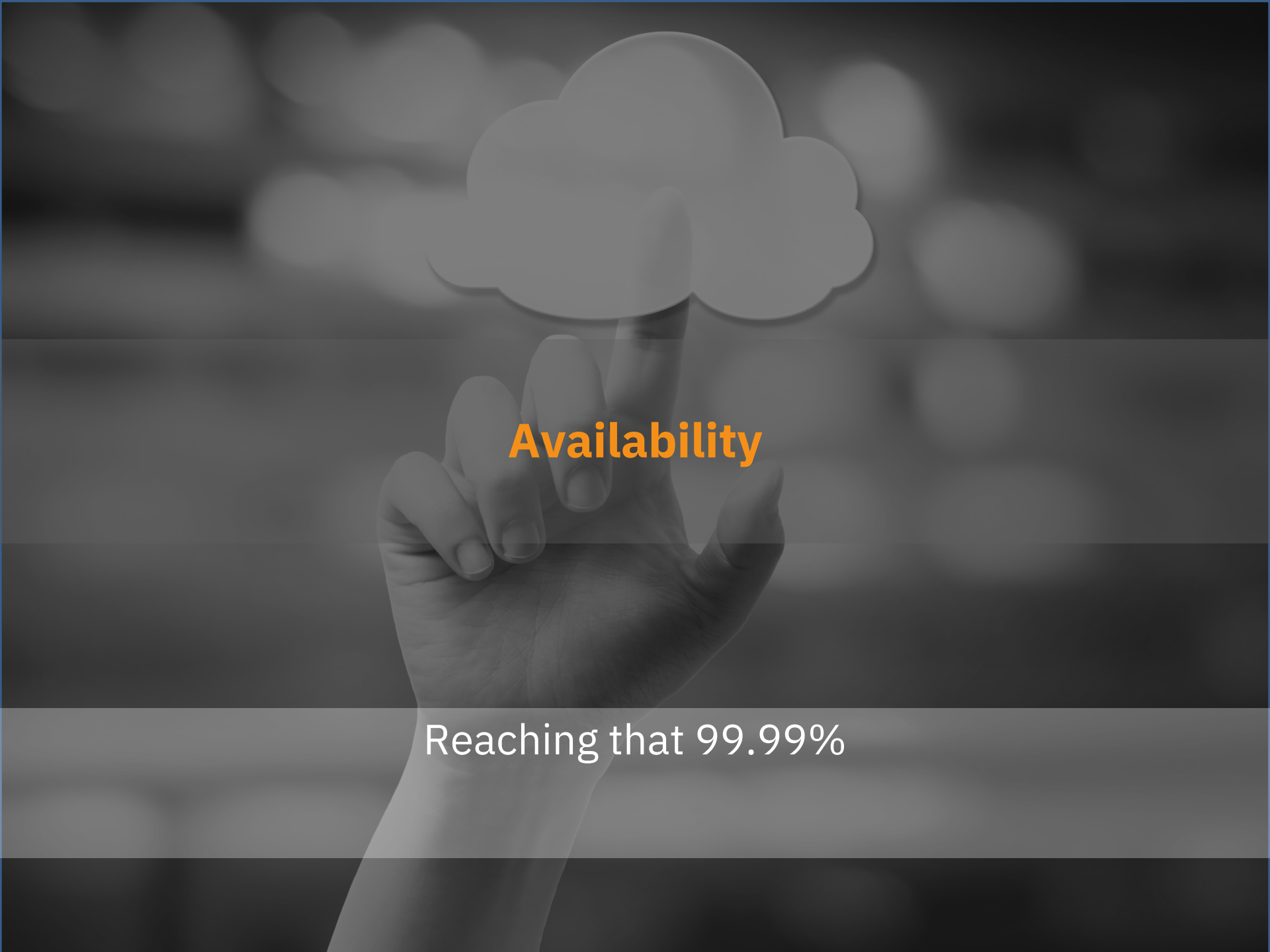
Service Oriented Architecture

Application Design – Availability

Lecture 8

Dr. Alessio Bonti


Availability
Reaching that 99.99%

Availability

CONCEPTS

What is availability?

DEFINITION

Availabilty is a measure of performance of a system, not in terms of speed or response time but in terms of service outage over time. Very often it is expressed as the proportion of time that the system is accessible and performs its function over a defined period.

Why do we care about availability?

01 AVAILABILITY HAS A DIRECT IMPACT ON USER EXPERIENCE

02 AVAILABILITY HAS IMPLICATIONS ON ARCHITECTURE AND IMPLEMENTATION

Availability

CONCEPTS

What is availability?

