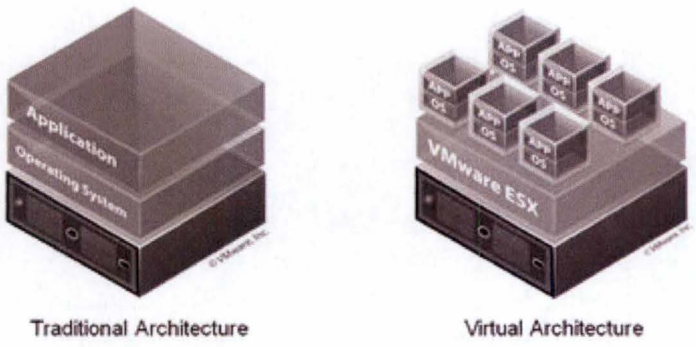


Figure 2.7 - Traditional versus Virtualized Architecture

Most Cloud providers provide pre-made virtual images. For example, AMI (Amazon Machine Image) is proposed by Amazon with a variety of operating systems and pre-installed applications. It is also possible to create its own image with everything needed inside [57]. Most types of IT resources can be virtualized including servers, storage, network, and so forth.

1.3.5 Rapid elasticity

"Capabilities can be rapidly and elastically (A) provisioned, in some cases automatically, to quickly scale out and rapidly released to quickly scale in. To the Cloud consumer, the capabilities available for provisioning (B) often appear to be unlimited and can be purchased in any quantity at any time" [2].

Once again, NIST uses advanced concepts in its explanation about the Elasticity. The following sub-chapters detail them more precisely.

A. Elasticity

Scalability and *Elasticity* may be a bit confusing because they are often used interchangeably. However, they are quite different even if they share the same purpose which is obviously the *Adaptability*. Managers and deciders have to perfectly understand both terms because their needs and uses in business environments may differ [55].

On the first hand, *Scalability* is only the ability for a system to scale. "*Scalability* is much more specific and gradual than *elasticity* and it is very controlled by the Cloud consumer and his Cloud provider in conjunction with the IT department" [55]. Let us consider the adaptability as manually performed to match the needs. Entreprise can work without *Elasticity* for their production environments. However, it is much more preferrable to work with *Scalability* for the benefits we pointed earlier.

On the other hand, *Elasticity* mainly applies to e-commerce, mobile and web development because of its capacity to adapt "on the fly" without human interaction. *Elasticity* is also a term that was coined to promote and enable metered use which is so prevalent in public Cloud [55]. The distinction should therefore be clearly understood: *Scalability* is more about manual adaptability in private Cloud. Unlike *Elasticity* which is widely used in *public Cloud* where the adaptabily is really needed and automated. We will see the difference between *private* and *public Cloud* at a later stage.

Elasticity is about the adaptability to the *workload*. The term *workload* is used to refer to the utilization of resources on which an application is hosted. *Workload* is the consequence of users accessing the applications or the process that have to be handled automatically. *Workload* exists in different forms depending on the type of resource: Data, Network, Processing, etc. *Workload* has to be measurable in some form because it allows to decide whether we have to increase or decrease the number of IT resources assigned to an application [10]. Without *Workload*, we

would not be able to know the different utilization and thus not able to *scale* and charge Cloud consumers efficiently.

Multiple workload models exist, however, they will not be discussed in this thesis:

- Static workload,
- Periodic workload,
- Once-in-a-lifetime workload,
- Unpredictable workload,
- Continuously changing workload.

B. Provisionning

The term "*on demand*" mentioned in the NIST definition is easy to explain thanks to the previous discussion about *Scalability* and *Elasticity*. On-demand is a principle wherein IT resources are provided on an as-needed and when-needed basis [59]. The term *Provisioning* defines this action of providing resources. *Provisioning* exists in "multiple versions", mainly Cloud consumer *self-provisioning* when resources are allocated manually and *dynamic provisioning* when resources are allocated automatically based on metrics [66].

1.3.6 Resiliency

The *Resiliency* is excluded from the NIST definition. However, according to Thomas Erl, *Resiliency* has emerged as an aspect of significant importance and its common level of support constitutes its necessary inclusion as a common Cloud characteristic [24].

Resilient computing is a form of failover that distributes redundant implementations of IT resources across physical locations. IT resources can be pre-configured so that if one becomes deficient, processing is automatically handed over to another redundant implementation. Within Cloud Computing, the characteristic of resiliency can refer to redundant IT resources within the same Cloud (but in different physical locations) or across multiple Clouds. Cloud consumers can increase both the *Reliability* and *Availability* of their applications by leveraging the *Resiliency* of cloud-based IT resources [60].

1.4 Delivery models

Before diving into the different *delivery models*, it is important to understand the six layers which technically constitue the Cloud (*Figure 2.8*). The model is comparable to the OSI[3] layer model. That makes it quite easy to interpret for IT professionals.

[3] Open System Interconnect

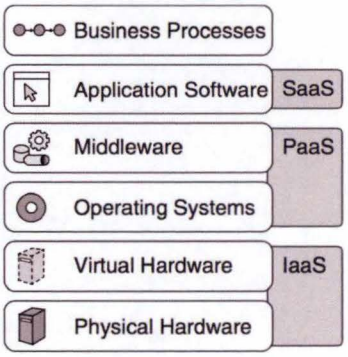


Figure 2.8 - Application Stack and Cloud services models

- **Physical Hardware**. The first layer consists of the basic hardware technologies for almost everything: networking, storage, calculation and so forth.
- **Virtual Hardware**. The second layer is used to share Physical Hardware between multiple virtual counterparts. This ensures that the *Virtualization* is correctly managed and that users perceive the system as if they were the one and only owner.
- **Operating Systems**. The third layer is the software and OS which are installed on top of the virtual or physical hardware layer. Indeed, these systems provide functions and applications to interact with virtual and physical layers.
- **Middleware**: The fourth layer completes the third one by adding more specific softwares which can be called Middlewares. These software can be Execution Environment such as Java or Python. But also more complex Middleware such as IBM Websphere or Messaging systems such as ActiveMQ. Data storage is also included in this layer.
- **Application Software**: The fifth layer consists of custom applications providing functionalities to human users or other applications. Almost every kind of software could stand in this layer.
- **Business Processes**: The sixth and last layer contains the business/domain specific processes. These processes are supported thanks to the use of multiple applications which stand on the previous layer.

Figure 2.8 also highlights the three main *Delivery Models*: IaaS, PaaS and SaaS. These three delivery models could be compared to the three main Cloud provider's offers.

- **Infrastructure as a Service**. *IaaS* refers to on-demand provisioning of infrastructural resources. The Cloud provider which offers IaaS is called an IaaS provider.

- **Platform as a Service**. *PaaS* refers to providing platform layer resources, including operating systems support and software development frameworks.

- **Software as a Service**. *SaaS* refers to providing on-demand applications over the Internet (GMail, Google Docs, Google Drive, etc.).

In Cloud Computing State of the art [13], Qi Zhang, Lu Cheng and Raouf Boutaba from MIT[4] introduce a new point of view of the delivery models. Indeed, they say that according to the layered architecture of Cloud Computing, it is entirely possible that a PaaS provider runs its Cloud on top of an IaaS provider's Cloud. However, in the current practice, IaaS and PaaS providers are often part of the same organization. This is why PaaS and IaaS providers are often called the *infrastructure providers* or *Cloud providers.* This idea recudes the number of models to two as shown in *Figure 2.9.*

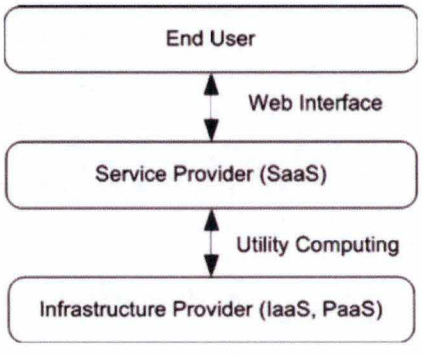


Figure 2.9 - Delivery Models - New point of view

1.5 Deployment models

The NIST definition introduces several Cloud Deployment Models. *Figure 2.10* depicts the different Cloud deployment models according to two factors: level of elasticity and pay-per-use. A *public Cloud* having the most tenants can enable the highest levels of elasticity and *pay-per-use* where only the operational costs are billed to consumers. A *community Cloud* serves fewer tenants, often collaborating companies. An upfront investment may be required by these companies to establish the *community Cloud.* Also, *elasticity* may be reduced as the collaborating companies may experience similar *workloads.* This effect is even more predominant in a *private Cloud* used by only one *tenant* making upfront investments and reduced *elasticity* even more likely. The *hybrid Cloud* spans all these properties as it integrates applications hosted in the different environments. "Note that the properties displayed by *Cloud deployment* types are not generic. A *private Cloud* accessed by a similar large and diverse user group as a *public Cloud* is likely able to present the same properties. *Public Clouds* used only by a few customers that experience similar *workload* will face similar challenges than *private Clouds"* [10]. No distinction is made between the available deployment models in this thesis.

[4] Masachuset Institue of Techonology

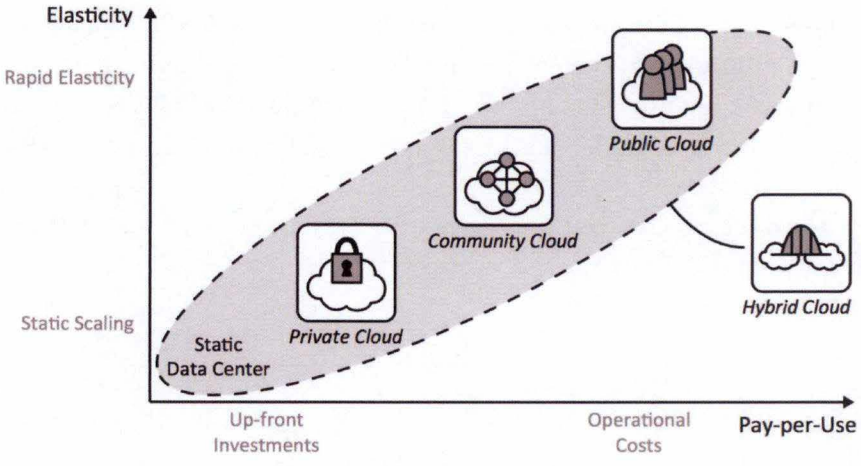


Figure 2.10 - Level of elasticity and pay-per-use of different Cloud deployment types

1.6 Cloud challenges

Although *Cloud Computing* has been widely adopted by the industry, researchs on *Cloud Computing* are still at an early stage. Many existing issues have not been fully addressed, while new challenges keep emerging from industry applications. In this chapter, we will introduce a series of these challenges and then try as much as possible to link them with the Cloud characteristics we previously defined. Indeed, Cloud challenges are not clearly defined in the literature and are rather specific to each business domain and strategy. However, we will try to take a generic point of view and dress a partial map. At the end of this chapter, the reader should be able to define a hierarchy which will be used later to classify the patterns and understand their impact on the general Cloud purposes.

- **Security**: since service providers typically do not have access to the physical security system of data centers, they must rely on the infrastructure provider to achieve full data security. "The moving of business data to the Cloud means that the responsibility over data security becomes shared with the Cloud provider" [71]. Furthermore, in multi-tenancy environment, infrastructure is shared between multiple Cloud consumers and it introduces an overlapping with trust boundaries.

- **Availability** as defined above.

- **Network Latency**: highly scalable and high performance servers do not guarantee that the application will perform well. This is due to the main performance challenge that lies outside of raw computational power: the transfer of data. Transmitting data across a network does not happen instantly and the resultant delay is known as *network latency*. The challenge is really important. Indeed, according to kissmetrics [26], 1 second delay in response page could result to 7% reduction in switching from a fictive buy to a real one.

- **Consistency**: according to Harrap's definition, one thing is consistent with another when it matches or when it fits in with it. To ensure *Resiliency*, data has to be replicated in multiple location. Data alterations have to be propagated properly to every *replica of data* to ensure the *Consistency*. The concept is quite straightforward but the challenge is real.

- **State**: this challenge is a bit more technical but is so important it has to be declared here. The reader should already be familiar with both concepts: *stateful* and *stateless*. It refers to the capacity of something to retain (or not in the stateless case) a state: a bunch of information, a transaction or whatever. Statefull nodes/instances are in opposition with the *Resiliency* characteristic we mentioned earlier. Work around this obstacle is not trivial and demands energy and ingenuity.

- **Reliability**: stands for the ability of a system or a machine to consistently perform its intended or required function or mission, on demand and without degradation or failure. This definition is large and can define at least everything explained earlier. Otherwise, reliability is a keyword the reader has to keep in mind when beginning with *Cloud Computing*. In fact, reliability is one of the most important challenges and everyone should work to make it a characteristic and not a challenge anymore. To give a straightforward example, one of the Cloud characteristic is its ability to provide measured services. As explained in the introduction, Cloud consumers are charged only for their resource consumptions thanks to the pay-per-use model. What would it be if this Monitoring is not reliable and does not charge consumers efficiently?

