OPEN DATA CENTER ALLIANCE℠ BEST PRACTICES: ARCHITECTING CLOUD-AWARE APPLICATIONS REV. 1.0

TABLE OF CONTENTS

CONTRIBUTORS

Sachin Ashtikar – Intel Corporation
CJ Barker – The Walt Disney Company
Dave Casper – Moogsoft
Bette Clem – Intel Corporation
Pankaj Fichadia – National Australia Bank
Vlad Krupin – The Walt Disney Company
Karen Louie – Hewlett-Packard
Gaurav Malhotra – Hewlett-Packard
Dave Nielsen – Hewlett-Packard
Nigel Simpson – The Walt Disney Company
Catherine Spence – Intel Corporation

LEGAL NOTICE

This "**Open Data Center Alliance℠ Best Practices: Architecting Cloud-Aware Applications**" document is proprietary to the Open Data Center Alliance (the "**Alliance**") and/or its successors and assigns.

NOTICE TO USERS WHO ARE NOT OPEN DATA CENTER ALLIANCE PARTICIPANTS: Non-Alliance Participants are only granted the right to review, and make reference to or cite this document. Any such references or citations to this document must give the Alliance full attribution and must acknowledge the Alliance's copyright in this document. The proper copyright notice is as follows: "**© 2014 Open Data Center Alliance, Inc. ALL RIGHTS RESERVED.**" Such users are not permitted to revise, alter, modify, make any derivatives of, or otherwise amend this document in any way without the prior express written permission of the Alliance.

NOTICE TO USERS WHO ARE OPEN DATA CENTER ALLIANCE PARTICIPANTS: Use of this document by Alliance Participants is subject to the Alliance's bylaws and its other policies and procedures.

NOTICE TO USERS GENERALLY: Users of this document should not reference any initial or recommended methodology, metric, requirements, criteria, or other content that may be contained in this document or in any other document distributed by the Alliance ("**Initial Models**") in any way that implies the user and/or its products or services are in compliance with, or have undergone any testing or certification to demonstrate compliance with, any of these Initial Models.

The contents of this document are intended for informational purposes only. Any proposals, recommendations or other content contained in this document, including, without limitation, the scope or content of any methodology, metric, requirements, or other criteria disclosed in this document (collectively, "Criteria"), does not constitute an endorsement or recommendation by Alliance of such Criteria and does not mean that the Alliance will in the future develop any certification or compliance or testing programs to verify any future implementation or compliance with any of the Criteria.

LEGAL DISCLAIMER: THIS DOCUMENT AND THE INFORMATION CONTAINED HEREIN IS PROVIDED ON AN "**AS IS**" BASIS. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, THE ALLIANCE (ALONG WITH THE CONTRIBUTORS TO THIS DOCUMENT) HEREBY DISCLAIM ALL REPRESENTATIONS, WARRANTIES AND/OR COVENANTS, EITHER EXPRESS OR IMPLIED, STATUTORY OR AT COMMON LAW, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE, VALIDITY, AND/OR NONINFRINGEMENT. THE INFORMATION CONTAINED IN THIS DOCUMENT IS FOR INFORMATIONAL PURPOSES ONLY AND THE ALLIANCE MAKES NO REPRESENTATIONS, WARRANTIES AND/OR COVENANTS AS TO THE RESULTS THAT MAY BE OBTAINED FROM THE USE OF, OR RELIANCE ON, ANY INFORMATION SET FORTH IN THIS DOCUMENT, OR AS TO THE ACCURACY OR RELIABILITY OF SUCH INFORMATION. EXCEPT AS OTHERWISE EXPRESSLY SET FORTH HEREIN, NOTHING CONTAINED IN THIS DOCUMENT SHALL BE DEEMED AS GRANTING YOU ANY KIND OF LICENSE IN THE DOCUMENT, OR ANY OF ITS CONTENTS, EITHER EXPRESSLY OR IMPLIEDLY, OR TO ANY INTELLECTUAL PROPERTY OWNED OR CONTROLLED BY THE ALLIANCE, INCLUDING, WITHOUT LIMITATION, ANY TRADEMARKS OF THE ALLIANCE.

TRADEMARKS: OPEN CENTER DATA ALLIANCE℠, ODCA℠, and the OPEN DATA CENTER ALLIANCE logo® are trade names, trademarks, and/or service marks (collectively "**Marks**") owned by Open Data Center Alliance, Inc. and all rights are reserved therein. Unauthorized use is strictly prohibited. This document does not grant any user of this document any rights to use any of the ODCA's Marks. All other service marks, trademarks and trade names reference herein are those of their respective owners.

OPEN DATA CENTER ALLIANCE℠ BEST PRACTICES: ARCHITECTING CLOUD-AWARE APPLICATIONS REV. 1.0

INTRODUCTION

To effectively leverage the potential of cloud computing, enterprises must make fundamental shifts in the way they address development and operations. The full range of cloud-computing benefits can be better realized by adopting a fresh approach to application development that builds on the strengths of this computing model. With cloud computing, applications are built differently, run differently, and are consumed differently. These differences require new modes of thinking, consideration of the inherent trade-offs when planning solutions, and an understanding of the design patterns that achieve the best results.

This document offers a perspective for developers focused on advancing application architectures and development practices—from multi-tier architectures to building applications that can take advantage of the unique capabilities of the cloud.

To efficiently make the transition to cloud computing, developers will need to:

- Gain an understanding of the novel paradigm presented by cloud computing and the fundamental differences that apply to the cloud—both in terms of hardware and software architectures—versus the familiar, traditional computing platforms.
- Recognize and apply the fundamental software design patterns that characterize well-designed cloud-aware applications.
- Develop and refine the skills required to gain the full benefits of transformational paradigms, particularly as represented by cloud computing.

SUMMARY

To keep pace with the increasing adoption of cloud computing, developers must utilize application architectures designed with the cloud in mind, taking into consideration essential cloud characteristics. While developers can easily deploy multi-tier applications on cloud infrastructure, this physical-to-virtual (P2V) approach prevents applications from taking advantage of the unique capabilities of the cloud. Design patterns devised explicitly for cloud environments are better able to maintain high levels of reliability, enhanced security, and resiliency.

Written for developers, this paper:

- Summarizes the fundamental principles of cloud-aware application architecture
- Describes the structural principles for cloud-aware applications
- Presents some design patterns that have proven effective in cloud use cases
- Documents key operational principles for cloud-aware applications and helps enterprises to evolve applications through cloud application maturity levels

Developers, architects, CTOs, and CIOs can all benefit from the techniques for developing cloud-aware applications presented in this paper.

INTENDED AUDIENCE

This document is intended primarily for the software development community, but provides information of value for anyone involved in architecting, designing, implementing, deploying, or operating applications or services in the cloud.

Among the target audiences:

- Software developers involved in the designing, development, and deployment of cloud-aware applications
- Business decision makers evaluating the design and deployment of cloud-aware applications for the enterprise
- Enterprise IT groups tasked with planning, operations, and procurement
- Standards organizations intent on improving the interoperability of cloud platforms, APIs, and related cloud standards and guidelines

TERMINOLOGY

In this document, the term "cloud application" refers to the collection of components, hosted in the cloud, together with a client "app" that runs on a device or in a web browser. For example, a user might watch a movie on an iPad using a video player app, but this depends on authentication, authorization, media streaming, media storage, caching, and other service components that run in the cloud. Collectively these components constitute a "cloud application." This document focuses on the cloud-hosted components of the application.

THE EVOLUTION OF APPLICATION ARCHITECTURE

Application architecture has evolved concurrently with advances in computer hardware, networking, and devices from personal computers to smartphones. Cloud computing is the latest evolutionary force that is transforming application architectures.

To understand how cloud computing is changing application architecture, consider how applications are currently architected for conventional non-cloud environments.

Multi-Tier Application Architecture

The multi-tier application architecture (see Figure 1) distributes application functionality across multiple tiers. For example, in a three-tier architecture:

- The **presentation tier** provides the user interface.
- A **middle tier** handles user requests from the client and implements the application's business logic.
- A **data tier** provides data storage for the application.

An email application is an example of a three-tier architecture. This type of application consists of a presentation tier client, such as Outlook running on a PC, a middle tier messaging server (Exchange Server), and a back-end message store. The middle tier exposes interfaces in the form of an API. Clients use the API to communicate with this layer, using an application protocol, such as Internet Message Access Protocol (IMAP). The middle tier typically interfaces with multiple back-end services. In the case of an email system, the middle tier might integrate with a directory service, a message store, and a message transport agent.

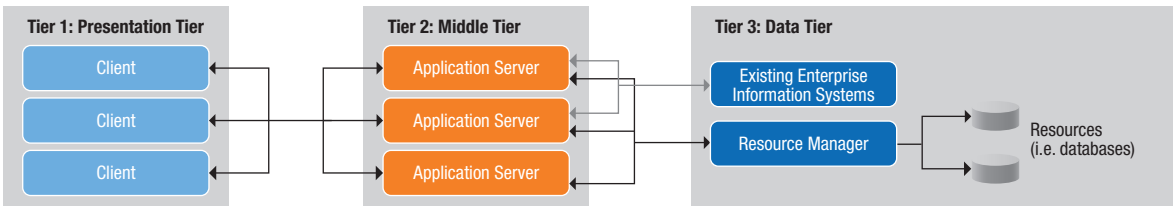


Figure 1. A three-tier architecture.

Each component of a multi-tier architecture typically runs on a dedicated server, and each tier is usually statically configured with the hostnames of the servers in the other tiers that it depends on.

This approach has many advantages over a monolithic architecture. Each of the components can operate independently and load balancing can be provided between the tiers to improve performance and reliability. Because the middle tier is accessible on the network by means of an API, multiple clients can share a single middle-tier server. Each tier can be developed independently of the others as long as the interfaces and data representations remain compatible.

In a multi-tier architecture, services are designed and developed as a set and this is reflected in how the services are deployed. The deployment is static. Dependencies between the tiers are captured in configuration files at installation time. For example, an email client knows the hostname or IP address of the mail server. The client does not find the nearest email server dynamically; it must know the hostname and be able to directly connect to that server.

Web applications are also examples of multi-tier architectures. A web server delivers application code and content to the user's web browser where it is rendered, and JavaScript code is executed to deliver an interactive user experience. A middle-tier application server executes application code in Java, Ruby on Rails, PHP, and other languages to generate dynamic content for delivery to the browser. In turn, it communicates with back-end services and databases.

Multi-Tier Architecture in a Virtualized Environment

Many businesses have migrated to virtualized infrastructures in their data centers. In a virtualized environment, application components are deployed to virtual machines rather than deployed on physical hardware. Virtualization lets available compute resources be used more effectively and improves efficiency through the use of automation—reducing provisioning times and costs.

To an application, the virtualized environment appears to be identical to a physical data center environment. The same approaches for constructing applications are often used, including static configuration. However, virtualization does enable the application to scale more efficiently, because new instances of application components can be created and configured on demand. This dynamic behavior is enabled through the use of virtualization-aware load balancers and virtual IPs (VIPs). Each tier of a multi-tier architecture can scale independently of the others, transparently.

Cloud Application Architecture

When creating applications for the cloud, developers don't necessarily need to change the way in which they architect applications. For example, a developer can easily deploy a multi-tier application on a cloud infrastructure, especially using infrastructure as a service (IaaS) where one can take application components that run on individual servers and migrate each server to its own virtual machine (VM). This P2V approach, however, prevents applications from taking advantage of the unique capabilities of the cloud. Many multi-tier applications, for example, are coupled to specific infrastructure locations by incorporating server names, IP addresses, or particular web server configurations. This approach makes it impossible to use automation to scale out to multiple VMs or burst instances among private and public clouds. To effectively leverage the full range of cloud capabilities, developers must adopt an application architecture designed with the cloud in mind, taking into consideration essential cloud characteristics, including elasticity, self-service, and multitenancy. The diagram in Figure 2 compares the traditional multi-tiered application to a cloud-aware application.

By nature, applications that use the cloud as a platform are subject to the constraints and concerns that impact distributed systems. Peter Deutsch described the Eight Fallacies of Distributed Computing[1] in 1994. These fallacies represent prevailing but incorrect assumptions that programmers often make when developing distributed applications. While these fallacies apply to any application that consists of multiple network-distributed services, they are especially relevant to cloud applications because of the characteristics of the cloud environment. Countering the misconceptions and misunderstandings at the core of the fallacies provides a useful model for creating effective design and operational principles around proven tenets and best practices associated with distributed applications. These best practices can be valuable guides when developing cloud-aware applications optimally suited for the cloud environment.