Is availability "a new function" of the system?

01

AVAILABILITY IS A NON-FUNCTIONAL PROPERTY

Together with scalability and security, availability is defined as a "non-functional" requirement for a system. Non functional requirements indicates properties – or better, qualities – of the system that indicate how a the system achieves something, rather than what they do.

Functional requirements on the other side are specifications of the functions of a system and they explain what the system is required to do.

Some examples:

- *"the system must be up 99.999% of the time, in a year"* ⬅ **AVAILABILITY, NON FUNCTIONAL REQUIREMENT**
- *"the system allows you create an event"* ⬅ **FUNCTIONAL REQUIREMENT**
- *"as a user I can post comments to an event"* ⬅ **FUNCTIONAL REQUIREMENT**
- *"the system must encrypt passwords"* ⬅ **SECURITY, NON FUNCTIONAL REQUIREMENT**

Availability

CONCEPTS

What is availability?

How availability is measured?

At a very high-level availability is measured as a proportion between the time that the system was accessible and operational and the total time of a specified interval.

02

$$\text{AVAILABILITY} = \frac{uptime(T_{PERIOD})}{total(T_{PERIOD})} = \frac{uptime(T_{PERIOD})}{uptime(T_{PERIOD}) + downtime(T_{PERIOD})}$$

There exist more refined characterisation of this quantity but this is the one generally used in SLAs, with a reference time of a year.

Availability

CONCEPTS

Standards of availability

In relation to availability, the pre-cloud computing era used the "5 9s" standard, as an expected target of availability for production systems:

03

"FIVE 9s" AVAILABILITY = 99.999%

As mentioned, availability is often related to a period of a year and a "5 9s" availability implies a total period of 5.256 minutes of downtime over a year.

04

AVAILABILITY WARRANTIES IN THE CLOUD

"5 9s" have been considered an enterprise standard in the pre-cloud era. With the advent of the cloud and the use of hardware more prone to failures, the warranties on availability that the vendors are willing to endorse have dropped and it is usual to find 2, 3, to 4 9s.

Availability

CONCEPTS

Standards of availability

Wait a minute… did you say less availability? … in the cloud?

How come a system that provides infinite scalability has become less reliable?

05

AVALABILITY, SCALABILITY, and ECONOMIES OF SCALE

As noticed before, cloud computing gives the illusion of infinite scalability. To deliver such experience, data centers utlise large arrays of hardware. Datacenters can be built either out mid-high end hardware or standard hardware.

By leveraging standard hardware, more capacity can be built at the expense of performance and sometime availability, for the single node. The resulting system can be overall result less reliable.

Availability

FOCUS POINT

Engineering availability

Despite the underlying infrastructure might be delivered with a lower target in availabilty, it is still possible to achieve high levels of availability for cloud applications.

ENGINEERING APPLICATION AVAILABILITY

As happens to scalability, availability needs to be engineered into the application architecture and behaviour of the system, and cannot only be achieved by purchasing more reliable hardware.

This is to best take advantage of some of the key advantages of cloud computing. In particular, the indefinite availability of resources.

Let's see what this entails.

Availability

AVAILABILITY DESIGN

Engineering availability

There are essentially two causes that can impact on application availability:

01

FAILURES

This is the primary cause. Failure can occur at different levels: network connections and equipments, computing nodes, storage. The end result is that either the entire or part of the application cannot perform its function. In the worst case the application is not accessible.

02

PERFORMANCE DEGRADATION

If the application (and/or the supporting infrastructure) is under considerable stress caused by excessive load or demand, requests can be dropped thus making the application not available for some users. As happen for failures performance degradation can either interest a set of components or the entire application.

Availability

AVAILABILITY DESIGN

Engineering availability

Availability issues caused by performance degradation can be addressed by using a scalable architecture and design as already discussed.

What can we do for addressing failures and therefore increase application availability?

UNDERSTANDING FAILURES

In order to engineer availability in the design of applications, we need to understand the failure model first. In particular, which type of failures our application is subject to, how frequent these failures are, and which types of failure we can control and which ones we cannot.