- **Multitenancy**: is a reference to the mode of operation of software where multiple independent instances of one or multiple applications operate in a shared environment. The instances (tenants) are logically isolated, but physically integrated. The degree of logical isolation must be complete, but the degree of physical integration will vary. The more the physical integration is, the harder it is to preserve the logical isolation. The degree of isolation is the main challenge.

This section shows us how arduous and ambitious can well-designed Cloud architectures and applications be. Indeed, to maintain all the Cloud characteristics, architects have to use many powerful and improved patterns which will be described in this thesis.

All patterns and best practices which are explained here aim to respect and maximize the previous challenges. It could be interesting to add to these challenges the essential Cloud application properties which are defined as *IDEAL*: Isolated state, Distribution, Elasticity, Automated management and Loose coupling [27]. The majority of those concepts are already taken into consideration except for the last one:

- **Loose coupling**: the goal of a loose coupling architecture is to reduce the risk that a change made within one element will create unanticipated changes within other elements. Limiting interconnections can help isolate problems when things go wrong and simplify testing, maintenance and troubleshooting procedures [37].

1.6.1 Cloud challenges impact on Cloud characteristics

Before diving into the pattern chapters, let us try to regroup challenges and show which characteristics they may impact (*Figure 2.11*). We will explain why and how they impact them. This point of view is simplified and personal. Indeed, security could be linked with every characteristic if we use detailed and technical purposes but it is not the purpose of this section.

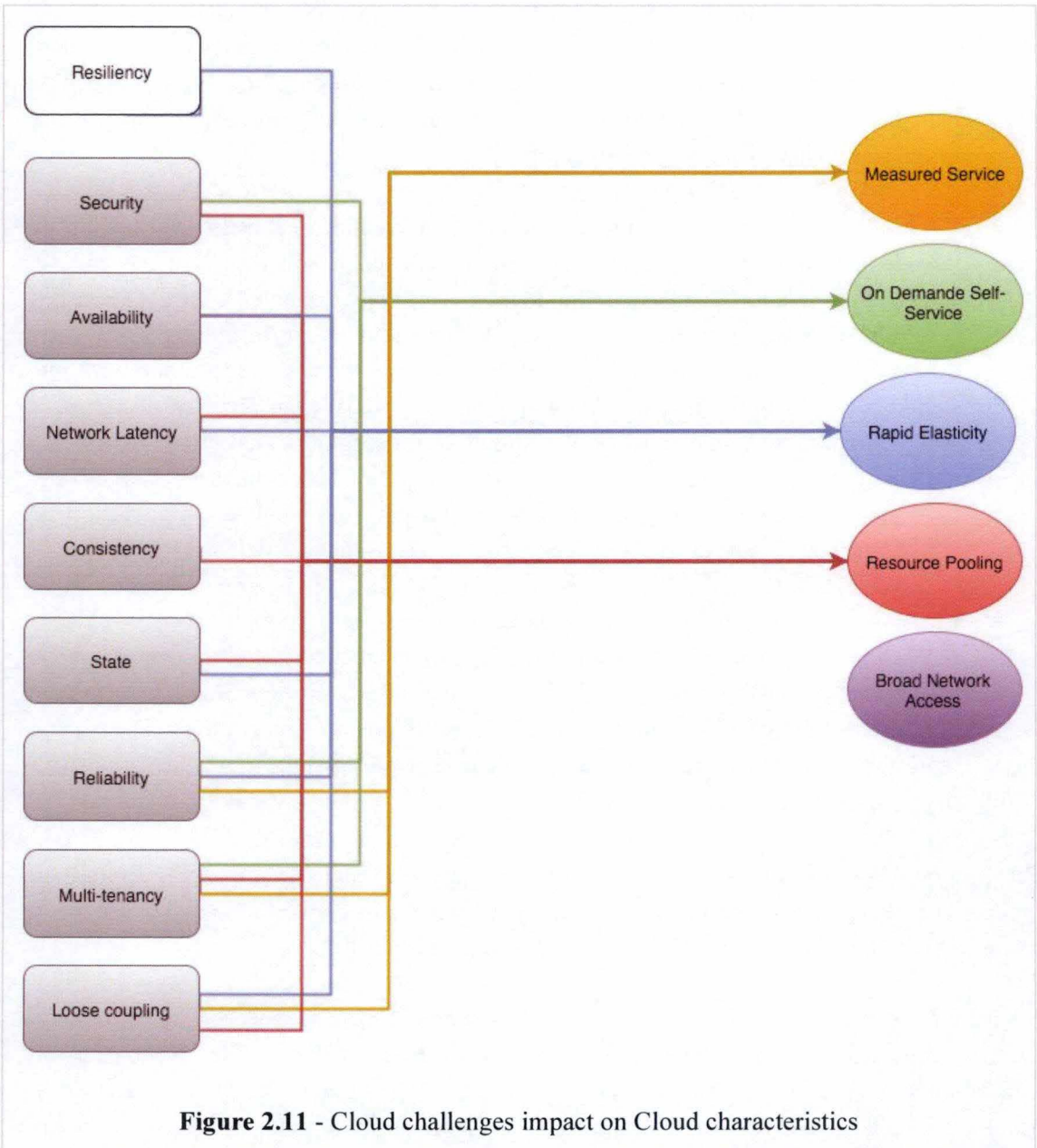


Figure 2.11 - Cloud challenges impact on Cloud characteristics

1.6.2 Broad Network Access

None of our challenges is linked to the Broad Network Access characteristic. Indeed, having a large coverage over network and devices does not depend directly on Cloud technologies. Nowadays, On-site infrastructure and Cloud both have a large diversity of protocols and tools which allow developers and users to use it anywhere and on any device.

1.6.3 Measured Service

- *Reliability.* Metrics and statistics of Cloud solution metering are multiple. The main purpose is to charge Cloud consumers only for what they have actually consumed: the pay-per-use model. The second is to perform Elasticity according to the workload and the user preferences. Having unconsistent metrics will have a negative impact on this characteristic.

- *Multi-tenancy.* Collecting and saving metrics may look quite straigthforward. However, there are multiple considerations that have to be taken into consideration when we deal with multi-tenant environements and therefore when resources are shared.

- *Loose coupling.* Microservices are considered as the new architectural trend [28]. What about the possibility to charge every part independently? Does each service has to integrate a measurable system? How to globally integrate measuring inside a distributed environment?

1.6.4 On-Demand Self-Service

- *Security, Multi-tenancy.* Offering all the capabilities to manage a Cloud infrastructure is technically complicated and may be dangerous, especially in multi-tenant environments. Cloud providers have to provide efficient solutions to prevent Cloud consumers from impacting other tenants in shared environments.

- *Reliability.* Through interfaces Cloud consumers can usually scale by adding or removing resources. This action is led thanks to statistics which are collected and communicated to Cloud consumers. *Reliability* is therefore indirectly linked because of the measure service which is integrated.

1.6.5 Rapid Elasticity

- *Reliability.* As discussed in the Measured Service, Elasticity is enabled by the metrics which are collected and interpreted to know whether or not, the system needs to scale. Having unrealiable data will degrade the efficiency of the Elasticity.

- When the need for provisioning is detected, available resources have to increase rapidly. Such consideration introduces new dependencies:
    - *State* and *Loose coupling*. We know that scaling concerns adding and removing capacities, services or nodes. These nodes have to be rapidly duplicated and available. Keeping each node independent and stateless improves our capacity to add or remove them.
    - *Network Latency* is essential when the process is initialized. Indeed, it is not acceptable for users to wait for 60 seconds before the new node availability. It has to be hidden and the user experience should not be impacted.
    - *Availability* is extremely linked with the *Network Latency*. A resource which does not respond within a certain amount of time may be considered as unavailable.

1.6.6 Resource Pooling

The complexity of the resource pooling is often managed by the Hypervisor. All the complexity and the challenges therefore rely on him. However, it is easy to understand how Cloud challenges could impact it:

- *Network Latency* is important because when it fails, it creates feeling of unavailability. Resource pooling has to be quick in order to avoid the user experience alteration.
- *Multi-tenancy, Loose coupling* and *State* can be regrouped together as they serve a common purpose: separation of concern (SoC).

The reader has been guided through *Cloud Computing* and more precisely through its definitions, benefits, characteristics and challenges. A further sub-chapter unveils how Cloud challenges could impact its characteristics giving the reader a large understanding of the real stakes. Now that the primers and fundamentals are set, we will introduce the purpose of patterns and learn on what they are essentially focused.

2 Understanding design patterns

2.1 Definition

"The simplest way to describe a pattern is that it provides a proven solution to a common problem individually documented in a consistent format and usually as part of a larger collection" [29]. Without acknowledging it, we use patterns in everyday life to solve common issues. In software engineering, patterns are usually called design patterns. There are many types of design patterns and many types of contexts wherein they can be applied.

"Patterns should be prescriptive, meaning that they should tell you what to do. They do not just describe a problem, and they do not just describe how to solve it, they tell you what to do to solve it. Each pattern represents a decision to make. The point of the patterns and the pattern language is to help making decisions that lead to a good solution for a specific problem, even if the initial problem was not entirely the same, and even if the knowledge and experience to develop that solution are not yet known" [30].

2.2 History

Patterns were introduced by Christopher Alexander during the 80's in his book *A pattern language*. The book intended to provide a complete working alternative to our present ideas about architecture, building and planning [31]. "This book provides rules and pictures, and leaves decisions to be taken from the precise environment of the project. It describes exact methods for constructing practical, safe and attractive designs at every scale, from entire regions, through cities, neighborhoods, gardens, buildings, rooms, built-in furniture, and fixtures down to the level of doorknobs" [72]. In fact, Christopher Alexander was just explaining how to solve commun architectural and design (chairs, color, tablement placement, etc.) issues.

In 1994, after the release of *Design Patterns: Elements of Reusable Object-Oriented Software*, design patterns gained in popularity [32]. Nowadays, there are a lot of different books speaking about architecture, development and application design.

2.3 Pattern profile

In *Smalltalk Best Practice Patterns [33],* Ben Kent uses a style that is fairly close to the Alexandrian form. The Alexandrian form is very appreciated due to its style which defines patterns in a prose-like way. As a result, even though each pattern follows an identical,

well-defined structure, the format avoids headings for each individual sub-section, which disrupts the discussion flow. To improve readability, the format uses style elements such as structures, indentation, and pictures to help the reader quickly identify important sections [30]. Our patterns are described as much as possible in the Alexandrian way.

Title	Names the thing created as a result of executing the pattern. Intended to be used conversationally to refer to the pattern.
Pictogram	An image used to illustrate the pattern.
Problem(s)	Stated as a question. Reading the problem will quickly tell the reader whether he is interested in the pattern or not.
Solution and discussion	Gives the reader a concrete recipe for creating the thing named by the title of the pattern. The solution will give him insights to tackle efficiently the initial problem and tell him how to make practical use of the pattern. It may also contain examples of use and further explanations or clues and issues to watch out for.
Impacted challenge(s)	According to the title of this thesis, patterns are explained to help us achieving Cloud Computing challenges. However, this part shows which challenges are positively but also negatively impacted.

Table 3.1- Best Practice Patterns

Patterns are now demystified and their purposes are clear. The following chapter will use the pattern approach in addition to its profile to explore the reader doubts and incomprehensions about the Cloud and its concrete setup. With a pattern-oriented approach, the idea is to take the reader point of view and ask ourself questions that may not have been covered in previous chapters.

3 Patterns

3.1 Introduction

This thesis does not intend to teach the reader how to implement and install a Cloud solution. It only gives him insights on how and why such issues exist and how he places his system in a bad position by neglecting it.

Several patterns include concepts which were briefly introduced in *Chapter 2*. However, we will give a more precise description, mostly by defining potential issues and solutions. Patterns are introduced in a logical way starting from the easiest to the more complex one. Each pattern gives the reader an overview of potential solution(s) and introduces new issues and interrogations. We will also discuss how Cloud challenges are impacted by using such pattern. Indeed, changing the way systems are architectured, impacts and complexifies them.

3.2 Enterprise integration pattern

3.2.1 The need for integration

"Enterprises are typically comprised of hundreds, if not thousands, of applications that are custom-built. These applications are acquired from a third-party, part of a legacy system, or a combination thereof, operating in multiple tiers of different operating system platforms. It is not uncommon to find an enterprise that has 30 different Websites, three instances of SAP and countless departmental solutions" [34]. Creating a single, big application to run a complete business is impossible. That is why distributed application have appeared. In distributed environment, it is possible to execute business functions, regardless of how many internal systems the business function cuts across. In order to support such things, these systems need to be integrated. Application integration has to provide efficient, reliable and secure data exchange between multiple enterprise applications.

The difference between Cloud architecture and distributed architecture is blur. Cloud architecture may be compared to distributed architecture depending on the way it is implemented. We can compare it to the paradigm Interface vs Implementation. Distributed architecture is an interface and the Cloud architecture is one way to deal with it. According to this statement, Cloud Architecture should benefit from *EAI*[5] Patterns. All these patterns were

[5] Enterprise Integration Architecture

defined a long time ago by Gregor Hohpe and Bobby Woolf in their book: "Entreprise Integration Patterns" [34]. This book offers a broad range of patterns such as message transformations, message enrichments, message distributions and so forth. As it is not the purpose of this thesis, the reader is invited to refer to this book if more explanations about integration patterns are needed. *Figure 4.1* illustrates a series of available EAI patterns..