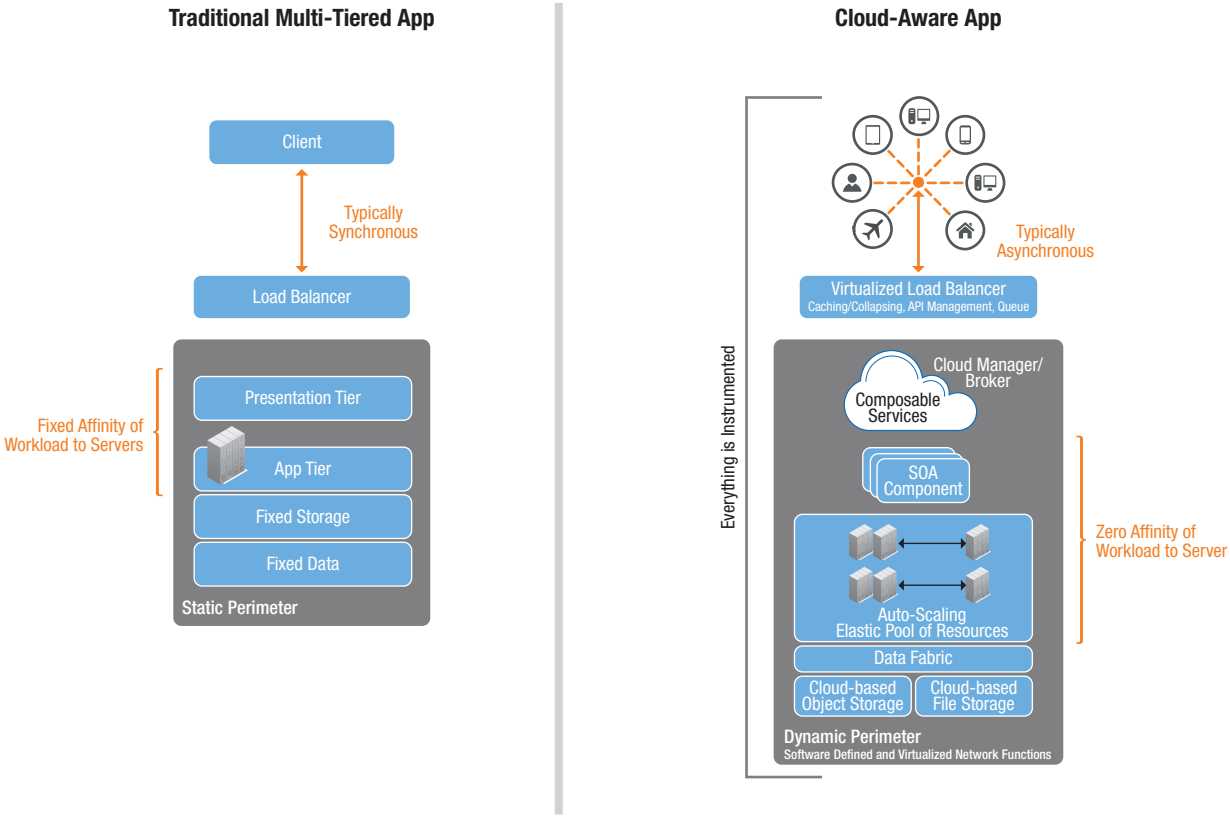


Figure 2. Traditional versus cloud-aware application architecture.

[1] Rotem-Gal-Oz, A., Fallacies of Distributed Computing Explained. www.rgoarchitects.com/Files/fallacies.pdf

PRINCIPLES OF CLOUD APPLICATION ARCHITECTURE

This section summarizes the fundamental principles of cloud-aware application architecture. Table 1 lists the structural principles that affect how the application itself is organized and coded and determines how end users interact with it. Table 3 summarizes operational principles that take the entire system into account, including how the application is deployed and operated, and how other system components interact with it.

These principles generally apply based on the context of the specific application and are likely to be phased in over time. Those items marked high priority are strongly recommended when developing cloud-aware applications. Lower priority items are less important in designing cloud-aware applications.

Table 1. Structural principles.

No.	Principle	Attribute	Priority
1	Resilient to failure	• Resiliency is designed into the application, rather than wrapped around it after the fact. Failures in cloud infrastructure are handled fluidly without interruption of service.	High
2	Resilient to latency	• Applications adapt gracefully to latency rather than timing out/failing.	High
3	Secure	• Applications are based on secure lifecycle standards and include built-in security. • Data at rest and in transit is encrypted. APIs are protected by authentication and authorization.	High
4	Location independent	• Applications discover services dynamically rather than relying on hard-coded dependencies.	High
5	Elastically scalable	• Applications respond to demand levels, growing and shrinking as required, in and among clouds.	High
6	SOA/Compose-ability	• Applications consume and expose web services with APIs discoverable at runtime. The structure incorporates small, stateless components designed to scale out.	High
7	Designed for manageability	• Applications are instrumented and expose metrics and management interfaces.	Medium
8	Infrastructure independent	• Applications make no assumptions about the underlying infrastructure, using abstractions in relation to the operating system, file system, database, and so on.	Medium
9	Defined tenancy	• Each application should have a deliberate, defined single tenancy or multitenancy model.	Medium
10	Available end-user self-service	• Users should be able to register themselves to use the app through a self-service registration interface, without entering an IT service request.	Medium
11	Bandwidth aware	• APIs and application protocols are designed to minimize bandwidth consumption.	Low
12	Cost and resource consumption aware	• Application architecture is designed to minimize costs due to bandwidth, CPU, storage consumption, and I/O requests.	Low

Structural Principles

This section details the structural principles for cloud-aware applications. These principles refer to how the application is organized and coded, and determine how end users interact with it.

Resilient to Failure

In a large-scale cloud environment, applications run on an infrastructure prone to certain types of failures that can disrupt operations. Potential vulnerabilities include hardware failure of a compute, network, or storage physical appliance, a software failure in an application component or cloud service, or a network failure due to an unresponsive component or a transient network connectivity problem. To be resilient to these types of failures, applications must be architected to handle them seamlessly and continue to operate without intervention, either at a degraded performance level or with gracefully degraded functionality.

Loose coupling minimizes dependencies between components, allowing components to be substituted without impact. For example, a decision might be made to replace a key-value store with a different implementation that has higher performance or reliability. Loose coupling also provides a way to manage failure and latency. A given component might fail, but by not statically binding to any particular instance, another instance can take over from the one that failed without the application having to take action to recover from the failure.

Developers sometimes underplay the importance of designing in resiliency for network reliability issues. These types of failure are difficult to detect and difficult to handle gracefully. For example, if critical data cannot be written to a database because of access issues, how is the operation gracefully handled? One can argue that if the network is down, the consequences are so significant that it is simpler for the application to just fail and let the user try later. From their experiences using mobile devices, users know that the network is not 100-percent reliable. Ignoring resiliency is not feasible in a large-scale distributed application where a single failure can cascade throughout the system resulting in failure of the entire service, not just an inconvenience to a single user.

In some instances, intermittent failure may be acceptable. Network features, such as application load balancing, can minimize many types of failures, enhancing resiliency and improving performance. However, in other instances failing to handle network issues gracefully can produce unacceptable user experiences. Imagine if commuters were unable to enter the subway because a network outage in some distant service prevented the turnstile from recognizing their ticket. Also, consider a game that fails to start because it could not establish communication with the high-score service (which isn't necessarily required to start the game). These kinds of unacceptable outcomes, as well as the possible cascading of network issues, make it essential that architects take a holistic view of the entire system.

"Each system has to be able to succeed, no matter what, even all on its own. We're designing each distributed system to expect and tolerate failure from other systems on which it depends.

If our recommendations system is down, we degrade the quality of our responses to our customers, but we still respond. We'll show popular titles instead of personalized picks. If our search system is intolerably slow, streaming should still work perfectly fine."

- John Ciancutti, Netflix Tech Blog: "5 Lessons We've Learned Using AWS"

Applications designed for the cloud must be resilient to infrastructure failures in many forms.

"Rule of thumb: Be a pessimist when designing architectures in the cloud; assume things will fail. In other words, always design, implement and deploy for automated recovery from failure.

In particular, assume that your hardware will fail. Assume that outages will occur. Assume that some disaster will strike your application. Assume that you will be slammed with more than the expected number of requests per second some day. Assume that with time your application software will fail too. By being a pessimist, you end up thinking about recovery strategies during design time, which helps in designing an overall system better.

If you realize that things fail over time and incorporate that thinking into your architecture, build mechanisms to handle that failure before disaster strikes to deal with a scalable infrastructure, you will end up creating a fault-tolerant architecture that is optimized for the cloud."

- Jinesh Varia, Technology Evangelist, Amazon: "Architecting for the Cloud: Best Practices"

Resilient to Latency

Cloud applications generally run in a multitenant, shared environment. Loads generated by other applications in the same environment create higher variability in response times and network latency than would be the case with dedicated data center infrastructure. Cloud applications must be designed to gracefully handle latency. Developers building resilient applications of any type generally pay close attention to latency. For applications running in the cloud, resilience to latency is essential. Infrastructure failures causing connectivity delays in the cloud can take many forms. Examples of infrastructure failures causing delays include:

- Network congestion or network partition
- Request contention in non-scalable infrastructure services
- I/O bandwidth saturation in storage systems
- Software failures in shared services
- Denial-of-service attacks that trigger service failure or resource exhaustion

Developers of mobile applications have developed techniques for addressing unreliable network connectivity, and these apply equally to the cloud. For example:

- **Queue requests.** Implement a request queue and a retry mechanism to ensure that requests are never lost. This ensures that the application state does not become corrupted. Retries may not be required in all cases. For example, it might be acceptable to allow requests to timeout if the data will be available the next time the application polls the service. It seldom makes sense to abandon write requests.
- **Handle failure gracefully.** If a failure occurs, handle it gracefully. For example, don't block the application when a service is slow to respond. Instead, return a response indicating that the request is in progress.

Latency is a factor in the performance of all network-distributed systems. The speed of light defines the lowest possible latency between two network nodes. In a best-case scenario, the latency between a service running in San Francisco and another in New York is 13.8 milliseconds, or 27.6 milliseconds for a round trip.[2] In reality, the latency is much higher, due to reduced signal propagation speeds in network links, networking protocol overhead, and delays caused by network switches and network congestion. According to AT&T, the measured latency between San Francisco and New York is 70 milliseconds.[3]

Latency varies over time, depending on network traffic volumes and routing. With applications hosted in third-party cloud infrastructure, which a developer has no control over, there is little visibility into the cloud provider's network and no visibility into how other customers are using it. Consider how much different the situation is in a managed data center where network operations staff can provide visibility into traffic and can control it. Ultimately, latency varies significantly in a cloud environment.

"AWS is built around a model of sharing resources; hardware, network, storage, etc. Co-tenancy can introduce variance in throughput at any level of the stack. You've got to either be willing to abandon any specific subtask, or manage your resources within AWS to avoid co-tenancy where you must."

- John Ciancutti, Netflix Tech Blog: "5 Lessons We've Learned Using AWS"

With distributed applications that have large numbers of nodes, latency can accrue cumulatively due to cascading service dependencies—where A depends on B, B depends on C, and so on. A "deep stack" results in slow response times that are resistant to scaling because they are constrained by network connection performance. In a data center, co-locating these dependent services can improve performance, but in a cloud environment there is no way to control this other than to specify the geographic zone in which the services should run. When architecting applications for the cloud, developers should limit the level of service nesting and ensure that time-dependent tasks are not subject to response times that result from traversing a deep stack of services. For example, a data store API could provide a fire-and-forget write method that guarantees that the commit will eventually complete. The application can issue the write and continue processing without waiting for the write to be committed to storage. Of course, the application must also be able to handle periods during which the data in storage is not consistent with the application's state. Otherwise, user confusion can result.