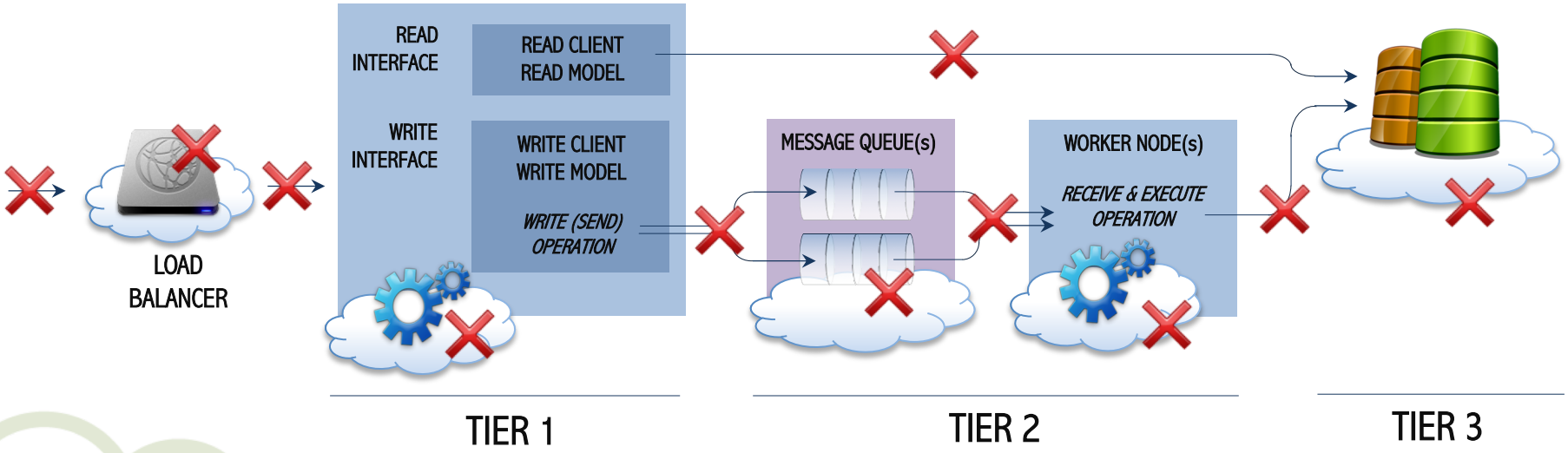
This analysis will inform us on what counter measures are appropriate to take to ensure application availability.

Availability

AVAILABILITY DESIGN

Engineering availability

Let's look at our application… what can possibly go wrong?



The answer is … everything!

Availability

AVAILABILITY DESIGN

Engineering availability

We can classify the failures identified in the previous diagram into three major categories:

ENGINEER FOR AVAILABILITY

DECLARE "DISASTER"

01 FAILURES UNDER OUR CONTROL

These failures primarily relate to the component that we own and develop in the application. These are "under our control" because we can fix them.

02 FAILURES PARTIALLY UNDER OUR CONTROL

These are failures primarily related to services that we leverage in our application. We do not own them, but we can provision and dismiss failing instances of these services to attenuate the effect.

03 FAILURES COMPLETELY OUT OF OUR CONTROL

Failures related to the core infrastructure and services of the provider. These are more likely to affect not only us but many other tenant as well. Another example is when the Internet is down.

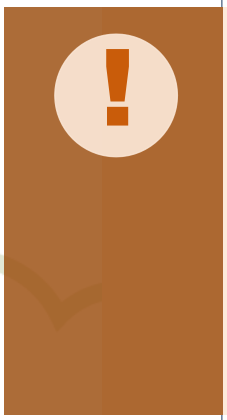
Availability

AVAILABILITY DESIGN

Engineering availability

By engineering availability into our application, we will be only able to address a portion of the failures, but we will still be at the mercy of those that we cannot cater for.

This is where the SLA we offer to our customers, and the one provided to us from the cloud vendor, play an important role.



SLAs AND AVAILABILITY

SLAs can be used not only to differentiate the service level into different classes, but they also play an important role in describing what actions the vendor will do in case of failures that cannot be avoided as they originate from extraordinary conditions.

For certain type of failures the vendor provides a "disaster recovery plan" for others it simply states it won't do anything, and this implies that the customer signing the SLA will accept the risk.

Availability

AVAILABILITY DESIGN

Addressing failures under control

Failures that are under our control primarily relate to the components of the application we own and develop, most of them can be addressed with a defensive approach to programming and robust design.

In practice, this includes:

- *proper exception management*
- *input validation*
- *type safe design and programming*
- *extensive testing (unit, functional, integration)*
- *introduction of checkpoint of verification steps*
- *appropriate logging and monitoring*

→ **ROBUST AND CORRECT CODEBASE**

→ **FAILURE PREDICTION OR EARLY DETECTION**

Availability

AVAILABILITY DESIGN

Addressing failures partially under control

Failures that are partially under control are failures that we cannot control directly, but we can either prevent or leverage vendor's services to address them.