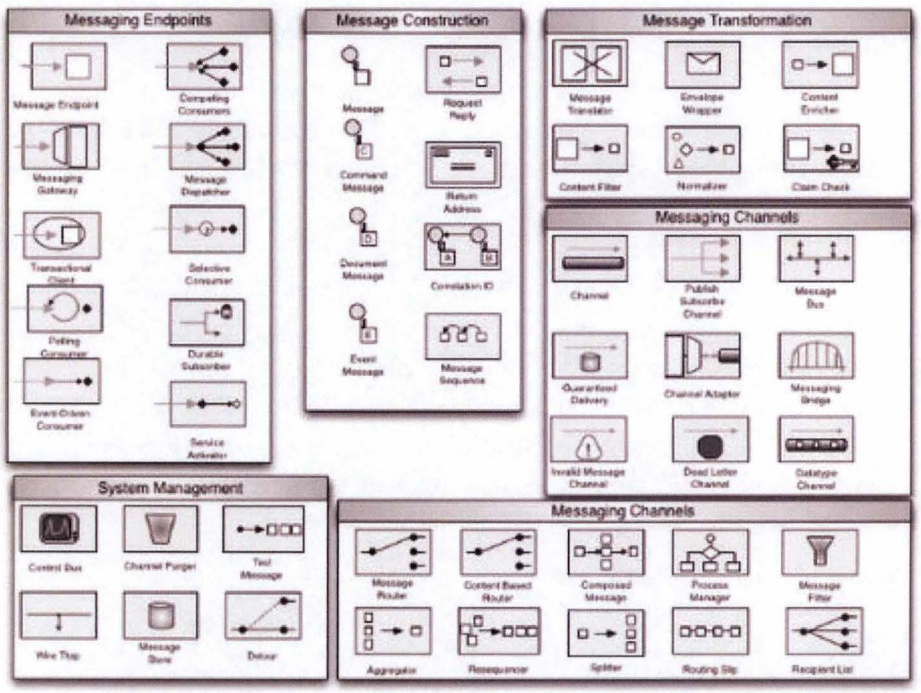
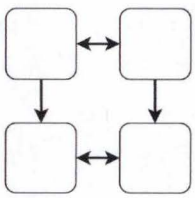


Figure 4.1 - Enterprise Integration Patterns overview

3.3 Divide and Conquer



How is it possible to build a complex application without building a big single one that contains everything? As described in the EAI pattern section, even if it is possible, creating a monolithic application with everything inside, is not recommended at all. Indeed, Cloud applications have to rely on multiple, possibly redundant IT resources to ensure that the unavailability of one IT resource does not affect the application as a whole.

3.3.1 Solution & Discussion

The solution is quite simple: divide and conquer. *Divide and Conquer* is an algorithm design paradigm based on multi-branched recursion. A divide and conquer algorithm works by recursively breaking down a problem into two or more sub-problems (divide), until these become simple enough to be solved (conquer). The solutions to the sub-problems are then combined to provide a solution to the initial problem [35]. There are multiple logical decomposition approaches to split an application into multiple components which will briefly be introduced in the following sections.

3.3.1.1 Layer-based Decomposition

This type of decomposition is well-established in Software Engineering. This approach decomposes the application into separate logical layers. The below example shows the three main layers usually used:

- User Interface (UI),
- Processing (Business),
- Storage (Persistence).

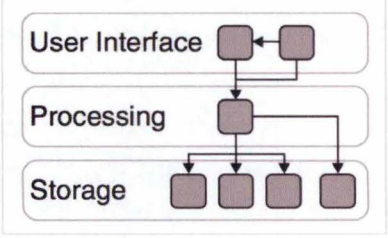


Figure 4.2 - Layer-based

3.3.1.2 Pire-and-Filter-based Decomposition

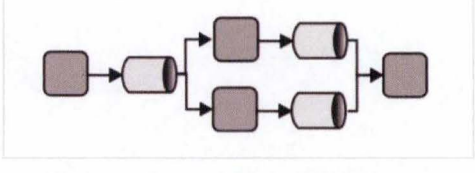


Figure 4.3 - Pipe-and-Filter-based

This type of decomposition is well-established in Enterprise Integration Architecture. Each filter provides a certain function that is performed on an exchange. This processing takes an input exchange and produces an output exchange. All those filters are interconnected with pipes ensuring that the output of one filter is the input of the next filter in a processing chain.

3.3.1.3 Process-based Decomposition

This decomposition is widely inspired by the Microservices paradigm. A process is composed of a series of activities which are executed in a specific order. Each activity provides a simplistic process which is not necessarily significant when used alone. However, grouped with the others, it becomes a complete and viable solution. Multiple solutions which enable communication between them are introduced in the next section.

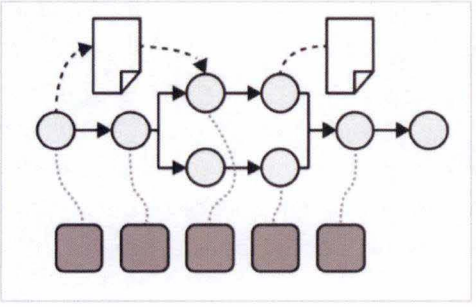


Figure 4.4 - Process-based

3.3.1.4 Orchestration versus choreography

There is a huge debate about *Orchestration* and *Choreography* definitions [20]. The easiest way to explain both is to illustrate them (*Figure 4.5 and 4.6*). Service *orchestration* can be compared to a control tower. The pilot of the plane approaching or departing the terminal area communicates with the tower rather than explicitly with other pilots. The tower literally orchestrates the flight. One way to think about service *choreography* is to think about a dance company. All dancers move in synchronization with the others but no one is leading or directing. Dances are choreographed through the individual dancers working in conjunction with one another [23]. Both solutions could be suitable depending on the environment.

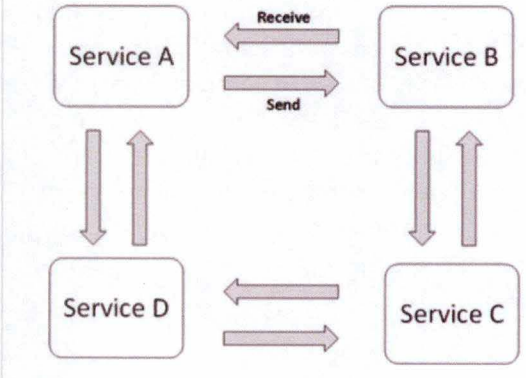


Figure 4.5 - Choreography

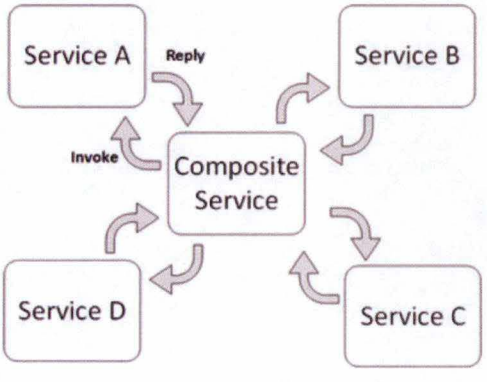


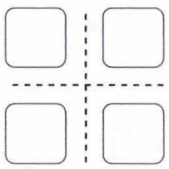
Figure 4.6 - Orchestration

3.3.2 Impacted challenge(s)

- *Loose Coupling.* Obviously, by dividing and separating logic and concern, components and services become increasingly feebly tied together.
- *Reliability* also benefits from this separation. Indeed, when properly handled, isolated components which fail do not impact the others.

- *State*, however, tends to be harder to deal with. Indeed, components and services which are part of the same business logic, often have to share states. During Online shopping for example, the service which is responsible for sending the confirmation email and the service which is responsible for debiting the account should both share the state of the basket.

3.4 Loose Coupling



The previous chapter brings about a new interrogation. **How to reduce component dependencies?** Availability and Network Latency are very important challenges. To achieve them, the main way seems to be the Scalability through the Elasticity. How could it be possible to rapidly scale if all components are tied together?

3.4.1 Solution & Discussion

Loose coupling is defined here as a pattern. However, it could also be a good practice to consider it as a goal which can essentially be achieved by using a communication canal between components. The metaphor that properly illustrates this idea is the communication between people from different origins. What makes them capable of communicating? Firstly, communication tools such as speech and hearing, secondly: the language and messages. Even if a person can speak to anybody, if these people do not share a common language, they will not be able to understand each other. It is the same for services or components in Computer Sciences.

To communicate, services have to provide endpoints. Endpoints are entries that enable other services to send them information and therefore to communicate. Endpoints are generally configurable. For example, services could decide to accept only requests which come from a specific environment, with a specific format in a specific language and so forth. In Software Oriented-Architecture, the terms REST and SOAP are often used. Both are message transmission protocols which define a series of rules which have to be respected to enable the communication. That is why services developed in REST are not able to communicate with services developed in SOAP, and vice versa. They both have endpoints to interact with but are not capable to understand all the messages they receive. It is quite similar to the previous example about people and the way they communicate.

As the reader may have understood, having loose coupled component is not straigtforward. The complexity lies in the way they interact with each other. Multiple solutions exist: webservices, queue-centric workflows, remote procedure calls (RPC) and so on.

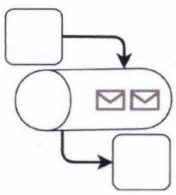
3.4.2 Impacted challenge(s)

Loose coupling was considered as a challenge in *Chapter 2*. Therefore, it directly impacts it. It also impacts almost all the other challenges:

- *Network latency* grows because the communication is no more realised through direct call inside a global system. Depending on the solution you put in place, performance could be badly impacted.
- *State* becomes more and more complex to manage as your object travels through multiple systems which are likely to be hosted in different environments.
- *Availability*. As more and more systems are added to enable communication, it adds more and more possible points of failure. Moreover, as the system becomes distributed across multiple systems (Divide and Conquer), it still adds other points of failure.
- *Consistency*. As data go through more and more services, it can be more easily degraded or lost.
- *Security*. Data transit over the Network and all services have endpoints.

Divide and Conquer with *loosely coupled* components is an important topic in distributed architectures and therefore, in Cloud architectures. Such architecture impacts everything and completely changes the way software development is classically managed.

3.5 Queue-Centric-Workflow



How to practically make loose coupled components or services communicate with each other? Which kind of intermediate system is used in common distributed architectures?

3.5.1 Solution & Discussion

3.5.1.1 Message-Oriented Middleware

The Queue-Centric-Workflow pattern uses queues which are provided by a specific software called a Messaging System or Message-Oriented Middleware *(MOM)*. To understand the QCW pattern, it is important to clarify what Messaging system does imply. A Messaging System manages messaging the way a database system manages data persistence. Just as an administrator must populate the database with the schema for an application's data, an administrator must configure the messaging system with the channels that define the paths of communication between components/services/applications. The messaging system then coordinates and manages the sending and receiving of messages. "The primary purpose of a database is to make sure each data record is safely persisted and likewise the main task of a messaging system is to tranfer messages from senders to receivers in a reliable way" [14].

The reason a Messaging System is needed to move messages from one service to another is that services and networks that connect them are inherently unreliable. Just because one service is ready to send a message does not mean that the others are ready to receive it. Even if both are ready, the network may not be working or may fail to transmit the message properly. A

messaging system offers solutions to overcome failures such as reprocessing, persistence, transactions and so forth.

Messaging system enables two kind of channels:

- Queue (*Figure 4.7*)
- Topic (*Figure 4.8*)

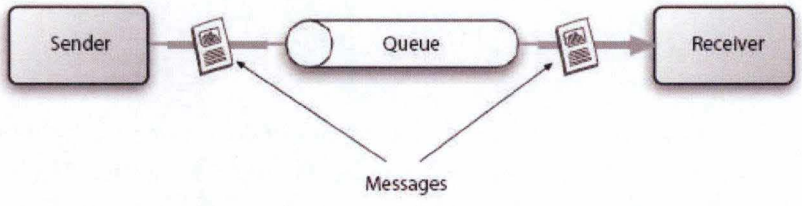


Figure 4.7 - Messaging system with a queue

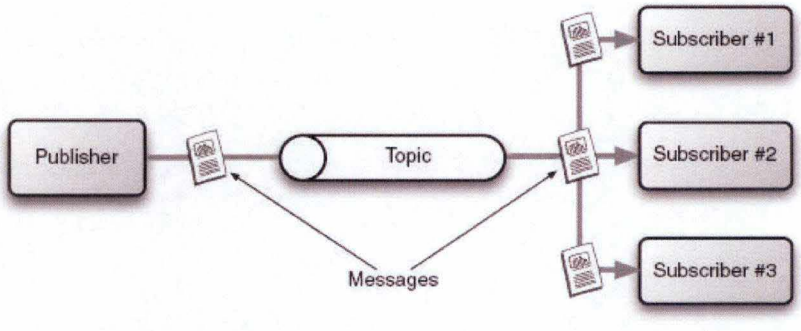


Figure 4.8 - Messaging system with a topic

The flow is quite simple: senders add messages to the queue and receivers read them from it. When a message is dequeued (otherly said consumed from the queue), it is not removed entirely from the queue. The message is hidden for a specified amount of time (the duration is specified during the dequeue operation and can be increased). We call this period the *invisibility window* (*Figure 4.9*). When a message is within its invisibility window, it is not available for dequeuing and therefore not visible for other receivers/consumers. One issue can be introduced when message processing takes more time than its window invisibility period. Indeed, the message becomes again available and is processed for the second time. There are two copies of the message which transit over the system.