Reducing the number of network requests can help manage latency. This can be accomplished by designing less "chatty" protocols, where a service can request only the data it needs to function (rather than make multiple requests over time to accumulate the necessary data). For example, instead of requesting a user's ZIP code in one request, and then their phone number in another, an API could return just the contact information the application needs in a single response. In representational state transfer (REST) APIs, this is the concept of "partial response."

"…In the Netflix data centers, we have a high capacity, super fast, highly reliable network. This has afforded us the luxury of designing around chatty APIs to remote systems. AWS networking has more variable latency. We've had to be much more structured about "over the wire" interactions, even as we've transitioned to a more highly distributed architecture."

- John Ciancutti, Netflix Tech Blog: "5 Lessons We've Learned Using AWS"

This approach does have limitations, however. Consolidating data in a single response increases payload size that increases bandwidth use.

Another solution is to use caching between nodes or within each node. With caching, the number of network requests is reduced. Another advantage of caching is that if a service that the node depends on becomes unavailable due to a network issue, then the service can, if properly designed, continue to operate with cached data.

Secure

Security is needed everywhere, whether or not applications are exposed on the Internet. Increasingly, the line between enterprise and public network environments is blurring, as applications and services become more open for BYO mobile devices and software-as-a-service (SaaS) application integration. Applications should encrypt all data transfers, encrypt data at rest in cloud storage, implement secure authentication and authorization access controls, and developers should adopt secure coding practices.

In a firewall-protected data center, developers assume that the network is relatively secure and communication within the data center does not need to be encrypted. In this environment, applications need to secure communication between the user's application or web browser and the app server only in the data center's DMZ. In a cloud-hosted application, however, there is no firewall protecting communication between the application's back-end components. Although these components are not necessarily exposed to the public Internet, they are in a multitenant

[2] Distance, as the crow flies, between New York and San Francisco = 2,565 miles; speed of light = 186,000 miles/second.

[3] Measured on 8/8/2013 with http://ipnetwork.bgtmo.ip.att.net/pws/network_delay.html

network where security is not guaranteed. This reduces the degree of protection to the lowest common denominator. If the network cannot be trusted, the developer must also secure the application's communication and the services that it exposes through APIs.

Developers of applications to be deployed in private clouds should not underestimate the importance of security, because the application may eventually transition to a public or hybrid cloud. Security should be designed into every application, not added as an afterthought.

- **Secure Data in Transit.** Distributed applications depend on the network to send requests and receive responses. The integrity and security of the data that is transferred over the network must be protected. Just as it is necessary to use secure protocols when communicating sensitive information over the public Internet, developers should adopt the same approach within cloud environments. Secure network protocols, such as SSL and HTTPS, must be used to prevent network traffic snooping or man-in-the-middle attacks. Note that this includes communication with services provided by the cloud provider, as well as third-party services. Essentially, all network communication should be secured.
- **Control API Access.** Application components and services that expose APIs must also be protected. Access to an application's APIs can be managed by using an API management solution that employs OAuth to authorize and control access to APIs. With this approach the developer can constrain communication between API consumers and API end points so that only approved consumers have access.

  Unfortunately, API management is not a typical feature of most public cloud environments. Developers must therefore secure communication between components using protocols such as SSL or HTTPS and implement their own authentication approaches such as OAuth.
- **Secure Data at Rest.** If the network is not secure, data in network accessible storage is at risk. Applications should therefore ensure that sensitive data is encrypted when written to storage. Cloud providers may offer encrypted storage services that protect stored data from third-party access, but if the cloud provider owns the private key, this may offer inadequate protection. Pay particular attention to protecting personally identifiable information, ensure that credit card information is secured using Payment Card Industry Data Standard practices, and that HIPAA-compliant (Health Insurance Portability and Accountability Act) applications protect sensitive health information.

Applications can operate securely in the cloud only when network communication, access, and data are all under control.

Location Independent

Perhaps the most significant difference between a conventionally hosted application and one that runs in the cloud is that the network topology is almost guaranteed to change. For example, a cloud provider might dynamically change network routing to reduce congestion, a shared service might move to a new VM in another part of the network, or cloud bursting might result in the components of an application being spread across multiple cloud providers.

It is possible to lock down the configuration of the application with hard-coded IP addresses, but this will make the application fragile, extremely sensitive to change, and limit the ability of the application to scale elastically.

To fully leverage the capabilities of a cloud environment, applications need to be designed to operate in a dynamically changing environment where instances of services and application components come and go over time. In this type of environment, applications cannot make assumptions about which service will respond to any given request or the network location of that service. As the load increases, new instances are spawned, elastically scaling to accommodate the load. Cloud service providers provide auto-scaling capabilities that monitor the load, adding or removing instances based on rules that the developer defines. In conjunction with auto-scaling, elastic load balancing automatically handles the distribution of the load across the currently available instances and can decommission unhealthy instances.

Developers should architect their applications to support the dynamic nature of the cloud. Best practices include the following:

- There should be no hard-coded configuration information (such as static IP addresses of services). Consider using or implementing a service discovery service that application components can query to determine the location of a given service dynamically.
- Application components should be stateless, assuming nothing about the current state of the application. So if an instance is decommissioned because of auto-scaling, no loss of data occurs. Conversely, when a new instance is launched, it should not have to determine the current state.
- Applications should use established protocols such as HTTPS to avoid unnecessarily tightly coupling application components to the particulars of the network.
- Applications should be designed to utilize the auto-scaling capabilities of the cloud infrastructure provider.

Operationally, consider the following:

- Utilize API management (if available) to proxy API end points, so that API consumers are shielded from changes in the API end points.

Elastically Scalable

Elasticity is a fundamental principle of cloud computing. Elasticity enables applications to scale out and back automatically in response to changes in load. This scalability enables infrastructure to be consumed more cost effectively—more so than an environment that is over-provisioned to provide headroom for scaling up. Applications must be specifically architected to support elastic scalability. Elastic scalability does not happen as a result of simply deploying to the cloud. Simply put, the application should be structured as small, stateless components that are designed to scale out as opposed to scale up.

SOA/Compose-ability

Discoverability enables an application to dynamically locate components, rather than be statically bound to a specific instance. This capability is particularly important in the cloud where an application scales elastically—and there may be many possible instances of a component—and where failure is common. By making the application's components discoverable (and stateless), new components can spin up and start handling load without any configuration changes to the application. Avoiding the use of persistent state data can also be helpful in this regard.

Decomposing application functionality into single-purpose components enables reuse, facilitates elastic scalability, and supports Agile development. For example, rather than architecting a single message store component, break down the functionality into smaller functional components that handle storing a message, retrieving a message, searching for a message, and so on. With this approach, an improved search implementation can be deployed without impacting or redeploying the rest of the message store code.

Designed for Manageability

In a cloud environment, applications depend on infrastructure and services that the cloud service provider administers. Typically, the provider's goals are compatible with the developer's goals. The provider wants to provide reliable service and ensure high availability. However, the provider is not dedicated to supporting the developer's application. It must focus on the overall availability of shared cloud services. For example, some cloud providers have shut down services, deliberately partitioned the network to isolate problems, and inadvertently generated failures in shared infrastructure as part of their troubleshooting process. All of these actions may impact the developer's application. Unfortunately, the application owner has no control over what the cloud administrators do and may not receive the same level of detail that they would get with their own data center's technical operations team. In effect, application owners have no control over the resolution of issues, no vote on solutions, and, with some cloud providers, no escalation path.

Given that the cloud administrators are not administering or optimizing their infrastructure for the developer's application, the application owner needs to administer it themselves. This requires an understanding of what is going on with the cloud infrastructure that is outside the owner's control. One way to accomplish this is to instrument the code to provide visibility into cloud infrastructure problems, such as slow requests and service timeouts. By making sure this data available to the application owner's management and monitoring tools, they can respond proactively to failures in cloud infrastructure.

Instrumentation provides visibility into all aspects of application behavior, including performance, failures, resource consumption, and user transactions. The availability of timely, relevant data makes the application easier to maintain, support, and troubleshoot in a highly dynamic environment. This data should be logged to enable forensic analysis of problems and to provide insight into user behavior that can help improve the application and the user experience.

Where multiple applications are hosted in the cloud and are dependent on shared services owned by the same organization, it is likely that a different administrator will manage each of these services.

Infrastructure Independent

Maintaining infrastructure independence is consistently a recommended software engineering practice, but in a cloud environment using the cloud provider's proprietary interfaces results in cloud vendor lock-in. Vendor lock-in reduces portability and makes it difficult or impossible to run the application on other cloud platforms. With abstraction, it is possible to replace the implementation of an underlying service with another without rewriting application code as long as the interface contact between the application and the service is maintained.

Heterogeneity is common in a conventional data center. An application server might run on a Linux-based VM, but have a dependency on a database resident on a Windows server. Developers choose platforms based on how well they meet their development and runtime requirements, and systems engineers might have standardized system configurations for various types of services. This approach results in an environment that is heterogeneous by design and usually controlled.

Cloud environments are also heterogeneous. They provide a set of VM images that are preconfigured with specific operating systems and software. Unfortunately, the default images may not match the application's requirements. For example, a Windows VM image might not have the operating system patch level that SQL Server requires. Utilizing the available images also limits portability because there is no consistency in images across cloud providers. Subtle environmental differences between VM images that ostensibly look identical may cause application problems at runtime.

The solution to image heterogeneity is to create images customized to the application. This can be accomplished using one of these ways:

- Create a custom VM image with the precise environment that is required.
- Take a vanilla image and customize it at boot time with the required configuration.
- Create a set of application preferences that enable the best infrastructure-configuration match (by enabling brokers or administrators to identify the best-suited configuration for the app).

Runtime configuration provides an opportunity to install patches to ensure that the instance is up to date and enables the creation of a wide variety of configurations. For example, a standard base image could be created with a particular version of Linux, then configured with a security profile, updated with a particular version of Java, and subsequently configured with different Java application containers. Each configuration builds on the one that came before it. The disadvantage with runtime customization is that it takes longer for the instance to become operational, which might impact performance when dynamically auto-scaling.

With widespread adoption of TCP as an Internet standard transport protocol, cloud applications do not have to be concerned with network heterogeneity. Application layer protocols such as HTTP provide a further layer of abstraction. The popular REST architectural pattern builds on HTTP to provide a convenient way for applications to expose resources through data representations such as JSON and XML to simplify the transfer of data between components of a distributed application. A limitation of REST is the lack of standards for API design or request-and-response formats. For example, implementations may represent the API version in the resource path (for example, /api/v1/users), as a query parameter (/api/users?version=1), or in an Accept header (Accept: application/json; apiversion=1).

As a result, applications cannot assume API design or payload homogeneity across different API providers. In addition to using standard protocols such as HTTP/HTTPS, developers should establish standards for REST API design to make their APIs consistent and easy to consume. This level of standardization will become increasingly important in the future where services may be discovered and consumed based on schema definitions, instead of being hard-coded into a program.

Defined Tenancy

Unless initially designed for use as a SaaS application, most enterprise applications do not factor in tenancy as a part of the design. Nonetheless, best practices suggest that an understanding and handling of tenancy models should be an integral component of a cloud-aware application. Consider the possible impact if another group of end users wanted to use a developer's application (for example, following an acquisition) and their data had to be isolated. Also, consider table separation and data tagging (in a shared table) in this context to simplify compliance requirements (with reporting tagged to the data).

Provide End-User Self-Service

Many enterprise applications require some type of authorization, where users request access to an application and access rights are associated with a particular role. Ultimately, some provisioning occurs and a type of user account is created with profile and access information. With consumer applications, users provision themselves using self-registration tools. Following the same concept, users are starting to expect similar capability for enterprise applications.

Bandwidth Aware

It is convenient to assume that bandwidth is not an issue when designing cloud applications. At an individual request level, the payloads seem small compared to the available bandwidth. However, in large-scale applications that are typical in cloud environments, thousands of requests per second multiplied across numerous services can consume excessive bandwidth resulting in poor performance.

As mentioned earlier, cloud applications are running on a shared network infrastructure that is not under the developer's control. Although the predicted bandwidth utilization for an application may be a fraction of what is theoretically available, the actual available bandwidth may be significantly less. Lack of control over network topology may also have a negative effect on bandwidth. Unlike the situation in a conventional data center, in a cloud environment developers cannot specify that their applications' components be co-located with high-speed interconnects.