In our case, these type of failures can be:

- *failure of the front-end nodes (underlying VM or application runtime)*
- *failure of the worker nodes (underlying VM or application runtime)*
- *failure of the queue services*
- *some connectivity failures between internal components*
- *failure of the storage services*

Most of these failures can be addressed with a redundant design.

Availability

AVAILABILITY DESIGN

Redundant design

Redundant design is a very simple strategy that enables to address quite effectively some of the failures that we cannot control.

The implementation of the "N+1" rule is a standard practice to address availabiluty concerns due to failures:

THE "N+1" RULE

In order to minimise the impact of failures on availability of a system, its consitituent component will be deployed with redundance: at least "N_C+1" instances of each component, if N_C is the number of component instances that are required to address the demand.

Availability

AVAILABILITY DESIGN

Redundant design

The N+1 deployment ensures that a system that is designed to function with N instances of a component can sustain the complete failure of 1 instance of the component without creating service degradation or interruption.

FROM N+1 TO N+M

If we want to cater for more than one complete failure we can use the N+M approach, where M identifies the number of instances that are allowed to fail.

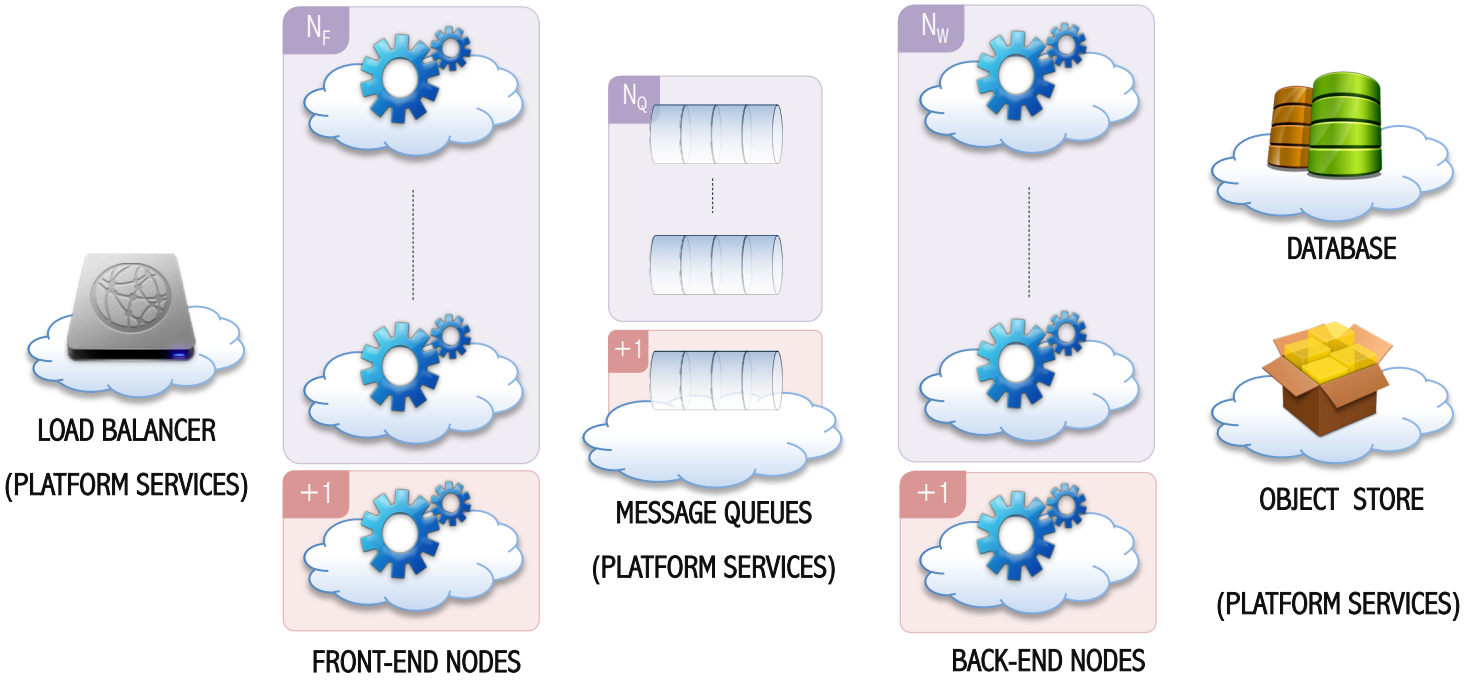
The addittional M nodes can be either in "hot-standby", which means that they do not execute work but are simply online, or can actually process requests.