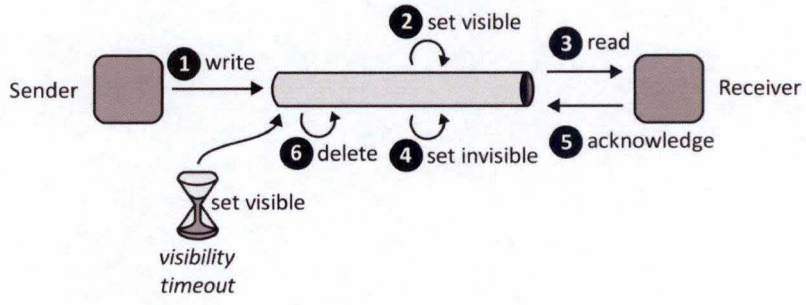


Figure 4.9 - Invisibility window

This issue introduces the notion of *idempotency*. In Computer Sciences, the term *idempotent* is used more comprehensively to describe an operation that will produce the same results if executed once or multiple times [15]. For example, according to the HTTP specification, the HTTP verbs PUT, GET and DELETE are all *indempotent* operations. Indeed, assuming their successes, all requests executed once or 100 times, will produce the same result [16]. If your system operations are not *idempotent*, you have to manage duplicated messages properly.

With all previous explanations about Middleware systems, the Queue-Centric-Workflow Pattern is straightforward as depicted in *Figure 4.10*. Another useful information is available on this picture: the possibility to have more than one sender (or producer) and more than one receiver (or consumer).

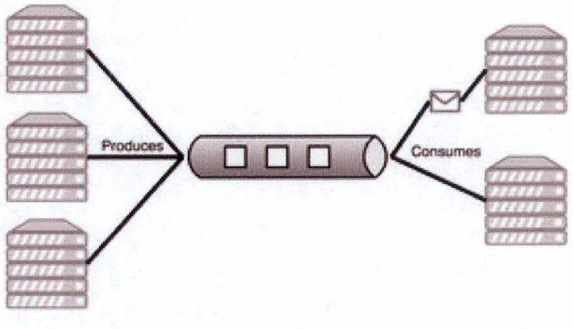


Figure 4.10 - Queue-Centric-Workflow Pattern

3.5.1.2 Synchronous versus asynchronous

Another benefit from using a Messaging system is that it enables *asynchronous* communications. As shown in *Figure 4.11*, with *synchronous* communication, a call is made to a remote component, which blocks until the operation completes. In the opposite, with *asynchronous* communication, the caller does not wait for the operation to complete before returning, and may not even care whether or not the operation completes at all.

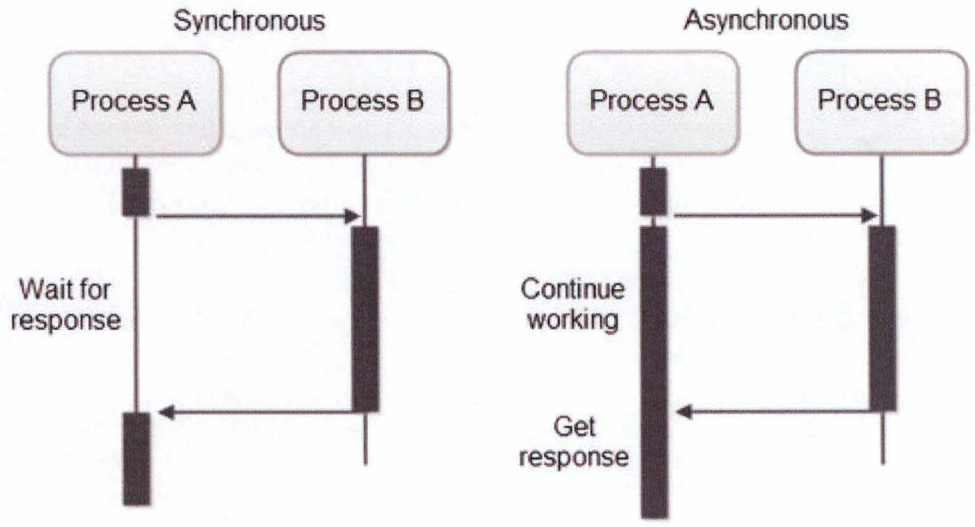


Figure 4.11 - Asynchronous versus Synchronous

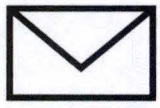
Synchronous communication can be easier to deal with: we know when processes have successfully completed or not. *Asynchronous* communication can be very useful for long-running jobs, where keeping a connection open for a long period of time between the client and server is impractical. It also works very well when you need low latency, where blocking a call while waiting for the result can slow things down and degrade the user experience. "Due to the nature of mobile networks and devices, firing off requests and assuming things have worked (unless told otherwise) can ensure that the UI remains responsive even if the network is highly laggy. On the flipside, the technology to handle *asynchronous* communication can be a bit more involved" [19].

These two different modes of communication can enable two different idiomatic styles of collaboration: respectively *request/response* and *event-based*. Each of these idiomatic can fit effectively, there is no best solution. One important factor to consider is how well these styles are suited for solving your problems considering their complexity and the real plus-value.

3.5.2 Impacted challenge(s)

- *Availability*. Multiple services could be added as senders or receivers. It therefore provides a nice mechanism to scale and enhance the availability.
- *Network Latency*. Unlike what we said in the previous point, *Network Latency* could also benefit from QCW pattern because of its asynchronous ability.
- *Reliability* and *Consistency*. Queue system isolates sub-components from each other so the failure of one component does not impact the others. Messaging System also provides mechanics to ensure that messages are not lost [74].

3.6 Idempotent Receiver



Messaging systems and idempotency are now familiar but how is it possible to simply deal with duplicates? **How to efficiently deal with duplicate messages in distributed architectures?**

3.6.1 Solution & Discussion

There are two ways to deal with duplicate messages:

1. Ignoring them,
2. Defining the logic to support them.

In order to ignore duplicates, receivers (or consumers) have to keep track of the previously received messages. Many Messaging systems automatically assign unique identifiers to each message without the application having to worry about them. When a message with an already consumed message identifier is received, the system ignores it as depicted in *Figure 4.12*.

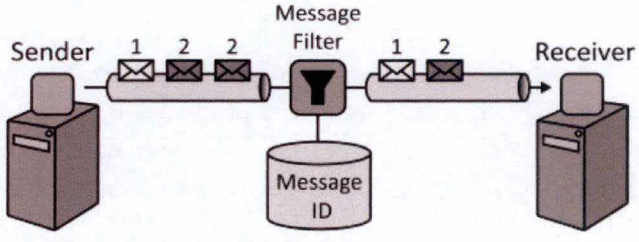


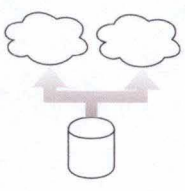
Figure 4.12 - Idempotent Receiver explained

Defining the logic to support idempotency mainly depends on your application. This case is not explained in this thesis because this is out of the architectural scope.

3.6.2 Impacted challenge(s)

Reliability.

3.7 Cdn



How is it possible to reduce Network Latency for commonly accessed files through globally distributed users? The goal is to speed up delivery of content. Content is anything that can be stored in a file such as images, videos or documents. If such content is located too far from the user accessing it, the network latency may increase.

Moreoever, storing content in only one centralized location could become a problem if the data center crashes.

3.7.1 Solution & Discussion

CDN or *Content Distribution Network* seems to be quite generic and the concept is used in each studied providers. The *CDN* is a service that functions as a globally distributed cache. The CDN keeps copies of application files in many different locations across the world (*Figure 4.13*). When a user needs a file, retrieving it from the closest location will be faster than retrieving it from the origin. "A *CDN* is really useful when files are accessed multiple times. Files that are intended to be rarely accessed, or that are for only a single user, are usually not good candidates for a CDN" [17]. Otherly said, CDN refers to content replication across multiple physical locations. The pattern is also called "Cache Aside Pattern" or "Static Content Hosting Pattern".

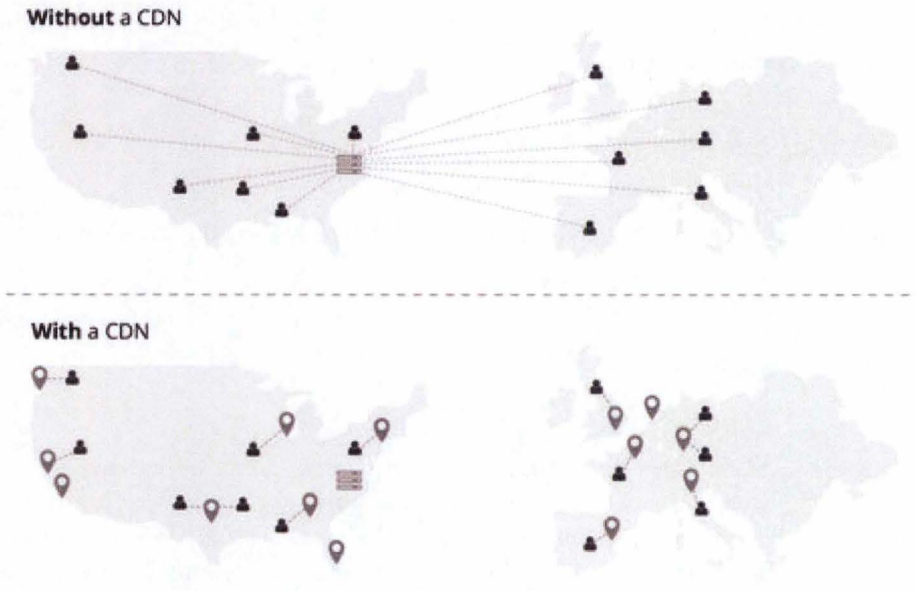
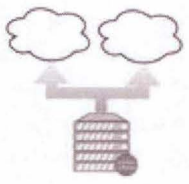


Figure 4.13 - CDN illustrated

3.7.2 Impacted challenge(s)

- *Network Latency* directly benefits from this pattern. The distance between users and files is reduced and therefore the transfer time is positively impacted.
- *Availability* is enhanced due to multiple locations where the data is stored. Indeed, when a node fails, as there are multiple replications across the network, it remains indirectly available.
- However, due the replication mechanism, *Consistency* could be harder to manage.

3.8 Multisite deployment



How is it possible to reduce network latency for the applications and improve user experience? As explained in the introduction, due to its broad network access, the Cloud is used from unpredictable locations, globally distributed across many geographical locations. Will Chinese users experiment the same user experience as Belgian ones if the application is located in France?

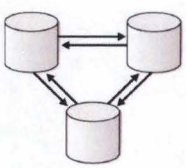
3.8.1 Solution and Discussion

Multisite Deployment Pattern is similar in some ways to CDN, in that it strives to bring applications closer to the users. *Multisite Deployment Pattern* focuses on deploying a single application to more than one data center.

3.8.2 Impacted challenge(s)

This pattern is quite similar to the previous one and therefore offers the same advantages and disavantages: *Network Latency*, *Availability* and *Consistency*.

3.9 Database replication



Static files and applications are now demultiplied across multiple locations but databases are not. Applying previous patterns just moves the problem to another point: the database. It will definitely not support the charge and become the single point of failure. **Is there a way to scale out the database?**

3.9.1 Solution and Discussion

Indeed, as it is the case for applications and static files, databases can be replicated into multiple locations. Replication is the process of copying data, but the problems associated with it are these of managing and maintaining multiple copies of the same information: the *consistency*.

The most simple replication concept is master-slave(s) (*Figure 4.14*) as it solves a lot of common problems which can be encountered with one single instance. Replication is not a pattern which is only applicable in Cloud or distributed architectures because it is broadly used to manage backup and loads in old on-premise, single node applications.

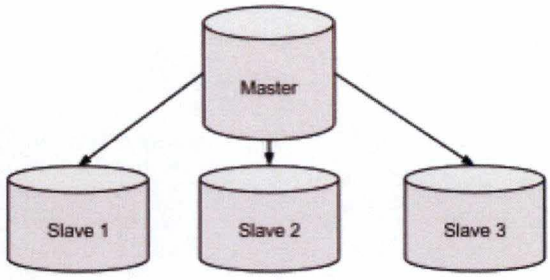


Figure 4.14 - Master-Slaves Replication

Write operations are performed on the master and read operations can be performed on every replicas (master and slaves). The difficulty results in the data replication from master to slave(s). Generally, these operations are performed through asynchronous updates using Messaging Sytems. However, this behaviour can be overriden by any particular need or preference. *Figure 4.15* from Cloud Computing Patterns [10] shows an example of replications using a Messaging system. This example contains information which are not discussed in this thesis such as the message obfuscation to hide some information or the message enrichment. This example aims to give the reader an idea of the complexity which is hidden behind the notions of *Consistency* and *Replication*.