The solution to limited bandwidth is, of course, to use less of it. Techniques for accomplishing this include the following:

- Design less "chatty" protocols.
- Tailor responses to only what the application component needs.
- Reduce duplicate requests through caching.

For applications with very high bandwidth requirements, such as video streaming, it may make sense to move these activities to the edge, external to the cloud. Netflix, for example, distributes video to an external content delivery network (CDN) rather than stream from within Amazon's cloud. However, Netflix does use AWS for transcoding video masters, leveraging the elastic scalability of the cloud to parallelize the creation of multiple video encodings. Network bandwidth is less of an issue for this use case because the process is not time sensitive and delays due to bandwidth limitations do not impact users.

Cost and Resource Consumption Aware

Cloud providers have a resource consumption billing model, charging for compute, storage, data transfer, and other services. Developers should architect applications to minimize cost, for example, by minimizing network data transfer. Minimizing costs in many instances leads to the use of techniques, such as caching, that also improves performance.

Minimizing Transport Cost

There are two primary aspects to transport cost:

- The time and processing cost of marshalling and un-marshalling data transported over the network
- The financial cost of transferring the data

In a conventional data center, applications are typically not billed for network traffic. Networking is considered a fixed infrastructure cost that can be amortized across all the applications that use the network. However, in the cloud all types of resource consumption are billed, including network data transfers. Cloud service providers such as Amazon charge for data transfers with tariffs that vary depending on the source and destination of the transfer. For example, transfers into EC2 are free, but transfers out can be quite expensive. Charging exit fees is one way in which cloud providers lock customers into their platform. If an application generates a lot of data (for instance, by supporting the uploading of user-generated content), the cost of migrating to another service provider could become exorbitant. Transfers to network storage can also incur substantial costs.

When architecting cloud applications, developers should factor data transfer costs into their design strategy. Clearly, data transfers are essential to building a successful distributed application, but developers should strive to minimize traffic. For example, it makes sense for Netflix to store its encoded video assets outside of EC2 in CDNs, where storage and transfer costs are lower than the fees charges for transferring data out of EC2 to the public Internet. Data transfer costs are actually so critical for Netflix that it took the extra step of designing its own CDN hardware and operating its own CDN services[4] in the geographies where it operate.

Likewise, marshaling and unmarshaling costs should be considered. For example, choosing to compress data to reduce data transfer costs may incur additional CPU costs. The load resulting from handling compressed data can sometimes require additional VM instances, adding to the cost.

[4] Netflix Open Connect Content Delivery Network, https://signup.netflix.com/openconnect/hardware

In general, developers should create resource consumption-aware applications that seek to balance application performance against cost.

"In other words, you need instrumented applications. Metered consumption changes service invocation patterns and design efficiency, depending on the relative cost of resources. For example, are you charged more per API call, per megabit, or per CPU/hour?"[5]

- Gartner

The following points highlight approaches that developers can use to reduce data transfer costs.

- Minimize payload sizes.
  - Provide APIs that return just the data the consumer needs ("partial response").
  - Consider data compression, but balance against CPU costs for encoding and decoding.
- Minimize data transfers.
  - Cache immutable data.
  - Replace or avoid "chatty" protocols.
- Instrument code.
  - Keep track of data transfers throughout an application to identify potential for optimization.
  - Use load generation tools to generate artificial traffic that can provide insight into the impact of optimizations.

Cloud Application Design Patterns

The following sections detail design patterns that are specific to effective cloud services and existing patterns that can be applied to cloud applications.

Circuit Breaker

The circuit breaker design pattern, defined by Michael Nygard, was further developed by Netflix to use as a part of its cloud services. In this pattern, a "circuit breaker" is inserted between the component making the request and the component that handles the request. Just as in an electrical circuit, a circuit breaker can have two states: closed or open. When the circuit breaker is closed the circuit is active; power is transmitted. In the software version, requests are made to the component that normally handles the request. Open means that the normal circuit path is broken; no power is transmitted, and a back-up code path is executed. The role of a circuit breaker is to detect a fault in a circuit and trip to a safe fallback state, or open. The decision to trip a software circuit breaker is based on the following triggers:

- A request to a remote service timed out.
- The request queue for the remote service is full, indicating that the remote service is unable to handle additional requests.

These collectively factor into the error rate for the service, and when a threshold is exceeded, the circuit for the service is tripped into an open state.

Once tripped (open), the component reverts to a fallback state. There are three fallback strategies:

- **Fail transparently.** Generates a custom response by means of a fallback API method. For example, this method might return a cached or default value.
- **Fail silently.** Returns a null value for the request. This might be useful when the value is not essential to the calling service.
- **Fail fast.** Generates a 5xx response code, which is necessary when there is no sensible fallback value and the calling service required the data. This requires the calling service to handle the error in a way that makes sense to the user.

Ideally, a component would only fail transparently, but this is not always possible.

Note that the circuit breaker behavior differs from conditional response handling ("if request times out, then return 500"):

- A circuit breaker trips as a result of behavior over time, a rolling window, not a single failure.
- A circuit breaker remains tripped, and the fallback behavior remains in place until the breaker is reset.
- The state of the breaker is visible from outside the component, providing visibility to tools that can act on the patterns of failures.
- The breaker is controlled from outside the component.

[5] Gartner. "Cloud Characteristics, Principles and Design Patterns." www.gartner.com/doc/2081915/cloud-characteristics-principles-design-patterns

In the Netflix implementation the circuit breaker periodically allows requests to the handling component to pass through. If the requests succeed, the breaker is reset, and all requests are allowed through. Figure 3 illustrates the circuit breaker logic.

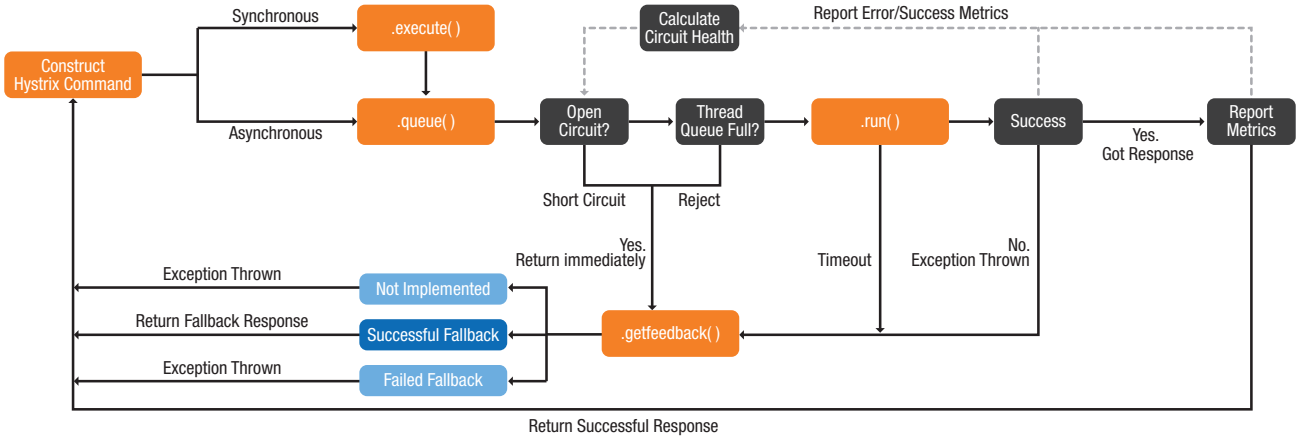


Figure 3. Circuit breaker logic that Netflix uses.

Netflix makes the state of circuit breakers visible in a dashboard that displays the same data that circuit breakers are acting on. This provides an at-a-glance view of component state, information that would not normally be visible in components that use conventional exception handling code.

Figure 4 is an annotated screenshot that shows what a circuit breaker-enabled service looks like in the dashboard.

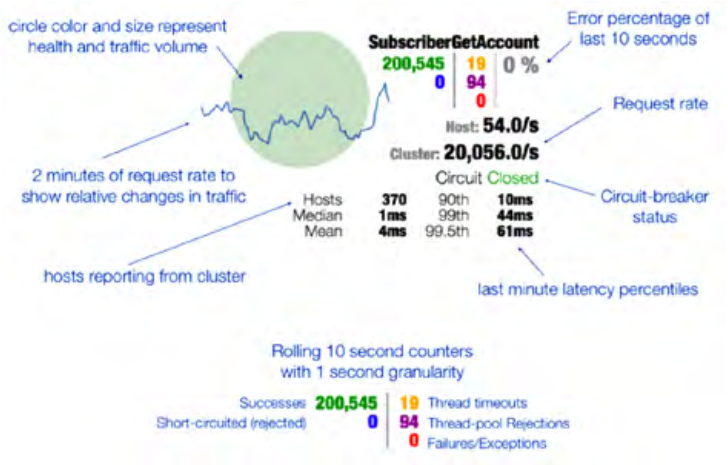


Figure 4. Example of a dashboard view of a circuit breaker-enabled service.

CIRCUIT BREAKER DESIGN PATTERN

The following points highlight the circuit breaker design pattern:[1]

- Benefits
  - Improved resilience to failure
  - To provide visibility into component failures (though visibility into tripped breakers)
- Where to use this pattern
  - To eliminate timeout delays due to network or VM instance failures
  - To prevent errors cascading and complicating upstream error handling (fail transparently, fail silently)

[1] Fault Tolerance in a High Volume, Distributed System - Ben Christensen, Netflix http://techblog.netflix.com/2012/02/fault-tolerance-in-high-volume.html

Request Queuing

The request queuing design pattern involves an application component enqueuing requests (messages or tasks) to one or more queues for processing. The requests are handled by compute nodes that process the requests when they are able. The queue acts as a buffer between the requests and processing service(s) preventing a heavy load from impacting the service causing it to fail or the request to time out. This pattern is a variant of the producer-consumer and queue-based load leveling patterns.

Rather than issue all incoming requests to a single end point for consumption, the load is staged in a queue for eventual consumption and distribution across an entire compute cluster. If a compute node fails, other workers will continue to dequeue items and handle processing requests. This provides a level of fault tolerance for ensuring high availability of access to the cloud-aware application.

The queue(s) can also serve as a mechanism to throttle or block incoming requests while maintaining control of the consumption of resources used by the service. Multiple incoming requests from a given client can be denied if they exceed a given number per second. Performance counters can be applied to incoming requests before they are enqueued. If a client's requests exceed the cloud application's configuration threshold, they are blocked from processing. When blocked, the client is sent an HTTP 429 error (Too Many Requests) and Retry-After response-header indicating how long the service is expected to be unavailable to the requesting client. Figure 5, from the Microsoft Developer Network, illustrates how a queue can effectively level the load on a service.

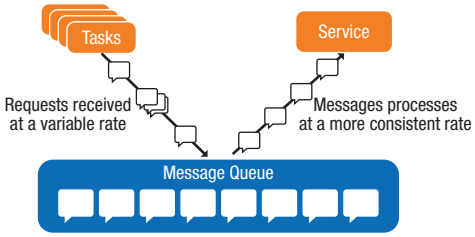


Figure 5. Using a queue to level the load on a service.

REQUEST QUEUING DESIGN PATTERN

The following points highlight the request queuing design pattern:[1]

- Benefits
  - Improved fault tolerance to ensure high availability
  - Improved performance for highly requested APIs
- Where to use this pattern
  - To manage application failover (fault tolerance)
  - To reduce end point load on a heavily requested API(s)
  - To apply alternate strategies to auto-scaling via throttling

[1] Advanced Message Queuing Protocol (AMQP) model explained www.rabbitmq.com/tutorials/amqp-concepts.html

Request Collapsing

The request collapsing design pattern is the API request counterpart to API response caching. In this pattern, multiple adjacent requests to popular API methods are collapsed into a single request. For example, a web application that displays a video might make a request to a Digital Asset Management system to get metadata about the video, such as its duration. In a conventional application, each user accessing the page would result in a separate request for the video's metadata. With request collapsing, multiple requests over a given time interval can be collapsed into a single request. This reduces network bandwidth and load on the API end point, allowing it to scale to support a larger number of concurrent users.

To collapse requests, a proxy for the API end point queues up requests over a defined window of time, for example: 10ms. Periodically, the queue is "drained" and a single request is made to the API. The response is then distributed concurrently to the requestors. This optimization makes sense for only very high concurrent requests. If only a single request occurs at the start of the request window, it will incur a penalty equal to the size of the window. This introduces latency and may impact performance. In general, requests will fall somewhere within the window and will average a latency equal to half the request window (that is, 5ms for a 10ms window). Figure 6 provides an example of request collapsing.