Availability

AVAILABILITY DESIGN

Redundant design

How does the N+1 rule applies to our design?



Availability

AVAILABILITY DESIGN

Redundant design

Some observations:

01

FAILURE COMPENSATION

N+1 and N+M deployments compensate failures but do not eliminate them. For instance the amount of processing that was executed before the node/component failure will either need to be resetted or compensated.

02

USING MONITORING TO TAKE PREVENTIVE ACTION

In a N+1/N+M deployment, if we have means to identify a trend leading to a failure, we can more easily prevent it, by taking out the component about to fail and redirect requests to the one(s) in standby.

03

AVAILABILITY AND STATELESS DESIGN

Most the countermeasures that enable us to improve availability are ultimately made possible by the fact we're leveraging a stateless design. The same principle ensuring scalability also helps availability.

Availability

AVAILABILITY DESIGN

Storage availability

The main difference betweeen storage and the other components of the application design is that storage is a container of state and therefore it cannot simply be made "redundant".

04 STORAGE REPLICATION

Storage cannot only be made redundant, but also need to be "replicated". Replication essentially copies the information to a secondary storage. Copy can be made as soon new updates are made or at intervals. The first technique is more demanding but ensure the copies are always in synch, while the second allows for transient inconsistency of state but less demanding.

05 EVENTUALLY CONSISTENT STORAGE

To increase storage availability we replicate the storage. In case where all the instances can act as a primary storage, failures in the network connecting them, could create partitions and therefore an inconsistent view of the state because updates become out of sync.

Availability

AVAILABILITY DESIGN

Additional considerations

A couple of additional scenarios can be considered from the perspective of application availability:

01

GEOGRAPHIC DISTRIBUTION

Failures can often be concentrated in locations, in particular at datacenter level. Therefore, when introducing redundancy it is worth considering geographic distribution (i.e. a different data center).

This approach is particularly suited for storage replication and less appropriate for other components because of performance penalties and additional data transfer charges.

02

AVAILABILITY AND APPLICATION PARTITIONING

When we consider availability we should also look at the specific application partitioning strategy used to scale (e.g. by function, by group, no partitioning) because these will have different implications on application availability.

Availability

AVAILABILITY DESIGN

Designing applications more accessible

Another reason that might lead to availability issues is degrading performance.

To reduce the causes of degraded performance, we could reduce unneeded "consumption of performance", if any.

Where can we save computation cycles?

A LOOK AT WEB APPLICATION COMPOSITION

Web applications are composed of both static and dynamic components. Static components (e.g. HTML files, javascript, CSS, media files) do not require server computation to be generated but they can be served as they are, while dynamic components (e.g. APIs, data depending application fragments) are generated and therefore strictly dependent on server computation.

Availability

AVAILABILITY DESIGN

Designing applications more accessible

A way to reduce un-needed server computation is to prevent that the front-end node will serve static content and isolate this function into a different system component (or better a third party service).

CONENT DELIVERY NETWORKS

INCREASING AVAILABILITY VIA CDNs

A content delivery network (CDN) is a distributed network of servers that is designed to optimally serve content. CDN servers are composed by distribution nodes and edge servers. The formers are in charge of dispatching the content to the latters which eventually will cache it and serve to the user. Edge servers are meant to be "closer" to the user and therefore minimise the response time of the request.

In our scenario we could use a CDN to serve all the application static content, so that unnecessary load does not reach the front-end nodes. This can reduce the chances of performace degradation and therefore improve application availability.

Availability

AVAILABILITY DESIGN

Multi-tenancy considerations

Availability can be offered with different degrees, but even defining a formula could be difficult for complex systems. A better approach is to concentrate on "service features".