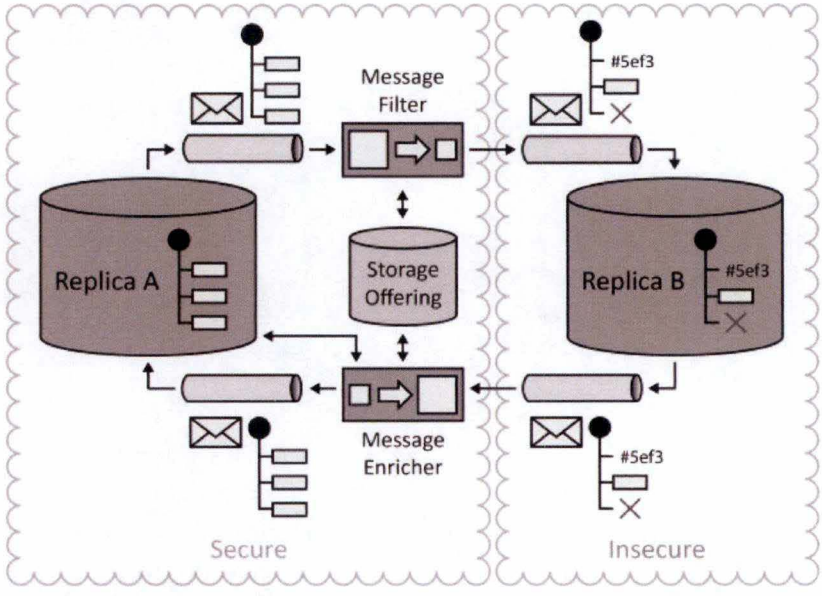
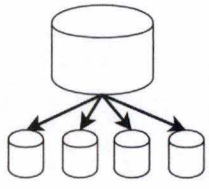


Figure 4.15 - Replication complexity with enrichment and obfuscation

3.9.2 Impacted challenge(s)

- *Availability* and *Network Latency* are positively impacted by replication due to the same condition than CDN and Multisite deployment patterns.
- *Consistency* and *Reliability* become quite tricky to manage because of the data replication.

3.10 Database Sharding



Having multiple database sounds efficient. However, **how could it be possible to store big data?** Indeed, such data are too big to be contained in only one database. Performances would be unacceptable and the system would become unstable.

3.10.1 Solution and Discussion

Sharding a database is starting with a single database and then split its data up across two or more databases called shards. Each shard shares the same database schema. The data is distributed so that each row appears in exactly one shard. The combined data from all shards represent the entire data and is the same as the original database data (*Figure 4.16*) [65].

There are multiple reasons to shard. For example, you can decide to shard because data do not fit into a single node instance. Therefore you have to divide the data into similarly sized shards to make it fit. You could also shard for performance reasons. Divide the data across shard nodes in such a way that all nodes experience the same volume of database queries and updates.

It is also important to know that all tables are not necessarily shared but rather replicated into each shard. It is for example the case with reference data which is mostly read and therefore replicated [54].

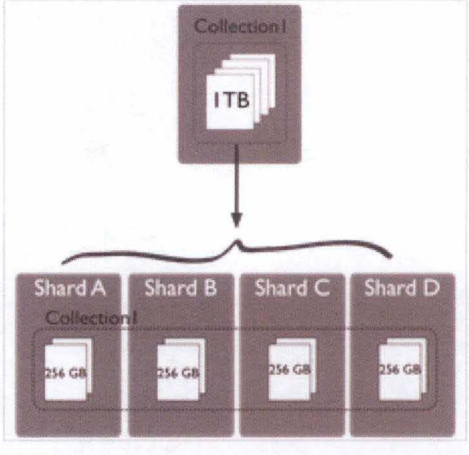


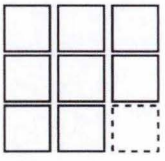
Figure 4.16 - Database Sharding

Historically, sharding has not been so popular because it was not built in and required specific application logic. The result was a significant increase in cost and complexity. Nowadays, Cloud platforms mask that complexity. For example, Windows Azure offers a system called *Federation.* The feature helps applications to manage a collection of shards, keeping the complexity out of the application layer [53].

3.10.2 Impacted challenge(s)

- A*vailability* and performances which could benefit to *Network Latencies.*
- The complexity about *consistency* and *reliability* is hidden by the vendor platform but it could also be impacted: positively because it does not require implementation but also negatively in case of very specific needs.

3.11 Consistency



Data availability is improved but is there any way to deal efficiently with its consistency? Consistency seems to be the black sheep, so are there ways to practically deal with it without impacting the availability?

3.11.1 Solution and Discussion

3.11.1.1 ACID versus BASE

In Distributed Computer Systems, the Eric Brewer's theorem, also known as *CAP theorem,* states that it is impossible to simultaneously provide all three of the following guarantees [18]:

- Consistency
- Availability
- Partition Tolerance (correct operation, even if nodes within the application are cut off from the network and unable to communicate.)

The *CAP Theorem* posits that out of the three guarantees, applications can only pick two of them. Therefore, it principally depends on what you have to achieve. Each solution has its own benefits and drawbacks. Obviously, any horizontal scaling strategy is based on data partitioning; therefore, architects are forced to decide between *Consistency* and *Availability.* In other words, they have to decide between *eventual* and *strict consistency,* which are detailled below. Each paradigm has its own characteristics and can be linked with ACID or BASE guarantees.

ACID guarantees a series of things and is often linked to *strict consistency* [62]:

- **Atomicity**: All of the operations in the transaction will complete, or none will.
- **Consistency**: The database will be in a consistent state when the transaction begins and ends.
- **Isolation**: The transaction will behave as if it is the only operation being performed upon the database.
- **Durability**: Upon completion of the transaction, the operation will not be reversed.

BASE, however, does not guarantee the consistency and is therefore associated with the *eventual consistency* [63]:

- **Basically Available**: This constraint states that the system does guarantee the availability of the data as regards CAP Theorem; there will be a response to any request. But that response could still fail to obtain the requested data or the data may be in an inconsistent or changing state, much like waiting for a check to clear into your bank account.
- **Soft state**: The state of the system could change over time, so even during times without input there may be changes going on due to 'eventual consistency,' thus the state of the system is always 'soft.'

- **Eventual consistency**: The system will eventually become consistent once it stops receiving input. The data will propagate to everywhere it should sooner or later, but the system will continue to receive input and is not checking the consistency of every transaction before it moves onto the next one. Werner Vogel's article "Eventually Consistent – Revisited" covers this topic in much greater detail.

3.11.1.2 Eventual versus Strictly Consistency

In an *eventually consistent* database, simultaneous requests for the same data value may return different values (*Figure 4.17*). This condition is temporary, as the value becomes "*eventually*" consistent [17]. *Eventual consistency* is not a deficiency or design flaw. When used appropriately, it is a real feature.

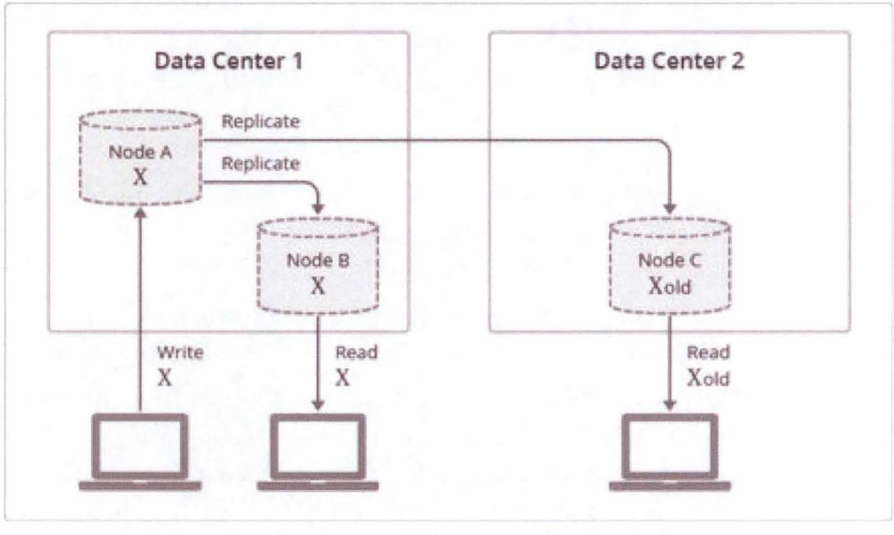


Figure 4.17 - Eventual consistency illustrated

A simple example can be found with DNS[6]. When the IP address for a domain name is changed, it usually takes hours to be propagated to all DNS servers across the Internet. This is considered as a good tradeoff. IPs change infrequently enough that we tolerate the occasionnal inconsistency in exchange for huge scalability.

The eventual consistency model has a number of variations that are important to consider:

- *Causal consistency*. If process A has communicated to process B that it has updated a data item, a subsequent access by process B will return the updated value, and a write is guaranteed to supersede the earlier write. Access by process C that has no causal relationship to process A is subject to the normal eventual consistency rules.
- *Read-your-writes consistency*. This is an important model where process A, after it has updated a data item, always accesses the updated value and will never see an older value. This is a special case of the causal consistency model.
- *Session consistency*. This is a practical version of the previous model, where a process accesses the storage system in the context of a session. As long as the session exists, the

[6] Domain Name System: DNS

system guarantees read-your-writes consistency. If the session terminates because of a certain failure scenario, a new session needs to be created and the guarantees do not overlap the sessions.

- *Monotonic read consistency*. If a process has seen a particular value for the object, any subsequent access will never return any previous values.
- *Monotonic write consistency*. In this case the system guarantees to serialize the writes by the same process. Systems that do not guarantee this level of consistency are notoriously hard to program.

The opposite of *eventual consistency* is called *strictly, strongly or guarantee consistency*. Whatever or whenever you read the data, this one is always up-to-date. Cloud Computing allows to work with such *consistency*. For example, some Cloud providers such as Windows Azure Storage offers this kind of consistency [17].

Strict consistency could be achieved by blocking the read until the write operation is finished (*Figure 4.18*). However, depending on the number of replicat, this operation could seriously impact the *network latency* and the perception of *availability* for all the users.

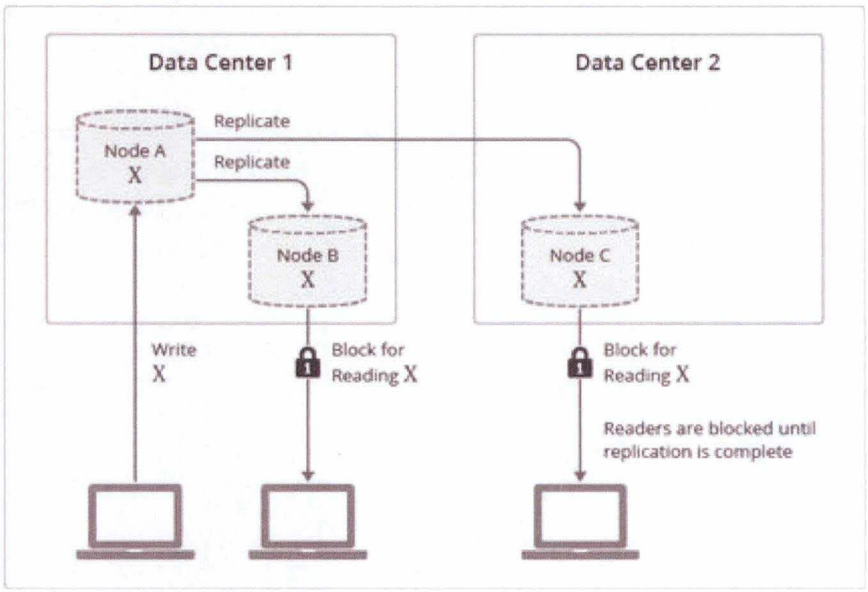
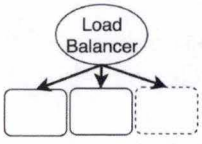


Figure 4.18 - Strict Consistency with blocking system

3.11.2 Impacted challenge(s)

Depending on the solution you have chosen, different challenges may be impacted: *Network Latency*, *Availability* and *Consistency*.