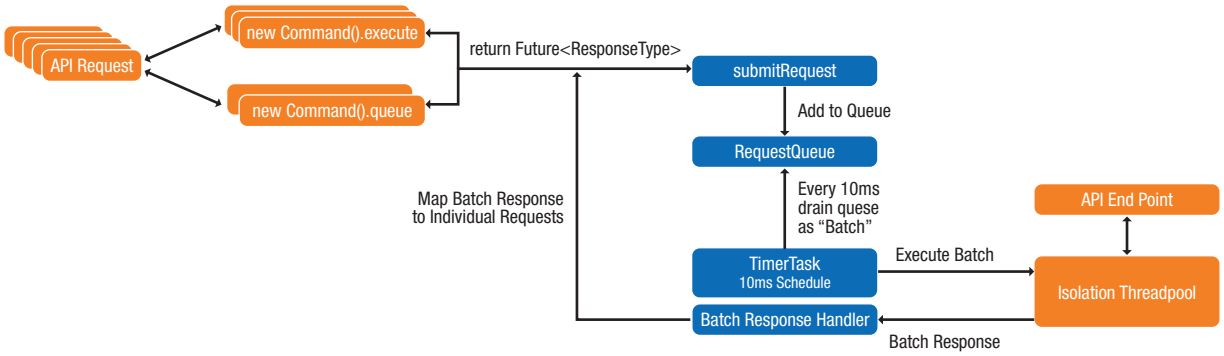


Figure 6. Example of request collapsing.

Object Change Notification

In a conventional, tightly coupled architecture, a fixed relationship exists between the components of the system. For example, two components, A and B, have a dependency where A notifies B of changes to objects that it manages. A also knows that there is only one instance of B, and it knows the location of B. If B is inaccessible for some reason, A cannot deliver the notification and must implement mechanisms for handling this scenario resiliently. The dependency between A and B creates a single point of failure and limits scalability.

A distributed system has many of these dependencies, with each one reducing the resiliency of the whole. One solution to this issue is to introduce redundancy into the architecture, so instead of a one-to-one relationship between components, there is a one-to-many relationship. For example, instead of a single B depending on A, there can be multiple instances of B, where a failure of any given B has a negligible effect on A.

The object change notification design pattern, also known as the observer pattern, enables this type of event handling and can be used to increase resilience.

With this pattern, the notifying component (A) implements an "observable" interface that enables "observers" (B) to register interest in changes in the observable component. When something changes in the observable, it notifies all observers of the change. Observers either receive the details of the change in the notification message (push model) or query the observable to get the event details (pull model).

In a cloud environment, this pattern enables an application to scale elastically. Since components are no longer tightly coupled, additional instances can be added dynamically to handle increased load. This is especially valuable for compute or I/O-intensive tasks that are amenable to parallelization. For example, Netflix must convert video masters into multiple versions with different resolutions and bit rates for different devices and network bandwidths. With the object change notification pattern, when a new video is loaded into storage, an agent detects the change to the storage service and can spin up a set of transcoding workers to create each of the resolution and bandwidth versions of the video in parallel. After completing processing, each agent can shut down.

OBJECT CHANGE NOTIFICATION DESIGN PATTERN

The following points highlight the object change notification design pattern:[1]

- Benefits
  - Improved resilience to failure
  - Increased scalability
  - More efficient resource utilization
- Where to use this pattern
  - Where compute intensive processing can be parallelized
  - To eliminate single points of failure
  - To reduce tight coupling between components to allow for implementations to be replaced

[1] Object Change Notification – Google Cloud Storage https://developers.google.com/storage/docs/object-change-notification

Service Discovery

In distributed applications, components make requests to service components. To make a request, a component needs to know the address of the service. In conventionally hosted multi-tier applications, the hostname of the service is stored in a configuration file. DNS is then used to look up the actual IP address. For scalability, the IP address may refer to a load balancer that enables multiple instances of a service to be referenced through a single address.

Cloud environments are highly dynamic, with instances of services and application components continually coming into existence and then disappearing. This occurs as auto-scaling responds to changes in load, and due to software failures in service instances or infrastructure failures in VMs, storage, or networking. The problem with DNS-based load balancing in the cloud is that requests may be directed to instances that no longer exist or have failed. The solution is a cloud service discovery mechanism through which only healthy service instances are located at request time. Such a service discovery mechanism should possess the following capabilities:

- Provide a way for components to direct requests to available instances.
- Support dynamic registration and deregistration of service instances.
- Maintain awareness of the state of any given instance.

Netflix implemented a service registry called "Eureka" with these capabilities.[6] This is a distributed service comprising Eureka servers and clients that are embedded in both application client and service components.

As shown in Figure 7, the Eureka server maintains a registry of healthy services. Services register with the server and regularly send a heartbeat to renew their lease. If a client fails to contact the server, the server removes it from the registry after a timeout period. So as new service instances are launched, these become visible to Eureka as they register. Likewise, as instances are terminated or fail, they are automatically purged from the registry without depending on the client deregistering. The registration approach also means that only services that are ready to process requests are listed in the registry, not services that are still coming online.

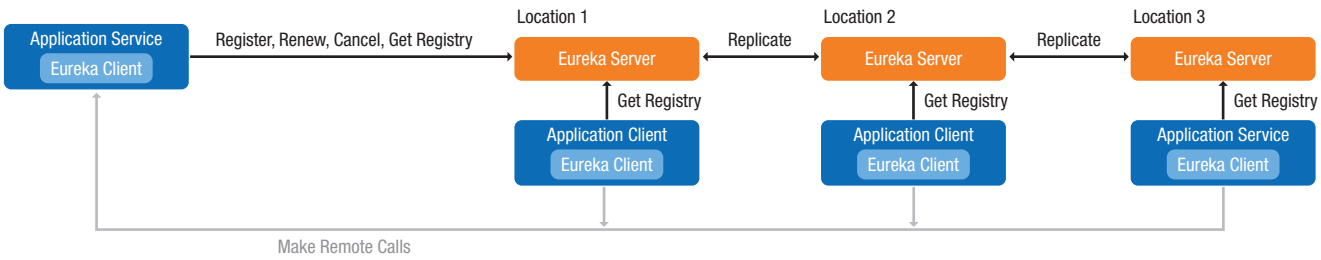


Figure 7. Netflix Eureka architecture.

[6] GitHub. Eureka at a glance. https://github.com/Netflix/eureka/wiki/Eureka-at-a-glance

Each application client component embeds the Eureka client. This maintains a cached copy of the services registry by regularly polling the Eureka server. In this way, knowledge of healthy services is disseminated throughout the environment, providing resiliency should the Eureka server fail or become unavailable for some reason. The Eureka client also acts as a load balancer, distributing requests across the available services according to an algorithm. The default algorithm is a round-robin, but this can be overridden.

In effect, Eureka pushes decisions about which service instance to contact for each request down to the clients. By encapsulating this behavior in a client library, service developers are largely freed from concerns about service availability. Since different services may process requests, Eureka requires services to be stateless. This is a prerequisite for scalability and resilience to failure and so is aligned with the general principles of cloud application architecture.

Because the client components have a cache of the available service instances, they can survive failure of the Eureka server. And, in a distributed environment with multiple data centers or cloud regions, per region instances of the Eureka server can survive a network partition because the services registry is replicated between regions.

SERVICE DISCOVERY DESIGN PATTERN

The following points highlight the service discovery design pattern summary:[1]

- Benefits
  - Improves resilience to failure
  - Simplifies infrastructure management
  - Simplifies application development in a dynamic environment
- Where to use this pattern
  - In an auto-scaling environment

[1] Eureka at a glance. https://github.com/Netflix/eureka/wiki/Eureka-at-a-glance

Microservices

With the microservices design pattern, a monolithic service is functionally decomposed into fine-grained microservices, each with a single function. For example, rather than a single message store service in an email messaging system that provides API methods for creating, reading, updating, or deleting messages, these functions would be decomposed into individual services. For instance, the read message feature would be provided by a service that just supports reading messages. The microservices approach has several advantages, including elasticity, performance, reliability, and ease of deployment.

Elasticity is improved with microservices because each microservice can scale independently of the others. For instance, if requests to read messages far outweigh requests to write messages, additional read microservices can be instantiated to handle the load. With a monolithic message store service, the whole service would need to be replicated, scaling unnecessarily and wasting resources.

Microservices simplify development and deployment of software updates. A software developer can own an entire microservice and can develop and release updates to the service independently from other developers. So if the developer of the read message service implements an improved message caching feature, this can be released to production without integrating with the other message store functionality. This decouples feature development so that developers can work in parallel and roll out updates on their own schedules.

"What is the best way to deploy onto the cloud, what is the best way to write reusable code, how do we respond rapidly to business change? The answer seemed to be micro-services. By creating small individual pieces, programs that did one thing very well – we reduce overhead and increase scalability. We give meaning to elasticity. Developing hundreds to thousands of small programs, removing deep levels of connescence (thank you Jim Weirich), and leveraging developer driven re-examination and re-factoring we create responsive, elastic cloud applications."
- Michael Forhan, AppFirst blog: "Lessons from Distill"

Netflix has developed an approach to deploying microservice updates. Rather than updating all instances of a service, Netflix will deploy one instance of a new version and smoke test it. If the new version fails, the impact on the system is negligible; the developer can fix the issue and redeploy quickly. If the microservice operates correctly, a new set of instances is spun up to replace the existing instances, which are left active. The new instances are monitored carefully for a few hours to ensure that issues that take longer to manifest, such as memory leaks, are identified. If problems are found, traffic can be redirected back to the old instances that are already running and ready to handle requests without being redeployed. If the updated microservices are functioning correctly, the old instances are automatically shut down after a period of time. Netflix utilizes a continuous deployment approach to automate this process, which speeds deployment, enables rapid iteration, and reduces the chance of errors. Netflix employs a simple philosophy: expect failures and be able to recover quickly without impacting the overall service.

Another benefit of microservices is that if a failure occurs, it is easy to identify the source of the problem because the scope is bounded by the microservice at fault. It is easier to identify which update triggered the failure and determine the developer who is responsible for the microservice.

MICROSERVICES DESIGN PATTERN

The following points highlight the microservices design pattern:[1]

- Benefits
  - Reduces overhead of deploying updates
  - Reduces side effects across co-resident services
  - Enables elasticity
  - Greater visibility into source of issues
  - Improved developer productivity
- Where to use this pattern
  - In large systems that need to scale elastically and cost effectively
  - To eliminate single points of failure
  - To improve performance
  - To support frequent updates to a critical production system

[1] Microservices. http://martinfowler.com/articles/microservices.html

Stateless Services

In a stateless service, no client state is maintained by the service between client requests. Consequently, each request must contain all the information the service needs to process the request. This design pattern provides many benefits in a distributed system, including the following:

- **Reliability.** Reliability is improved because the client can simply retry a failed request without the service having to recreate the state that existed prior to the failure.
- **Scalability.** Statelessness improves scalability for several reasons. Since a service instance does not store state, any instance can handle any request from any client. As request load increases, additional instances can be launched dynamically without impacting any of the existing instances. Also, since a service does not store any per-client state, the service remains lightweight in terms of resource consumption, ensuring more efficient use of resources and allowing each individual server to support a higher request load.
- **Visibility.** From an operational point of view, visibility is improved because both request and response payloads contain all the information needed to understand the transaction. This enables request proxying solutions, such as API management, to have greater visibility into the semantics of requests.
- **Simplicity.** Since the service does not store state, there is no need to manage request data or manage locks for the duration of a multi-request transaction. This makes it easier to implement stateless services. Services are easier to debug and more predictable; bugs caused by coincidences of state are eliminated.

The downside of stateless services is that payload sizes may be larger, because the state information the service needs to process a request must be provided in every request instead of being cached by the server. However, overall the benefits of stateless services significantly outweigh any disadvantages.