For instance we could have:

"GOLD" SERVICE	*The storage is replicated into a different datacenters, front-end and worker nodes and associated queues are deployed with N+M (M>1) redundancy.*	**MOST EXPENSIVE**
"SILVER" SERVICE	*The storage is replicated into a different datacenter at a specified interval, front-end and worker nodes and queues are deployed with N+1 redundancy.*	
"RUBY" SERVICE	*The storage is backed-up only, no replicas and front-end and worker nodes and queues are deployed without redundancy.*	**LEAST EXPENSIVE**

Availability

AVAILABILITY DESIGN

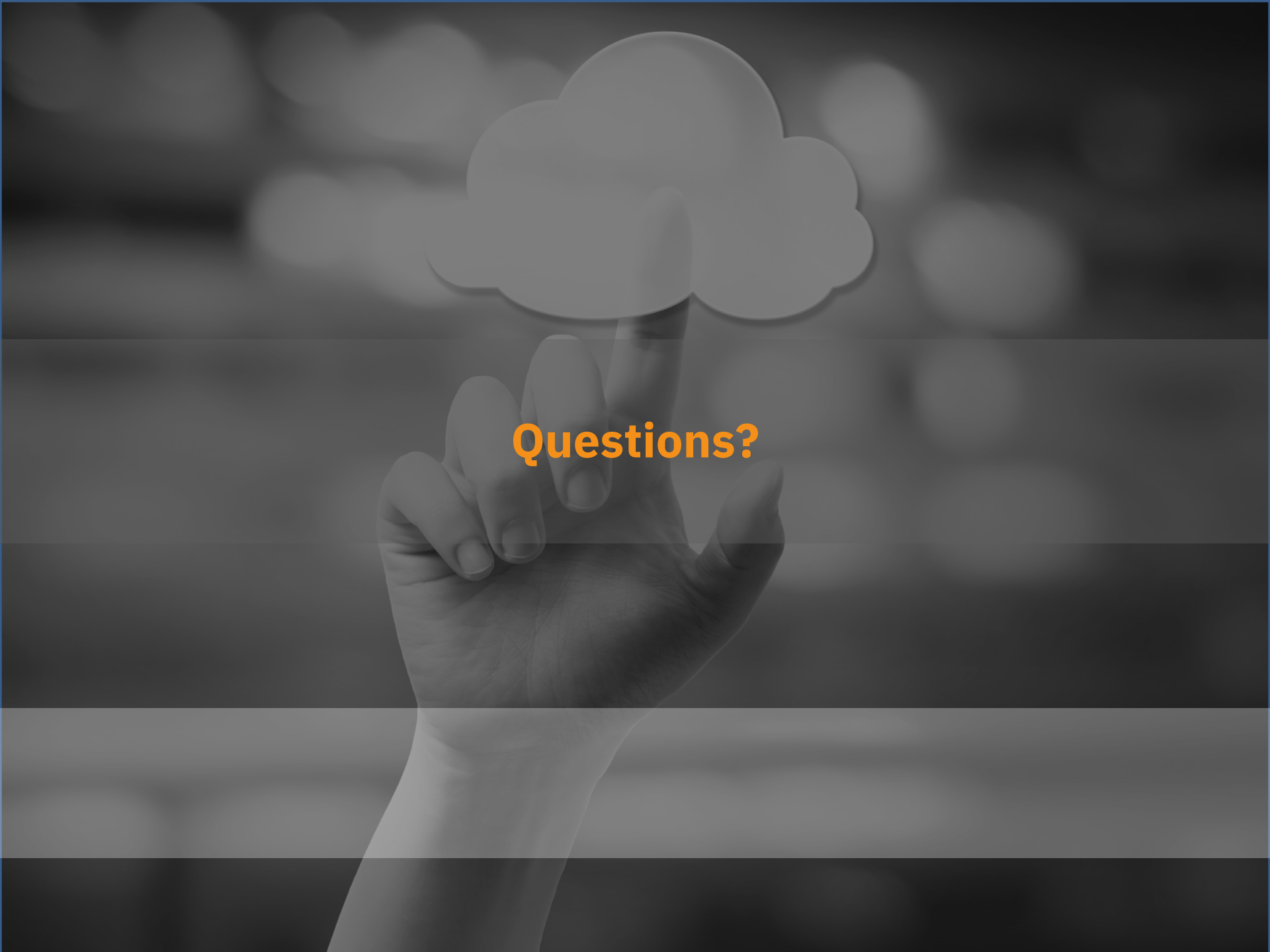
Multi-tenancy considerations

The service tiers defined previously imply that we handle separately our tenant groups and map them onto the corresponding application deployment according to agreed service level.

This enables us to implement different scaling policies per group of tenants.

FAILURES UNDER CONTROL AND MULTI-TENANCY

The differentiation among service tiers primarily applies to support, prevent, and address failures that we cannot fully control. In principle, for the failures that we can fully control, it is our intent to remove them for all tenants (i.e. code-related) unless this does imply additional and not reasonable charges.


Questions?