3.12 Load Balancer



Having multiple nodes in clusters introduces a new interrogation: **How is it possible to distribute requests across them?**

3.12.1 Solution and Discussion

Load Balancing consists of dividing the amount of requests that have to be processed to two or more nodes so that more work can be achieved in the same amount of time (*Figure 4.19*). Load balancer logic can be implemented in different ways:

- *Round Robin* is the simplest implementation as it distributes the load equally in a sequential manner. If the cluster consists of 3 nodes, each node will receive the same amount of requests: 1, 2, 3, 1, 2, 3 and so on.
- *Weighted Round Robin* works in the same way except that nodes are weighted. More weight means that more requests will be received.
- *Property based* is the more convenient way to deal with load balancing. Indeed, the load balancer analyses a specific property and decides whether or not the load can be sent to this node. The property could for example be the response time. When a node responses after a too long period, the load balancer considers it as not available and sends the request to another one. Other properties could be considered.

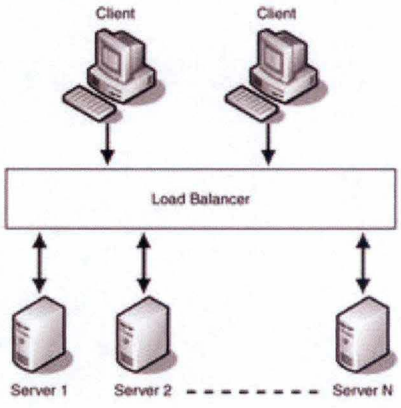


Figure 4.19 - Load balancer illustrated

Load balancer is a piece of software (or sometimes hardware) which can be redundant to minimize the risk of "single point of failure".

3.12.2 Impacted challenge(s)

- *Network Latency*. Requests are balanced to multiple instances.
- *Availability*. Requests are balanced to available instances (See next point).

3.13 Health Monitoring

Having multiple nodes and distributing the charge across all of them sounds very ingenious. **However, is it possible to ensure that all of them are available?** Indeed, load balancer will distribute requests accross all nodes but what about nodes which are down?

3.13.1 Solution and Discussion

Monitoring is a good practice and often a business requirement for web applications, databases, shared-services and so forth. Indeed, with such system, we are able to check whether everything is available and performs correctly or not. There are a lot of factors that can affect applications such as network latency, performance, storage, bandwidth. Furthermore, a service may fail partially or entirely due to any of these factors. It is therefore important to monitor them efficiently.

A Health Monitoring is usually performed in two phases:

1. **Collect**: a request is made to services which respond to it. Responses are often collected and saved into a persistence store for further analysis and statistics.
2. **Analysis**: the service responses are analysed and the result is compared to metrics to decide whether or not the service is healthy.

Several existing services and tools are available for monitoring web applications by submitting a request to a configurable set of endpoints, and evaluating the result against a set of configurable rules. It is relatively easy to create a service endpoint whose sole purpose is to perform some functional tests on the system [17].

6.13.2 Impacted challenge(s)

- *Network latency, availability and Reliability* could benefit from this pattern. Indeed, failures are now detected proactively. It is simple to identity and to correct them.

3.14 Watchdog

Node failure are now identified. However, is there a way to react to such information? How will this information help to provide a better solution? **Is it possible to use monitoring information to enable elasticity?** When a node fails, users may experiment increasing network latencies until a manual intervention is done: unpractical.

3.14.1 Solution and Discussion

Collecting and Analysing data as introduced in the previous point is called *metering*. *Metering* is the process of measuring and recording the usage of an entire application, individual parts of an application, specific services or resources [17]. Almost everything is measurable:

- The amount of storage for each user,
- The total size of data transfered,
- The number of queries for a specific service,
- The response time for a specific service,
- and so forth.

In Cloud Computing, Metrics are essential and are the basics of almost everything. As discussed, Cloud enables a *pay-per-use* model. This model is based on all the recorded metrics. Another concept mainly linked with metrics is the *Elasticity*. *Elasticity* was earlier defined as the ability to adapt the resources needed to cope with *workloads* dynamically. To cope with workloads and therefore to perform scalability, these metrics are essential: without them, it is impossible to know whether or not it is necessary to scale.

The watchdog (or Dynamic Failure Dectection And Recovery) pattern purpose is to ensure *High Availability*. It is therefore responsible for the *provisioning* according to *metrics* and *user preferences (Figure 4.20)*.

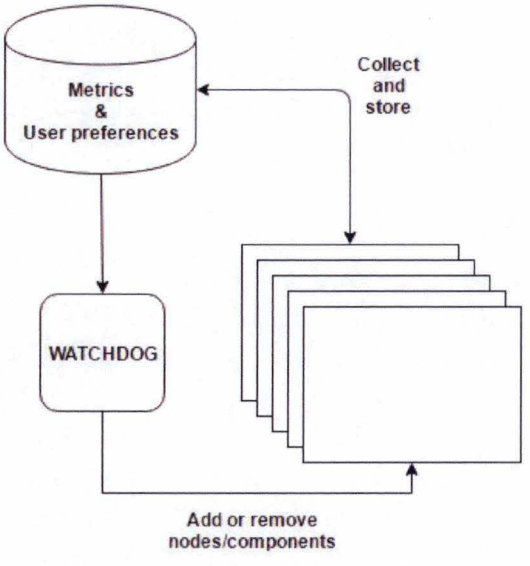
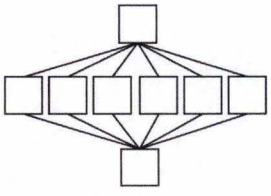


Figure 4.20 - Watchdog provisionning based on metrics

3.14.2 Impacted challenge(s)

- *Availability*, *Network Latency* and *Reliability*.

3.15 Map Reduce



Having multiple nodes and distributing the charge across all of them sounds very ingenious. **However, how is it possible to use a cluster to process one big file more efficiently?**

3.15.1 Solution and Discussion

Cloud applications often have large data to manipulate and process. *Map Reduce* is a data processing approach that enables the parallel processing of large data sets. By using this pattern, the limitation factor becomes only the size of the cluster. It is implemented as a cluster wherein each node works on a specific and smaller part of the data. This pattern can be compared to the *Divide and Conquer* one. This pattern is not typically used on small data sets but rather on what the industry refers to as *Big Data*. There is no specific rule to establish where *Big Data* starts and stops but we can reasonably consider data as "Big" when they are too big to be handle with a single machine.

Map Reduce requires two main functions: a mapper and a reducer. The large data set to be processed is firstly split up into multiple parts and then mapped to multiple nodes. Afterwards, the individual results of all the processing nodes are consolidated into a large result data set (*Figure 4.21*). This consolidation operation is called: reduce. During the reduce phase, additional functions can be added such as sum, average and so on.

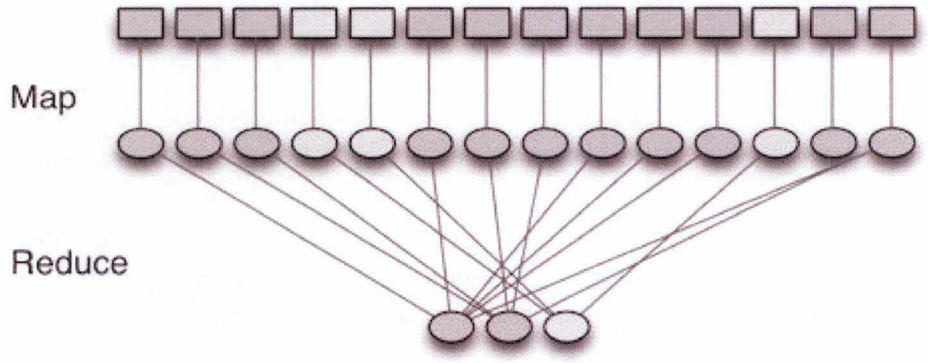


Figure 4.21 - Map-Reduce illustrated

3.15.2 Impacted challenge(s)

- *Network Latency* is one of the most impacted challenge as it is by far the purpose of this pattern. More nodes are available to parallelize the work that has to be done.
- *Consistency* is impacted as the treatment is shared by multiple nodes. Hopefully, such processes are performed by specific technologies such as Hadoop which guarantees the consistency [49].

3.16 NoSQL

The concept of Big Data is actually hype but **how is it possible to store all these data?** Classical database systems look obsolete when the data exceeds a specific amount: the performances decrease and the system becomes unstable.

3.16.1 Solution and Discussion

Before diving into the NoSQL paradigm, it is important to recall where we come from. Relational databases were born in the era of mainframes and business applications. In fact, the first commercial implementation was released by Oracle in 1979: Oracle 2 [50]. "These databases were designed to run on a single server and the only way to increase their capacities was to scale up by upgrading components" [1]. Needless to say that this behaviour is not optimal in distributed environments. Nowadays, Cloud databases present their own challenges. "They emerged as a result of the exponential growth of the Internet and the rise of distributed applications. These databases were engineered to meet a new generation of enterprise requirements: operate at any scale" [51].

NoSQL and classical RDBMS differ in their data models and in the way they distribute data among multiple nodes. RDBMS have strict data models which are hard to change and usually provoke downtime when it has to be updated on production environments. Moreover, RDBMS have difficulties dealing with semi-structured data.

NoSQL supports multiple data models:

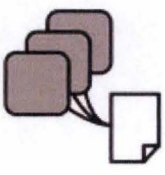
- **Key-value stores.** Similar to maps and dictionaries where data is identified by a unique key.
- **Document-oriented stores.** The evolution of the previous data model stores information as documents which are no more represented by a single value but by a series of information which are organized freely (often like in a JSON file).
- **Column family stores.**
- **Graph databases**. These systems tend to provide rich query models where simple and complex relationships can be interrogated to make direct and indirect inferences about the data in the system.

According to their internal functioning, NoSQL and RDBMS systems are very different in term of performances. NoSQL data models allow significant increase in term of persistence and research [52].

3.16.2 Impacted challenge(s)

- *Network latency.*
- *Availability* is also improved due to the built-in *sharding* system which are offered by NoSQL vendors.

3.17 External configuration Store



Dynamic Provisioning is performed by the *Watchdog. Virtual Images* are used when new resources are allocated but what about specific additional configurations? **How can configuration of the scaled applications be controlled in a coordinated way?**

3.17.1 Solution and Discussion

The majority of runtime environments includes specific information often contained in configuration files also called property files. In classical architecture, the software directly contains these files and it becomes harder to maintain because once you have to update some property values, you have to redeploy the entire application or service. Such deployment causes an unacceptable downtime and therefore impacts the availability.

Moreover, the limitation of such system is that each configuration file belongs to the service wherein it is contained. Otherly said, the property file cannot be used for the other services. For example, with Multisite Deployment pattern we discussed earlier, each of your services will contain a specific property file. This will badly impact the reliability: are each property files the same for all our instances?

The solution is to store these property files, these settings outside of our applications or services in an external storage system. Storage systems can be everything we can imagine: relational databases, file systems, key-value stores, and so forth. One thing to keep in mind is the ease to edit these configurations. As depicted in *Figure 4.22*, once these files are externalized, the information has to be transmitted to the services or the application to keep them updated.

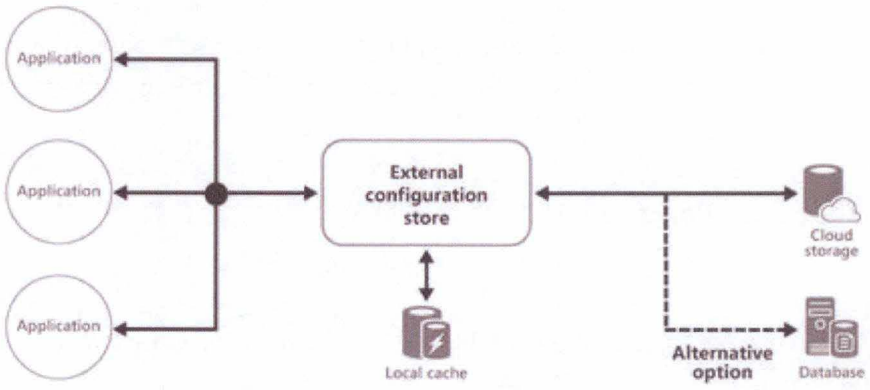


Figure 4.22 - Externalized configuration illustrated

In Computing Cloud patterns [10], they identified two ways to transmit information and therefore to refresh services with updated versions of the property files (*Figure 4.23*):

1. *Pushing*. the property files and their values are pushed to the services when a change is made. This is commonly done by asynchronous messages using Messaging Systems.
2. *Pulling*. Services or applications pull periodically the storage system to check whether any change has been made.

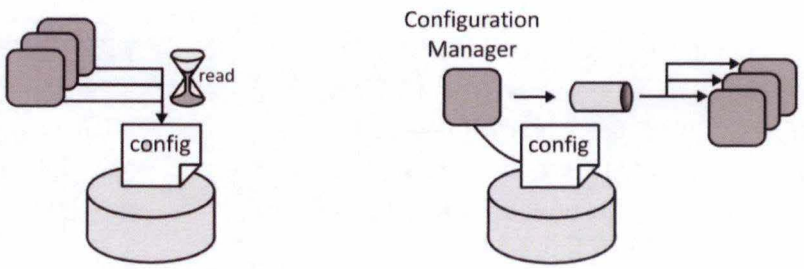


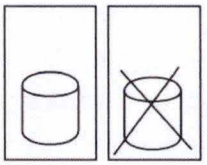
Figure 4.23 - Pushing versus Pulling information to services

Both solutions are suitable. However, having a lot of services which pull periodically is not the most optimal solution. It will generate network activities which are not needed when no changes have been performed. Therefore, pushing information to services may be a better solution.