STATELESS SERVICES DESIGN PATTERN

The following points highlight the stateless services design pattern:[1]

- Benefits
  - Increases reliability
  - Improves scalability
  - Improves visibility for monitoring and request management
  - Simpler to implement
  - Less prone to state-sensitive bugs
- Where to use this pattern
  - Any distributed system requiring high scalability

[1] Service statelessness design principle. http://en.wikipedia.org/wiki/Service_statelessness_principle

Configuration Service

Runtime configuration settings of conventional applications are typically held within one or many files within the application's file system. In some cases, changes to these files can be made at runtime (in other words, they are hot deployable). In most cases these configuration files are loaded upon application startup, which means that if a change is made the application must be redeployed or restarted. This process results in downtime and additional administrative overhead.

Typically, these configuration files are stored and managed locally within the application, which results in redundant files spread across each node of an application deployment. As the application cluster grows, so does the overhead of managing each node's configuration file(s).

External Configuration Store

Making the application's configuration file persistent within an external data store provides a centralized repository for managing the cloud-aware application settings. Moving the configuration settings out of the local application instances provides easy management and control of data while sharing it across applications. The type of external data store can vary across the deployment environment.

A database or file repository provides the greatest flexibility for reading and writing data to the configuration file. Sensitive information such as passwords can be loaded during continuous integration (that is, injected dynamically during application build) or by means of environment variables set at runtime when the application is launched. Figure 8 illustrates the external configuration storage and application load process.

Figure 8. External configuration storage and application load process.

Runtime Reconfiguration

Well-designed cloud-aware applications minimize downtime while ensuring high availability. Any configuration change that requires application redeployment increases downtime. Runtime reconfiguration equips an application to detect configuration changes and to load the settings while the application is running.

Supporting runtime reconfiguration requires the application source code to appropriately handle configuration change notification events. Typically, this is accomplished through the observer pattern: an application will register with a centralized configuration service to receive updates on any changes. When a notification event is triggered, such as when a new configuration file is uploaded, the application requests the latest settings, reads the setting(s), and loads the configuration into memory.

Apache Zookeeper is an open source project from the Apache Software Foundation that supports runtime reconfiguration. Zookeeper provides a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. Zookeeper can be applied as a Shared Configuration Service, also known as Cluster Coordination.

Using ZooKeeper to store configuration information offers two main benefits:

1. New nodes can receive instructions about how to connect to ZooKeeper and can then download all other configuration information and determine the appropriate role they have within the cluster.
2. An application can subscribe to changes in the configuration, allowing tweaks to the configuration through a ZooKeeper client and modifications of the cloud-aware application cluster's behavior at runtime. Zookeeper runs on a cluster of servers called an "ensemble" that shares the state of the application data (see Figure 9). This configuration allows distributed processes to stay coordinated within the cloud-aware application cluster.

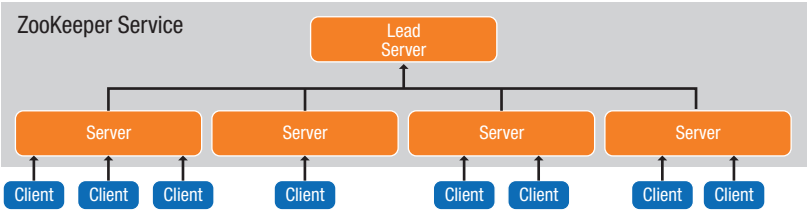


Figure 9. ZooKeeper service replication.

CONFIGURATION SERVICE DESIGN PATTERN

The following points highlight the configuration service design pattern:[1]

- Benefits
  - Single, centralized data store for configuration management
  - Reduces application configuration management overhead
  - Reduces application downtime
- Where to use this pattern
  - Any distributed system requiring application configuration
  - Where high availability with reduced downtime is required

[1] Apache ZooKeeper http://zookeeper.apache.org

Authorization Pattern

In a cloud environment, the network cannot be assumed to be secure because it is not under the application owner's control. This means that a third party may intercept and redirect or snoop data in transit. Another risk is that APIs may be accessible to other tenants in a multitenant cloud environment or from the public Internet. This exposure creates a risk that the API will become the target of a denial-of-service attack, enable someone to access proprietary information, or provide a way for the service to be deliberately sabotaged. Cloud applications are particularly susceptible to these attacks because they are distributed systems with interdependent services that communicate over a network. Each network dependency and API increases the risk to the application. The solution is to encrypt network traffic and to protect APIs with authorization mechanisms.

API Authorization

API authorization ensures that only trusted API consumers are able to perform API actions. This approach protects the service from malicious access and provides a way for the service to support multiple clients with different access rights. For example, some clients might be granted read-only access to the service, and others might have both read and write access. Authorization can also be used to enforce architectural constraints by limiting which clients are able to access an API. For example, a database API might be protected with access permissions that ensure that clients use a preferred data access service. In this case, the database API would grant read/write access only to the data access service.

Applications have traditionally used usernames and passwords for client-server authentication. There are two main problems with this approach: The client needs to store the username and password somewhere, which makes them potentially discoverable, and passwords are notoriously vulnerable. Managing usernames and passwords in each component of an application also adds complexity, management overhead, and the risk of misconfiguration.

The OAuth 2.0 standard (RFC6749) addresses these issues by introducing the concept of token-based access control. In OAuth, an authorization server issues tokens to clients that grant specific access rights to protected resources.

As shown in Figure 10, for OAuth-gated services:

- A client obtains an access token for a service by authenticating with an authorization server (1).
- The authorization server verifies the authentication credentials and, if valid, returns an access token (2).
- The client passes the access token in all requests to the service (5).
- For each request, the service checks the validity of the token with the authorization server (6).
- The authorization server provides the service with the set of permissions associated with the access token (7).
- The service uses the permissions to determine if the requested operation is permitted and conditionally performs the action and returns the data in a response to the client (8).

The request and response are no different from conventional web service standards apart from the passing of the access token in the request. Note that an access token has a limited lifetime for security reasons. When it expires, the client uses a special refresh token to request a new access token.

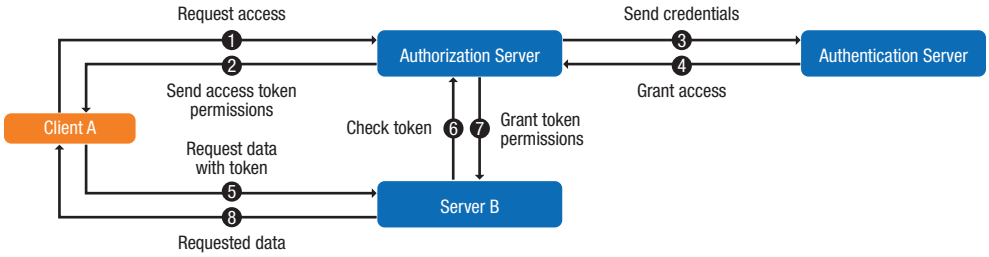


Figure 10. Authorization server communication.

AUTHORIZATION DESIGN PATTERN

The following points highlight the authorization design pattern:[1]

- Benefits
  - Improved security
  - Fine-grained access control to services
- Where to use this pattern
  - For all client-to-service and service-to-service requests

[1] OAuth 2.0 specification: tools.ietf.org/html/rfc6749; Practices for Secure Development of Cloud Applications: www.safecode.org/publications/SAFECode_CSA_Cloud_Final1213.pdf

ASSESSING APPLICATION CLOUD MATURITY

The Cloud Application Maturity Model provides a simple way to assess the level of cloud maturity of an application, just as the Richardson Maturity Model measures the maturity of a REST API. The maturity model suggests changes that can be implemented to increase an application's resilience, flexibility, and scalability in a cloud environment.

As listed in Table 2, there are four levels to the maturity model with level 3 representing the highest level of maturity and level 0 representing applications that are not cloud aware.

Table 2. Cloud application maturity level.

Maturity Level	Description
Level 3: Adaptive	• Application can dynamically migrate across infrastructure providers without interruption of service. • Application can elastically scale out/in appropriately based on stimuli.
Level 2: Abstracted	• Services are stateless. • Application is unaware and unaffected by failure of dependent services. • Application is infrastructure agnostic and can run anywhere.
Level 1: Loosely Coupled	• Application is composed of loosely coupled services. • Application services are discoverable by name. • Application compute and storage are separated. • Application consumers one or more cloud services: compute, storage, network.
Level 0: Virtualized	• Application runs on virtualized infrastructure. • Application can be instantiated from an image or script.

Maturity Level 0: Virtualized

Level 0 is the lowest and least sophisticated maturity level. An application at this level need only run in a virtualized compute resource such as a VM. Applications are instantiated either from a saved machine image or dynamically using scripts such as Opscode Chef. Such applications are typically tightly coupled, with explicit dependencies between all components of the application.

Maturity Level 1: Loosely Coupled

Level 1 introduces the concept of **loose coupling** to address the limitations of Level 0. Loose coupling means that components do not have tight dependencies on each other, so if one component fails, other components are unaffected and the application can continue to function normally. Loose coupling also makes it easier for an application to scale, by enabling components to operate asynchronously.

Another approach that enables loose coupling is to expose **implementation-independent interfaces** that enable a service to transparently swap out one implementation for another or support multiple underlying implementations. Implementation independence is crucial if an application is to run across multiple cloud providers.

Separating compute and storage also facilitates loose coupling. By separating compute and storage, applications become agnostic to the underlying storage mechanism, allowing storage to scale and be managed independently of compute resources. This is a fundamental requirement for delivering the Level 2 capability of being able to run anywhere on any cloud infrastructure. Note that decoupling compute from storage means all storage, including log and configuration data, not just the data that the service consumes or manages.

Maturity Level 2: Abstracted

This level introduces the ability to run on any cloud-computing infrastructure. This is a challenge today because the major cloud computing platforms do not have the same service interfaces or support the same VM image formats. This limits interoperability to the extent that cloud applications that can run without modification across multiple cloud providers are rare. Engineers must implement cloud-agnostic abstraction layers and scripts that enable applications to be instantiated on any virtualized infrastructure, and deliver **identical functional behavior**. This is significant additional work and is a decision that needs careful cost/benefit analysis. The benefits are that the application is not tied to a particular provider's proprietary interfaces, so vendor lock-in is avoided, and dynamic service migration at runtime is possible.

In addition to infrastructure abstractions, applications at this level are not dependent on the availability of specific instances of services. These applications are as completely isolated from failures of other services as they are from packet retransmission at the network layer. Indeed,

neither the application nor its users should have any awareness that a component of the application failed. One way to deliver this is through the Circuit Breaker design pattern that Netflix utilizes in their Amazon-based services.

Finally, since a service instance may fail at any time, it must be possible for another instance to pick up and resume where the failed service left off. This can be achieved only if service instances do not possess any private state. Either the state should be shared by all service instances, or there should be no stored state.

Maturity Level 3: Adaptive

Level 3 represents an application that has been designed to fully exploit the unique qualities of a cloud environment. Applications at this level of maturity are able to migrate seamlessly from one cloud environment to another, either partially or completely, without any interruption in service. This requires that they exhibit all of the preceding levels of maturity. The responsibility for orchestrating the migration of applications and their services lies in some kind of cloud "master control program" that can determine when to migrate applications and where to migrate them to. For example, the program might choose to migrate an application from one cloud environment to another to take advantage of off-peak resource pricing or in response to a degradation of service due to load or environment instability.

Level 3 applications can also exploit the elastic scalability of the cloud, dynamically scaling up in response to load or down in periods of lower activity. Applications might scale horizontally by adding more instances of a service or vertically by adding virtual CPUs or memory to existing instances. Again, this is likely not the role of the application, though it would need to define metadata that enables some kind of agent to determine the appropriate way to scale a given service.

Operational Strategies

This section contains the key operational principles for cloud-aware applications (see Table 3). Operational strategies refer to how the application is deployed and operated, and how other system components interact with it. The focus of these operational strategies is the entire system for end-to-end usage of an application over the course of its lifecycle.

Table 3. Operational strategies.

No.	Principle	Attributes	Priority
1	Ensure redundancy	Applications are designed to be resilent to failure though the use of redundancy.	High
2	Utilize caching	Caching is used to improve performance, increase resiliency, and reduce bandwidth costs.	High
3	Secure access to APIs	API end points are protected via API Management gateways.	High
4	Stage deployments	The risk of failure due to deployment is reduced by staging the introduction of updates to components of an application.	Medium
5	Plan for zone/ region failures	Applications are deployed in a way that is resilient to disasters, such as entire geographic zone or region failures.	Medium
6	Minimize inter-zone/ region latency	Applications are deployed geographically to minimize network latency.	Medium
7	Locate high-bandwidth consumers externally	High bandwidth consumers are hosted outside the cloud to reduce cloud bandwidth utilization costs.	Medium
8	Abstract dependencies	API abstractions are used to prevent lock-in with proprietary cloud services.	Low

Ensure Redundancy

Failures of application components, VMs, networking, storage, and other cloud infrastructure should be anticipated. By designing in redundancy, the impact of these failures can be limited. For example, if the cloud environment provides multiple geographic zones, the entire application and the services it depends on can be replicated in multiple zones. If a failure occurs in one zone, traffic can be redirected to the healthy zone, ensuring continuity of service.

Utilize Caching

Caching is a technique that is widely used in conventional as well as cloud environments. It improves performance by keeping frequently accessed data close to the services that consume it. It also improves resiliency by allowing cached data to be accessed in the event of a failure of the source system. In a cloud environment, caching reduces network traffic. This reduces operational costs when cloud service providers meter and bill for bandwidth use.

In addition, for data strategy in general, the ODCA suggests that enterprises adopt an IaaS approach. This approach helps to provide standardized and secure methods to create, manage, exchange, and extract meaningful information from all available data in the right format at the right time. See the Information as a Service Master Usage Model[7] to explore the overall information architecture required for IaaS implementation, and what aspects of that information architecture can be enhanced by big data, mobile applications, and cloud computing technologies.

Secure Access to APIs

Securing API access ensues that applications and services are only accessible to the clients that should have access to them. As explained earlier, there are no guarantees that the services deployed to the cloud are secure. One solution is to front each service with an API Management proxy. This not only limits access, but also enables auditing and reporting to track how the API is being used and by which clients. Netflix uses this approach to limit access to core shared services. In this way, Netflix can ensure that only its developers can route requests through the service access layers.

Stage Deployments

In a large, distributed system, software updates can lead to unexpected behaviors due to bugs, side effects, and unforeseen incompatibilities. As systems become larger, more dynamic, and more complex, the ability to anticipate and prevent these issues decreases. With Agile development practices, small, frequent updates from different teams are the norm, which also adds to the risk.

One approach to managing risk is to deploy updates on a small scale and carefully monitor the behavior of the new version. The new and old versions of a component operate in parallel, reducing the likelihood of a new version taking down the entire system. If the small-scale deployment is successful, the update can be deployed on a larger scale to a new set of VMs, with the old ones left running as failover support. If the new version fails, requests can simply be redirected to the old instances. If the new version operates successfully, the VMs associated with the old version can be shut down and reclaimed.

Plan for Zone/Region Failures

Availability Zones provide a way to group cloud resources such that they are isolated and independent of other zones. Amazon structures its EC2 environment into geographic regions, each of which contains multiple Availability Zones. The history of outages in Amazon's cloud services suggests that outages can impact entire regions or entire zones. It is therefore essential when planning a cloud deployment to design in resiliency to failure at both a region and zone level. For example, Netflix replicates its entire service across multiple regions so that if there is a failure in one region, another region can take over the traffic normally handled by the failed region. This requires data replication and synchronization across regions.

Applications can be deployed in active/passive or active/active models in availability zones. In active/passive, an application is deployed to two different locations or availability zones offering IaaS or platform as a service (PaaS). Both these locations can run instances of the application, but only one location is active at a time. A local load balancer handles the localized application instances, and a global load balancer performs health checks to make decisions where to send the end-user requests.

In active/active, the application is deployed to two locations but both are active, running simultaneously, handling different users, and ready to fail over to each other should it become necessary. If one location fails, global load balancers help redistribute the load across the remaining locations. The advantage of active/active is improved utilization of cloud resources and a better ability to provide service availability closer to users through multiple active locations.

Minimize Inter-Zone/Region Latency

If the cloud environment is geographically distributed over long distances, or zones are split across data centers, network latency can become a factor.

For services that require low latency, plan to deploy them in the same region and possibly in the same zone, subject to availability concerns. One of the challenges in deploying applications in the cloud is that the physical structure of the environment is opaque, unlike a data center where it is possible to request services be deployed to the same physical rack. Careful analysis can provide some insight into the structure of the cloud, but be aware that this may change over time and be outside one's control.

[7] See www.opendatacenteralliance.org/library

Locate High-Bandwidth Consumers Externally

Consider where data should be stored to minimize data transfer charges. For example, Amazon charges for data transfers from EC2 to the public Internet. Depending on the nature of the application, these costs could become excessive. Locating data externally to the cloud, such as in a CDN, can be more cost effective.[8]

Abstract Dependencies

Avoid creating static dependencies between clients and servers. For example, an application might have a dependency on a storage service. Over time the interface, implementation, and location of the storage service might change. For example, the application might migrate to a different cloud provider where an equivalent service has a different interface and will certainly have a different address. API Management can help abstract the underlying service and implementation from clients by exposing an implementation-neutral REST API. This loosely coupled approach makes is possible for components to evolve independently of each other.

CONCLUSION

As technology continues to rapidly evolve, developers need to continually expand and enhance their skills—refining and adopting new techniques to architect, design, develop, and support robust cloud applications. To successfully leverage cloud capabilities, organizations need to do the following:

- Invest in training programs for their staff to ensure that they are equipped with the skills to build cloud-aware applications.
- Find ways to free developers from spending large amounts of time working on low-level, low-value technical tasks.
- Deliver massively scalable, complex applications that are simple to modify and can easily integrate with their existing systems.

Recommendations for Developers

To benefit from the unique characteristics of the cloud-computing paradigm, application developers need to adopt new architectural design patterns that build on principles typically associated with distributed systems. These principles help cloud-resident applications to scale elastically, seamlessly handle failures in the infrastructure, and make efficient and optimal use of resources that are billed on consumption.

The strategies outlined in this paper address the challenges inherent in distributed systems. Design patterns that have proven successful in large-scale cloud-based applications are presented for developers to leverage as they architect their own applications. Undoubtedly, new design patterns will be created as developers learn how to more effectively exploit the cloud environment and gracefully handle new, unanticipated challenges unique to this environment.

The Cloud Application Maturity Model provides developers with a standard way to measure the cloud maturity of their applications. This can be used as a roadmap for determining what to focus on to evolve the cloud maturity of an application.

Cloud Platform Requirements

The following requirements apply to any cloud computing provider:

- Support cloud computing industry standards, preferably open standards or widely supported de facto standards. This support will prevent lock-in to proprietary vendor APIs and is a prerequisite for applications to run seamlessly across a hybrid cloud environment.
- Provide standard VM images that are consistent with those provided by other cloud computing platforms. For example, if developing a private cloud, consider creating images that match those provided by Amazon EC2 if application portability between private cloud and Amazon is a current or planned requirement.
- Support data affinity (colocate data with producer/consumer).
- Use standard version-controlled scripts for creating custom VM configurations.

We are in the midst of a significant technology transition heralded by cloud technology, promoting a truly distributed technology world for developers and businesses. Technologists have an opportunity to deliver business transformation and true value by adopting new approaches and techniques to suit cloud-aware application development. The ODCA hopes that this paper serves as a starting point for the study and adoption of new application architectures within the cloud. The ODCA believes that elevating application development skills to meet the rising demand for applications that take full advantage of cloud capabilities requires ongoing education and training, resulting in a sustained focus on development of skills, evolution, and innovation in the workplace.

---

[8] See http://aws.amazon.com/ec2/pricing/#DataTransfer

REFERENCES

- "Architectural Patterns for High Availability"
  www.infoq.com/presentations/Netflix-Architecture
- *Cloud Architecture Patterns*
  www.amazon.com/Cloud-Architecture-Patterns-Using-Microsoft-ebook/dp/B009G8PYY4/
- "Cloud Characteristics, Principles and Design Patterns"
  www.gartner.com/document/2081915
- "Cloud Design Patterns: Prescriptive Architecture Guidance for Cloud Applications"
  http://msdn.microsoft.com/en-us/library/dn568099.aspx
- "Building Cloud-Aware Applications"
  www.slideshare.net/cobiacomm/building-cloudaware-applications [slides 17, 18, 19, 31]
- "Searching for Cloud Architecture…"
  http://blog.cobia.net/cobiacomm/2011/11/25/searching-for-cloud-architecture/
- "Maximizing Cloud Advantages through Cloud-Aware Applications"
  www.intel.com/content/dam/www/public/us/en/documents/white-papers/maximizing-cloud-advantages-through-cloud-aware-applications-paper.pdf [page 6]
- Network Latency between U.S. cities (AT&T)
  http://ipnetwork.bgtmo.ip.att.net/pws/network_delay.html
- "The Fallacies of Distributed Computing Reborn: The Cloud Era – New Relic blog"
  http://blog.newrelic.com/2011/01/06/the-fallacies-of-distributed-computing-reborn-the-cloud-era/
- "5 Lessons We've Learned Using AWS" – Netflix Tech Blog
  http://techblog.netflix.com/2010/12/5-lessons-weve-learned-using-aws.html
- "Optimizing the Netflix API" – Ben Christensen
  http://benjchristensen.com/2013/04/30/optimizing-the-netflix-api/
- "Architecting Applications for the Cloud: Best Practices" – Jinesh Varia, Amazon Technical Evangelist
  http://media.amazonwebservices.com/AWS_Cloud_Best_Practices.pdf
- "Lessons from Distill" – Michael Forhan, AppFirst
  www.appfirst.com/blog/lessons-from-distill/
- "Understanding Security with Patterns" – Prof. Peter Sommerlad, Tutorial T39 @ OOPSLA 2006
  http://wiki.hsr.ch/PeterSommerlad/files/T39-Sommerlad.pdf
- "How it Works" (Circuit Breaker implementation in Hysterix open source library) – Netflix
  https://github.com/Netflix/Hystrix/wiki/How-it-Works
- Netflix Shares Cloud Load Balancing and Failover Tool: Eureka! - Netflix Tech Blog
  http://techblog.netflix.com/2012/09/eureka.html
- "Dystopia as a Service" – Adrian Cockcroft, Netflix
  www.slideshare.net/adrianco/dystopia-as-a-service
- "Netflix and Open Source"
  www.eiseverywhere.com/file_uploads/c6b285b64a18cc2742c0fb20061d6530_OBC_BreakoutSession_AdrianCockcroft_1pm.pdf