3.17.2 Impacted challenge(s)

- *Reliability* increases because all instances share the same configurations which are placed in a unique place.
- *Resiliency* is positively impacted because we are now able to have similar copies of our services into clusters. Indeed, by sharing property configurations and images, services are now created identically.

3.18 Stateless Configuration



New services are duplicated and available in the cluster. Load balancer is now able to send request to them. What if a user is redirected to a fresh new instance? It will not contain any information about him and will consider him as not authenticated or whatever. **How is it possible to create stateless instance and keep user information available?**

3.18.1 Solution and Discussion

3.18.1.1 5.6 Stateful versus Stateless

Before diving into the explanations, it is important to understand what stands behind the word "*state*". *Session state* or *state* is a context wherein several information are maintained such as security access token, user's name, shopping cart content and so forth. Consider an application which is deployed in a cluster with 2 or more nodes. A first-time visitor adds an item to its shopping cart. Where is this cart stored? Is it stored on the node or somewhere else? The answer lies on how the session state is managed.

When the information is stored on the node, this node is defined as *stateful* as it retains the information. The benefit of *stateful* node is that it is quite simple to store and retrieve information because it is located in the "local memory". However, the Cloud native approach is to have session *state* without *stateful* nodes. A node can be kept *stateless* simply by avoiding storing user information locally but rather storing them externally. For example, in cookies or external data stores. The idea is quite similar to the previous pattern: external configuration store.

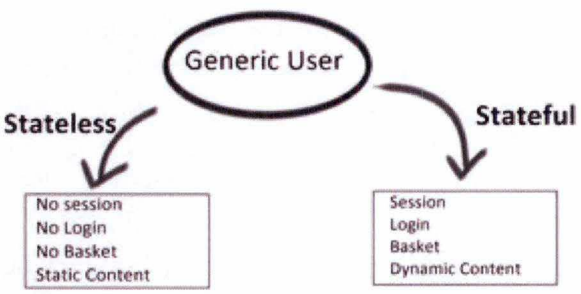


Figure 4.24 - Stateless versus Stateful

Stateful nodes are unpractical in a *cluster* environment and that is why they are not the best approach in Cloud environments. It is quite simple to understand: imagine a *Cluster* with 2 *nodes* whereon an application is deployed. A user firstly connects to the node A and logs in. The node A is *stateful* and stores all the user information. The user refreshes the page and his request is distributed to the node B. Unfortunately, node B has no information about this user (because

it is actually stored on the node A). The user is therefore considered as not logged in and his basket is empty.

The solution is similar to the previous one: states externalization. There are multiple options to realize the externalization. The first one consists of using *cookies* to store client information on his side. Cookies, web cookies, Internet cookies are small pieces of data which are stored in the user's web browser while the user is accessing services and applications. When users communicate with our services, these pieces of data are linked to the request and therefore the information is transmitted. The second solution is to externalize the state in a storage system. Users will be identified by a uniquer identifier and their states will be retrieved according to it. States could also be cached to avoid services to call the store systems and therefore improve the performances. *Figure 4.25* illustrates both behaviours.

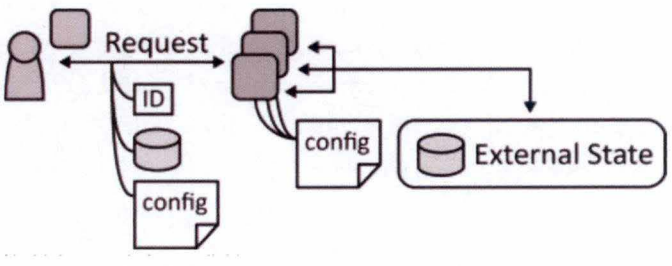


Figure 4.25 - External state storage

3.18.2 Impacted challenge(s)

- *State* is managed in a more profitable way, especially in Cloud and distributed environments where the scalability is performed and sometimes required.
- *Resiliency*. Node failure will not cause data corruption or loss. Each node of the cluster is now capable of taking charge of every requests. No more information is retained inside of them.
- *Network latency* could be impacted depending on the chosen solution. Indeed, if services have to call storage systems to retrieve information every times, the system will perform more slowly.
- *Reliability* looks easier because information are stored in more robust systems: browser, databases and so forth.

4 Conclusion

This thesis gives readers a broad overview of the complexity in distributed and more precisely Cloud environments. We started by introducing the Cloud and its benefits to raise awareness among readers about its potential and popularity. We then listed all its characteristics and highlighted the challenges they introduced. The analysis of the literature allowed us to provide the reader with a series of complementary information to deeply and comprehensively understand Cloud definitions which are broadly used in Computer Sciences books.

We also defined a pattern-oriented approach which was inspired by the Alexandrian form. However, the pattern profiles were simplified and written in a narrative approach to enable readers to focus on the problem, its solution(s) and the affected Cloud challenges.

We have seen that each step to get closer to the best architectural solution introduced new questions and new concepts. Fortunately, readers were literally guided through the Cloud complexity thanks to questions which were introduced in a logical and incremental manner.

We sincerely think that this thesis succeeds in providing a much more comprehensible approach for readers which are not familiar with Cloud or distributed environments. By giving them the basic concepts, they should now be able to properly define the Cloud, to understand its challenges and to complete their knowledge with additional readings more easily.

However, we distinguished several improvement axes which would require supplementary works and researches. Indeed, the chosen patterns in the last chapter were just an overview of all the questions the readers may ask themselves. Further reading about Enterprise applications, SOA architectures and Cloud architectures will be necessary to go in depth into implementation. The questions about Multi-tenancy and Security were not deeply covered in this thesis and should therefore be addressed in further readings. It is also important to keep in mind that having a broad overview of the Cloud architectures does not mean that we are able to implement them.

5 References

1. Couchbase, "*Why NoSQL?*", http://www.couchbase.com/nosql-resources/what-is-no-sql. [Online, Available on May, 13th 2016]
2. Mell, Peter, and Tim Grance. *The NIST definition of cloud computing*. p. 2. 2011
3. Erl, Thomas, Ricardo Puttini, and Zaigham Mahmood. *Cloud computing: concepts, technology, & architecture*. Pearson Education. Chapter 3, p. 28. 2013
4. Weinman, Joe. *Cloudonomics: The business value of cloud computing*. John Wiley & Sons. 2012
5. Reuven Cohen, "*Is Cloud Computing really cheaper?*", http://www.forbes.com/sites/reuvencohen/2012/08/03/is-cloud-computing-really-cheaper. [Online, Available on April, 25th 2016]
6. Erl, Thomas, Ricardo Puttini, and Zaigham Mahmood. *Cloud Computing: concepts, technology, & architecture*. Pearson Education. Chapter 3, p. 41. 2013
7. Dorband, J., Josephine Palencia, and Udaya Ranawake. *Commodity computing clusters at goddard space flight center*. Journal of Space Communication. p. 113-123. 2003
8. Nucleus research, "*Benchmarking availability and reliability in the Cloud: Amazon Web Services*", https://d0.awsstatic.com/analyst-reports/Benchmarking%20Availability%20and%20Reliability%20in%20the%20Cloud_Nucleus%20Research_2014%20.pdf. [Online, Available on May, 23th 2016].
9. Young, Marcus. *Implementing Cloud Design Patterns for AWS*. Packt Publishing Ltd. 2015. Kindle location 315/3258. 2015
10. Leymann, Christoph Fehling Frank, et al. *Cloud Computing patterns*. p. 50. 2014.
11. Wikipedia, "*Computer Cluster*", https://en.wikipedia.org/wiki/Computer_cluster. [Online, Available on April, 25th 2016]
12. IBM, "*IBM Cluster Systems*", http://www-03.ibm.com/systems/clusters/benefits.html. [Online, Available on April, 25th 2016]
13. Zhang, Qi, Lu Cheng, and Raouf Boutaba. *Cloud computing: state-of-the-art and research challenges*. Journal of internet services and applications 1.1. p. 7-18. 2010
14. Hohpe, Gregor, and Bobby Woolf. *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley Professional. 2004
15. Restcookbook, "*What Are Idempotent And/or Safe Methods?*", http://restcookbook.com/HTTP Methods/idempotency. [Online, Available on April, 25th 2016]
16. World Wide Web Consortium (W3C), "*HTTP Specifications*", https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html. [Online, Available on May, 23th 2016]
17. Wilder, Bill. *Cloud architecture patterns: using microsoft azure*. O'Reilly Media, Inc.. 2012
18. Wikipedia, "*CAP Theorem*", https://en.wikipedia.org/wiki/CAP_theorem. [Online, Available on April, 25th 2016]
19. Newman, Sam. *Building Microservices*. O'Reilly Media, Inc. p. 16. 2015
20. Boris Lublinsky, "*Orchestration vs. Choreography: Debate Over Definitions*", http://www.infoq.com/news/2008/09/Orchestration. [Online, Available on April, 25th 2016]
21. Mark Richards, *Microservices vs Service-Oriented Achitecture*. O'Reilly Media, Inc.. 2016.
22. Mell, Peter, and Tim Grance. *The NIST definition of cloud computing*. p. 2. 2011

23. Leymann, Christoph Fehling Frank, et al. *Cloud computing patterns*. Chapter 2.2 - Application Workloads. p. 4. 2014
24. Erl, Thomas, Ricardo Puttini, and Zaigham Mahmood. *Cloud computing: concepts, technology, & architecture*. Pearson Education. Kindle location 1570/7400. 2013
25. Amazon Web Services Inc., "*Amazon EC2 SLA*", https://aws.amazon.com/fr/ec2/sla. [Online, Available on April, 25th 2016]
26. Sean Work, "*How Loading Time Affects Your Bottom Line?*", https://blog.kissmetrics.com/loading-time. [Online, Available on April, 25th 2016]
27. Leymann, Christoph Fehling Frank, et al. *Cloud computing patterns*. p. 6. 2014.
28. Martin Fowler, "*Microservices*", http://martinfowler.com/articles/microservices.html. [Online, Available on April, 25th 2016]
29. Erl, Thomas, Robert Cope, and Amin Naserpour. *Cloud computing design patterns*. Prentice Hall Press. Kindle location 858/7529. 2015
30. Hohpe, Gregor, and Bobby Woolf. *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley Professional. 2004.
31. Alexander, Christopher, Sara Ishikawa, and Murray Silverstein. *A pattern language: towns, buildings, construction*. Vol. 2. Oxford University Press. 1977.
32. Wikipedia, "*Software Design Pattern*", https://en.wikipedia.org/wiki/Software_design_pattern. [Online, Available on April, 25th 2016]
33. Beck, Kent. Smalltalk *Best Practice Patterns. Volume 1: Coding*. Prentice Hall, Englewood Cliffs, NJ. 1997.
34. Hohpe, Gregor, and Bobby Woolf. *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley Professional. 2004.
35. Wikipedia, "*Divide and Conquer Algorithms*", https://en.wikipedia.org/wiki/Divide_and_conquer_algorithms. [Online, Available on April, 25th 2016].
36. Wikipedia, "*A Pattern Language*", https://en.wikipedia.org/wiki/A_Pattern_Language. [Online, Available on April, 25th 2016]
37. Margaret Rouse, "*What Is Loose Coupling?*", http://searchnetworking.techtarget.com/definition/loose-coupling. [Online, Available on April, 25th 2016]
38. Wikipedia, "*Commodity Computing*", https://en.wikipedia.org/wiki/Commodity_computing. [Online, Available on April 25th 2016]
39. What Is Cloud, "*Increased Scalability*", http://whatiscloud.com/goals_and_benefits/increased_scalability. [Online, Available on April, 25th 2016]
40. RabbitMQ, "*What Can RabbitMQ Do for You?*", https://www.rabbitmq.com/features.html. [Online, Available on April, 25th 2016]
41. Amazon Web Services Inc., "*Amazon SQS – Service De File D'attente De Messages*", http://aws.amazon.com/fr/sqs. [Online, Available on April, 25th 2016]
42. Wikipedia, "*Data Exchange*", https://en.wikipedia.org/wiki/Data_exchange. [Online, Available on April, 25th 2016]
43. Plummer, Daryl C., et al. *Five refining attributes of public and private cloud computing*. Gartner Research 167182.5. 2009
44. Herbst, Nikolas Roman, Samuel Kounev, and Ralf H. Reussner. *Elasticity in Cloud Computing: What It Is, and What It Is Not. ICAC*. 2013
45. Gagnaire, Maurice, et al. *Downtime statistics of current cloud solutions*. International Working Group on Cloud Computing Resiliency, Tech. Rep. 2012
46. Amazon Web Services Inc., "*Cloud Computing D'entreprise avec AWS*", https://aws.amazon.com/fr/enterprise/?nc2=h_ql_ny_livestream_blu. [Online, Available on April, 22th 2016]