APPENDIX A: GLOSSARY OF TERMS

Term	Definition
Agent-Based	Refers to agent-based cloud computing, a form of computing using software agents to facilitate the delivery of cloud services. These agents can be proprietary (specialized) APIs or open APIs.
Agentless	Refers to the use of standard programming interfaces (open APIs) to discover a cloud service in which a remote API is exposed by the cloud service or direct analysis of a communication network operating between the cloud provider and the cloud broker.
Application program interface (API)	Describes a set of routines, protocols, and tools for building software applications. A well-designed API makes it easier to develop a program by providing all the building blocks. A programmer uses the blocks to construct the application. Most operating environments, such as Microsoft Windows, provide an API to let programmers write applications that are consistent with the operating environment.
Cloud application	A cloud application (sometimes called a cloud app) is an application program in which the software and associated data are centrally hosted on the cloud, retaining some of the characteristics of a native desktop app and some characteristics of a pure web app.
Cloud consumer	A person or organization that engages with a cloud provider, either by means of a cloud broker or by means of a direct engagement with a cloud provider to receive cloud services.
Data store	A permanent storehouse of data. The term is often used to lump the storage of all types of data structures (including files, databases, text documents, and so on) into one generic category.
Hybrid cloud	The cloud infrastructure is a composition of two or more distinct cloud infrastructures (private, community, or public) that remain unique entities, but are bound together by standardized or proprietary technology that enables data and application portability (for example, cloud bursting for load balancing between clouds).
Infrastructure as a service (IaaS)	The capability provided to the consumer to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications. The consumer does not manage or control the underlying cloud infrastructure, but has control over operating systems, storage, and deployed applications. Control is sometimes granted over select networking components (such as host firewalls).
Multitenancy	Multitenancy refers to a principle in software architecture where a single instance of the software runs on a server, serving multiple client-organizations (tenants). Multitenancy contrasts with multi-instance architectures where separate software instances (or hardware systems) operate on behalf of different client organizations. With a multitenant architecture, a software application is designed to virtually partition its data and configuration, and each client organization works with a customized virtual application.
Multi-tier architecture	In software engineering, multi-tier architecture (also referred to as n-tier architecture) is a client-server architecture in which presentation, application processing, and data management functions are logically separated. The most widespread use of multi-tier architecture is the three-tier architecture.
OAuth	OAuth is an authentication protocol that allows users to approve an application to act on their behalf without sharing credentials such as usernames and passwords.
Platform as a service (PaaS)	The capability provided to the consumer to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages, libraries, services, and tools supported by the provider. The consumer does not manage or control the underlying cloud infrastructure, including networks, servers, operating systems, or storage, but has control over the deployed applications and possibly configuration settings for the application hosting environment.
Private cloud	The cloud infrastructure is provisioned for exclusive use by a single organization comprising multiple consumers (such as business units). It may be owned, managed, and operating by the organization, a third party, or some combination of them, and it may exist on or off premises.
Public cloud	A form of cloud infrastructure provisioned for open use by the general public. It may be owned, managed, and operated by a business, academic, or government organization, or some combination of them. It exists on the premises of the cloud provider.
Representational state transfer (REST)	A style of software architecture for distributed hypermedia systems, such as the World Wide Web. REST-style architectures consist of clients and servers. Clients initiate requests to servers; servers process requests and return appropriate responses. Requests and responses are built around the transfer of representations of resources. HTTP, for example, has a very rich vocabulary in terms of verbs (or "methods"), URLs, Internet media types, request and response codes, and so on. REST uses these existing features of the HTTP protocol and thus allows existing layered proxy and gateway components to perform additional functions on the network, such as HTTP caching and security enforcement. Conforming to these REST restraints is referred to as "RESTful."
Software as a service (SaaS)	The capability provided to the consumer, using the provider's applications running on a cloud infrastructure. The applications are accessible from various client devices through either a thin client interface, such as a web browser (web-based email), or a program interface. The consumer does not manage or control the underlying cloud infrastructure, including network, servers, operating systems, storage, or even individual application capabilities, with the possible exception of limited user-specific application configuration settings.
Solution provider	A technology vendor selling technology elements that can be used to build a cloud or other service, usually specializing in either a specific software product, a specific hardware product, or possibly by providing consulting services.

APPENDIX B: CLOUD ANTI-PATTERNS

Cloud anti-patterns—specific design practices that prevent applications from operating in the cloud—should be avoided. Not all design patterns are relevant and helpful within a cloud-aware application. The anti-patterns in the following list restrict the ability of an application to leverage the cloud:

- **Complex configuration.** Avoid unnecessarily difficult and verbose configuration settings. If hot-swappable configuration files or dynamically loaded files are not required, rely on persistent configuration settings. If persistent configuration settings are provided in the application's continuous integration environment, these settings can be accessed and loaded into the application at build time. This approach avoids placing an unnecessary load on the configuration server. Upon deployment, excessive requests by the cloud application to retrieve configuration settings across clusters of nodes could inadvertently result in a denial-of-service situation.
- **Complicated dependencies.** Avoid unnecessarily including library dependencies in an application. These dependencies can lead to large deployments that require longer application linking, loading, or introspection by various components. To minimize dependencies, use tools that show existing build dependencies and only include those that are required for runtime deployment of an application to the cloud.
- **Managed shared state.** Avoid retrieving an application's state upon startup, which creates latency that prevents an application from handling incoming requests quickly. Rely on lazy initialization to retrieve states only when needed, rather than unconditionally when the application starts.

APPENDIX C: SECURITY ACCESS CONTROL

Access control techniques provide the configurable mechanisms for restricting system access. Particular design patterns can be applied to enable access control across cloud-aware components. Both role-based access control (RBAC) and attribute-based access control (ABAC) provide system-level authorization.

RBAC is also referred to as "role-based security." The RBAC model defines roles that specify levels of authority. Each role is assigned a permission authorization that determines the level of security associated with the role (for example, access to compute resource to execute a specific function). In turn, that role and permission are assigned to one or many subjects. A subject could be a person or autonomous agent (such as an automated provisioning service for creating and deploying compute resources).

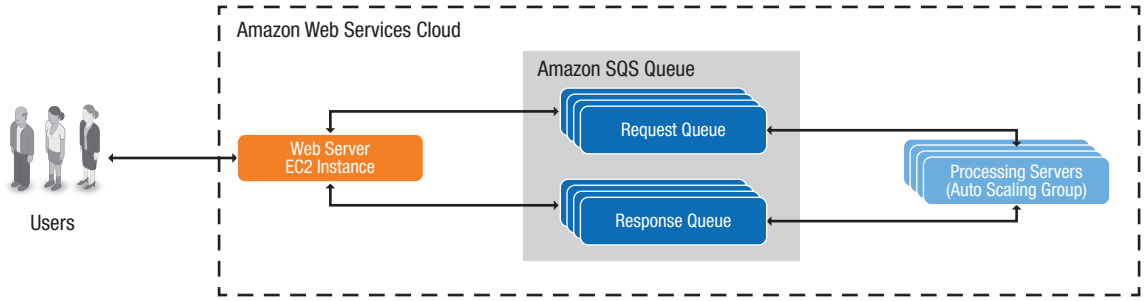
ABAC is also referred to as "attributed-based security." The ABAC model is a relatively new paradigm that combines subjects, policies, and attributes to define logical access control. This logical definition lets users simply define complex access control policy. Conversely, RBAC relies on manually updating a policy based on a subject's current state and corresponding role (for example, job role or responsibilities).

APPENDIX D: AUTO-SCALING

Auto-scaling controls the dynamic scaling of compute resources—up and down—to meet requirements (see below). The scaling relies on monitoring that can occur at a high level within the deployed application and/or at the low-level of the system. The monitoring attribute can be defined using upper or lower thresholds (such as CPU usage or message queue size). When a given load for a cloud application is too high or low, exceeding the predetermined threshold, an alarm can trigger provisioning of more or less nodes for a cluster.

Numerous commercial and open source software solutions offer auto-scaling at the IaaS and PaaS platform levels. The level of application integration and automation vary across solutions. Current examples of auto-scaling solutions include:

- Amazon Auto Scaling
- Azure Management Portal Scaler
- Rackspace Auto Scale
- Scalr
- Eucalyptus Auto Scaling
- RightScale Auto Scaling



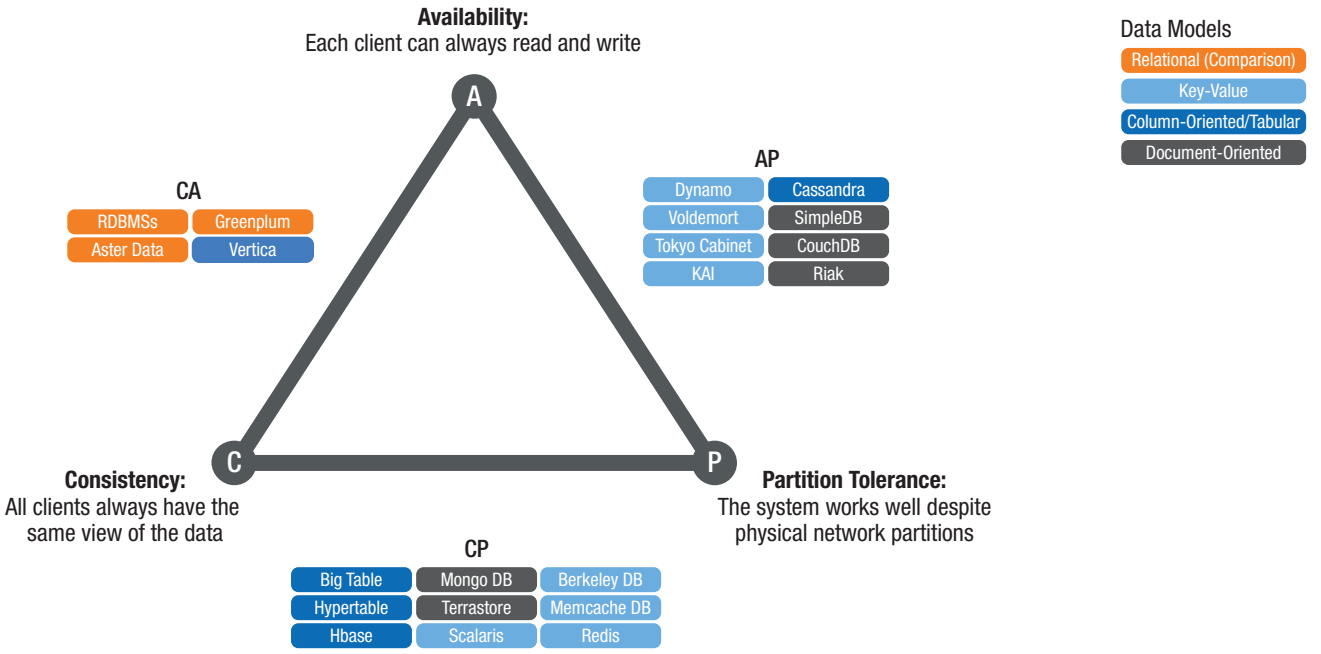
Determining when to auto scale using queue service (source: Amazon Web Services).

APPENDIX E: CAP THEOREM

The CAP theorem[9] developed by Eric Brewer states that a distributed system cannot simultaneously provide all three of the following desirable properties (see below):

- **Consistency.** All reads by across nodes see the same data (previous completed write).
- **Availability.** Reads and writes always succeed (acknowledge response).
- **Partition Tolerance.** System continues operation despite arbitrary message loss or failure of a component in the system (fault tolerance).

When implementing a BASE (Basically Available, Soft-State, Eventual Consistency) data store within a cloud-aware application, trade-offs exist that affect scalability, performance, and reliability. Brewer described these properties as being constrained to two of the three, leaving viable design options: CP, AP, and CA.



Visual guide to NoSQL systems (source: Nathan Hurst).

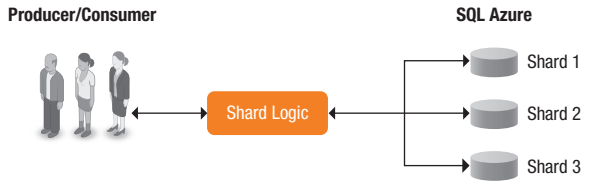
[9] http://en.wikipedia.org/wiki/CAP_theorem

APPENDIX F: DATABASE SHARDING

Data sharding is a design principle that defines the horizontal partitioning of rows in a database table (see below). Each partition forms part of a shard, which may be spread across a number of distributed servers. This partition scheme is defined as "shared-nothing" for large databases across a number of servers. Data sharding enables increased database performance and improved scalability, because of the reduction in the number of rows stored within a database table. Data sharding also improves throughput and boosts overall performance of the database. It also increases database storage capacity.

The term "sharding" originated from the Google research publication on the Bigtable architecture, "Bigtable: A Distributed Storage System for Structured Data."

Cloud-aware applications that rely on a large-scale data source are good candidates for sharding. Data sharding can reduce overall client latency within the cloud application, providing high scalability and throughput.



Multi-master sharding archetype (source: Microsoft).

APPENDIX G: DATA REPLICATION

Data replication involves creating and managing duplicate versions of a data store. The replicated data ensures consistency between redundant resources. Replication improves system reliability, fault tolerance, and accessibility in the event a data source failed and is unreachable. If a data source fails, a cloud-aware application can seamlessly roll over to a secondary data source, ensuring consistency and availability. This improves fault tolerance and enhances the availability and responsiveness of the cloud application.

Multiple replication models exist, each with a different approach to providing persistent data storage (for example, through a database, disk storage, or files).

APPENDIX H: STATIC CONTENT HOSTING

Web applications typically serve static content while processing dynamic page requests. Handling requests for clients to download static content consumes a large portion of compute cycles, reducing the efficiency of the application. A cloud-aware application relying on hosting static content can minimize this cost by offloading static content to a separate resource.

A content delivery network (CDN) can provide geographically dispersed storage and serving capabilities of static content. When static content hosting is removed from the application, more resources can be applied to serving requests faster resulting in higher throughput. Clients' request latency can be reduced by serving content from a CDN in close physical proximity. Overall, the clients' performance of page rendering and content serving improves while the cloud-aware application's throughput increases.