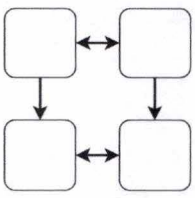
47. Microsoft Azure, "*Documentation Web Apps*", https://azure.microsoft.com/fr-fr/documentation/services/app-service/web. [Online, Available on April, 22th 2016]
48. Benoît Fleury, "*SOAP vs. REST : Choisir La Bonne Architecture Web Services*", http://blog.clever-age.com/fr/2006/10/27/soap-vs-rest-choisir-la-bonne-architecture-web-services. [Online, Available on May, 3th 2016]
49. Apache Hadoop, "*Introduction*", https://hadoop.apache.org/docs/r2.7.1/hadoop-project-dist/hadoop-common/filesystem/introduction.html. [Online, Available on May, 3th 2016]
50. Oracle, "*Oracle 2*", http://www.orafaq.com/wiki/Oracle_2. [Online, Available on May, 3th 2016]
51. MongoDB, "*NoSQL Databases Explained*", https://www.mongodb.com/nosql-explained. [Online, Available on ay, 03th 2016]
52. Sergey Sverchkov, "*Evaluating NoSQL performance: Which database is right for your data?*", https://jaxenter.com/evaluating-nosql-performance-which-database-is-right-for-your-data-107481.html. [Online, Available on May, 23th 2016]
53. George Huey, "*SQL Azure - Scaling Out with SQL Azure Federation*", https://msdn.microsoft.com/en-us/magazine/hh848258.aspx. [Online, Available on May, 23th 2016]
54. Agildata, "*Database sharding*", http://dbshards.com/database-sharding. [Online, Available on May, 23th 2016]
55. Guy Fardone, "*Cloud Elasticity and Cloud Scalability Are Not the Same Thing*", http://www.evolveip.net/cloud-elasticity-and-cloud-scalability-are-not-the-same-thing-2. [Online, Available on May, 03th 2016]
56. VM Ware, "*Virtualization Essentials*", https://www.vmware.com/files/pdf/GATED-VMW-EBOOK_VIRTUALIZATION-ESSENTIALS.pdf [Online, Availble on May, 03th 2016]
57. Amazon Web Services Inc., "*Amazon Machine Images (AMI)*", http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html. [Online, Availble on May, 03th 2016]
58. IBM, "*Hypervisors, Virtualization, and the Cloud: Dive into the VMware ESX Server Hypervisor*", http://www.ibm.com/developerworks/cloud/library/cl-hypervisorcompare-vmwareesx. [Online, Availble on May, 03th 2016]
59. Technopedia, "*What Is On-Demand Computing (ODC)?*", https://www.techopedia.com/definition/1308/on-demand-computing-odc. [Online, Available on May, 23th 2016]
60. VM Ware, "*The Benefits of Virtualization for Small and Medium Businesses*", http://www.vmware.com/files/pdf/VMware-SMB-Survey.pdf. [Online, Available on May, 23th 2016]
61. Google Cloud, "*Balancing Strong and Eventual Consistency with Google Cloud Datastore*", https://cloud.google.com/datastore/docs/articles/balancing-strong-and-eventual-consistency-with-google-cloud-datastore. [Online, Available on May, 23th 2016]
62. Technopedia, "*What Is ACID in Databases?*", https://www.techopedia.com/definition/23949/atomicity-consistency-isolation-durability-acid. [Online, Availble on May, 03th 2016]
63. ACM Queue, "*Base: An Acid Alternative*", http://queue.acm.org/detail.cfm?id=1394128. [Online, Available on May, 23th 2016]
64. Eric A. Brewer, "*Towards Robust Distributed Systems*", http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf. [Online, Available on May, 23th 2016]
65. DBShards, "*Database Sharding*", http://dbshards.com/database-sharding. [Online, Available on May, 23th 2016]

66. Margaret Rouse, "*What Is Cloud Provisioning?*", http://searchcloudprovider.techtarget.com/definition/cloud-provisioning. [Online, Available on May, 23th 2016]
67. Sreedhar Kajeepeta, "*Multi-tenancy in the Cloud: Why It Matters*", http://www.computerworld.com/article/2517005/data-center/multi-tenancy-in-the-cloud--why-it-matters.html. [Online, Available on May, 23th 2016]
68. Marketing School, "*What Is Cloud Marketing?*", http://www.marketing-schools.org/types-of-marketing/cloud-marketing.html#link1. [Online, Available on May, 23th 2016]
69. James Maguire, "*Cloud Computing Market Leaders, 2015*", http://www.webopedia.com/Blog/cloud-computing-market-leaders-2015.html. [Online, Available on May, 13th 2016]
70. IBM, "*Companies Look to Cloud to save Money, Build Business*", http://www.ibm.com/midmarket/us/en/article_cloud4_1209.html. [Online, Available on May, 13th 2016]
71. What Is Cloud, "Increased Security vulnerabilities", http://whatiscloud.com/risks_and_challenges/increased_security_vulnerabilities. [Online, Available on May, 13th 2016]
72. Wikipedia, "*Christopher Alexander*", https://en.wikipedia.org/wiki/Christopher_Alexander. [Online, Available on May, 13th 2016]
73. Technopedia, "*Why Industry Cloud Is the Next Big Thing?*", https://www.techopedia.com/2/31534/trends/cloud-computing/why-industry-cloud-is-the-next-big-thing. [Online, Available on May, 13th 2016]
74. Enterprise Integration Patterns, "*Enterprise Integration Patterns - Guaranteed Delivery*", http://www.enterpriseintegrationpatterns.com/patterns/messaging/GuaranteedMessaging.html. [Online, Available on May, 13th 2016]

6 Appendix

6.1 Divide & Conquer

How to avoid building a single monolithic application that contains everything?



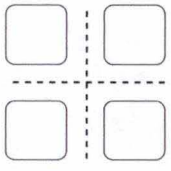
Divide and Conquer is an algorithm design paradigm based on multi-branched recursion. A divide and conquer algorithm works by recursively breaking down a problem into two or more sub-problems (divide), until these become simple enough to be solved (conquer). The solutions to the sub-problems are then combined to give a solution to the initial problem [35]

- Loose coupling
- Reliability
- State

- Layer-based Decomposition
- Pire-and-Filter-based Decomposition
- Process-based Decomposition

6.2 Loose coupling

How to reduce components/services/applications dependencies?



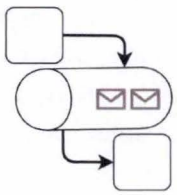
Loose coupling can essentially be achieved by using a communication canal between components. Having loose coupled component is not straigtforward. The complexity lies in way they interact with each other. Multiple solutions exist:

- Loose coupling
- Network Latency
- State
- Availability
- Consistency
- Security

- Web services
- Queues or topics
- Remote Procedure Calls (RPC)

6.3 Queue-Centric-Workflow

How to enable communication between components/services/applications?

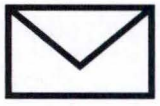


- Loose coupling
- Network Latency
- Availability
- Reliability
- Consistency

The Queue-Centric-Workflow pattern uses queues which are provided by a specific software called a Messaging System or Message-Oriented Middleware *(MOM)*. The reason a Messaging System is needed to move messages from one service to another is that services and networks that connect them are inherently unreliable. Just because one service is ready to send a message does not mean that the others are ready to receive it.

6.4 Idempotent Receiver

How to deal with duplicated messages in Messaging Systems?



- Reliability

In order to ignore duplicates, receivers (or consumers) have to keep track of the previously received messages. Many Messaging Systems automatically assign unique identifiers to each message without the application having to worry about them. When a message with an already consumed message identifier is received, the system ignores it.

6.5 Cdn

How to reduce Network Latency for commonly accessed files?

- Availability
- Network Latency
- Consistency

CDN or *Content Distribution Network* seems to be quite generic and the concept is used in each studied providers. The *CDN* is a service that functions as a globally distributed cache. The CDN keeps copies of application files in many different locations across multiple locations. When a user needs a file, retrieving it from the closest location will be faster than retrieving it from the origin.

6.6 Multisite deployment

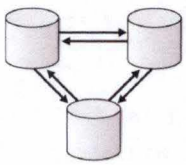
How to reduce Network Latency for applications and improve user experience?

- Availability
- Network Latency
- Consistency

Multisite Deployment Pattern is similar in some ways to CDN, in that it strives to bring applications closer to the users. *Multisite Deployment Pattern* focuses on deploying a single application to more than one data center.

6.7 Database replication

How to reduce Network Latency for databases?



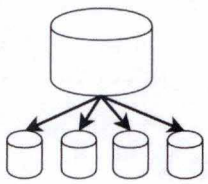
- Availability
- Network Latency
- Consistency

As it is the case for applications and static files, databases can be replicated into multiple locations. Replication is the process of copying data, but the problems associated with it are these of managing and maintaining multiple copies of the same information: *consistency*.

The most simple replication concept is master-slave(s) as it solves a lot of common problems which can be encountered with one single instance.

6.8 Database Sharding

How to deal efficiently with big data?



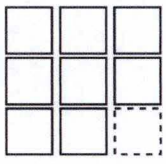
- Availability
- Network Latency
- Consistency

Sharding a database is starting with a single database and then split its data up across two or more databases called shards. Each shard shares the same database schema. The data is distributed so that each row appears in exactly one shard. The combined data from all shards represent the entire data and is the same as the original database data.

• Reliability	

6.9 Eventual and Strict Consistency

How to deal efficiently with consistency (eventual and strict)?



- Availability
- Network Latency
- Consistency

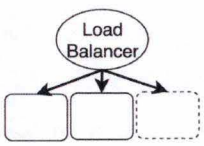
In an *eventually consistent* database, simultaneous requests for the same data value may return different values. There are multiple variations:

- Causal consistency
- Read-your-writes consistency
- Session consistency
- Monotonic read consistency
- Monotonic write consistency

The opposite of *eventual consistency* is called *strictly, strongly or guarantee consistency*. Whatever or whenever you read the data, this one is always up-to-date. It can be achieved by blocking the read until the write operation is finished.

6.10 Load balancer

How to distribute requests between all components/services/applications?



- Availability
- Network Latency

Load Balancing consists of dividing the amount of requests that have to be processed to two or more nodes so that more work can be achieved in the same amount of time. Load balancer logic can be implemented in different ways:

- Round Robin
- Weighted Round Robin
- Property based

6.11 Health Monitoring

How to ensure all components/services/applications are up and running?

- Availability
- Network Latency
- Reliablity

Monitoring is a good practice and often a business requirement for web applications, databases, shared-services and so forth. Indeed, with such system, we are able to check whether everything is available and performs correctly or not. There are a lot of factors that can affect applications such as network latency, performance, storage, bandwidth. Furthermore, a service may fail partially or entirely due to any of these factors. It is therefore important to monitor them efficiently.

6.12 Watchdog

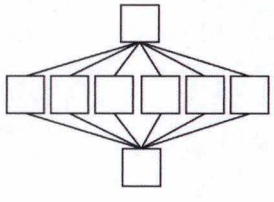
How to perform elasticity based on health monitoring information?

- Availability
- Network Latency
- Reliablity

The watchdog (or Dynamic Failure Dectection And Recovery) pattern purpose is to ensure *High Availability*. It is therefore responsible of the *provisioning* according to *metrics* and *user preferences.*

6.13 Map Reduce

How to use a cluster to perform Big data and parallelize process?



- Network Latency
- Consistency

Cloud applications often have large data to manipulate and process. *Map Reduce* is a data processing approach that enables the parallel processing of large data sets. By using this pattern, the limitation factor becomes only the size of the cluster. It is implemented as a cluster wherein each node works on a specific and smaller part of the data.

Map Reduce requires two main functions: a mapper and a reducer. The large data set to be processed is firstly split up into multiple parts and then mapped to multiple nodes. Afterwards, the individual results of all the processing nodes are consolidated into a large result data set.

6.14 NoSQL

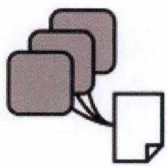
How to perform elasticity based on health monitoring information?

- Key-value stores
- Document-oriented stores
- Column family stores
- Graph databases

- Availability
- Network Latency
- Reliablity

6.15 External configuration Store

How to scale out and control the way images are created?

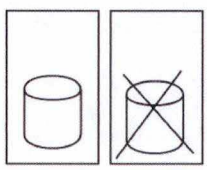


The solution is to store property files, settings outside of the applications or services in an external storage system. Storage systems can be everything: relational databases, file systems, key-value stores, and so forth. One thing to keep in mind is the ease to edit these configurations.

- Resiliency
- Reliablity

6.16 Stateless Configuration

How to work with stateless components/services/applications?



The solution is simple: states externalization. There are multiple options to realize the externalization. The first one consists of using *cookies* to store client information on his side. Cookies, web cookies, Internet cookies are small pieces of data which are stored in the user's web browser while the user is accessing services and applications. When users communicate with our services, these pieces of data are linked to the request and therefore the information is transmitted. The second solution is to externalize the state in a storage system. Users will be identified by a uniquer identifier and their states will be retrieved according to it.

- State
- Resiliency
- Network Latency
- Reliablity

	States could also be cached to avoid services to called the store systems and therefore improve the performances.