**AWS SQS- Message Queue Service**

**AWS SNS- Simple Notification Service**

**AWS RDS- Relational Database Service**

**AWS Security – AWS Security Directory & IAM**

**Build & Deploy Services-AWS Code Pipeline, Code Commit & Code Deploy**

**Message Service-AWS Api Gateway, AWS SWF & AWS SES**

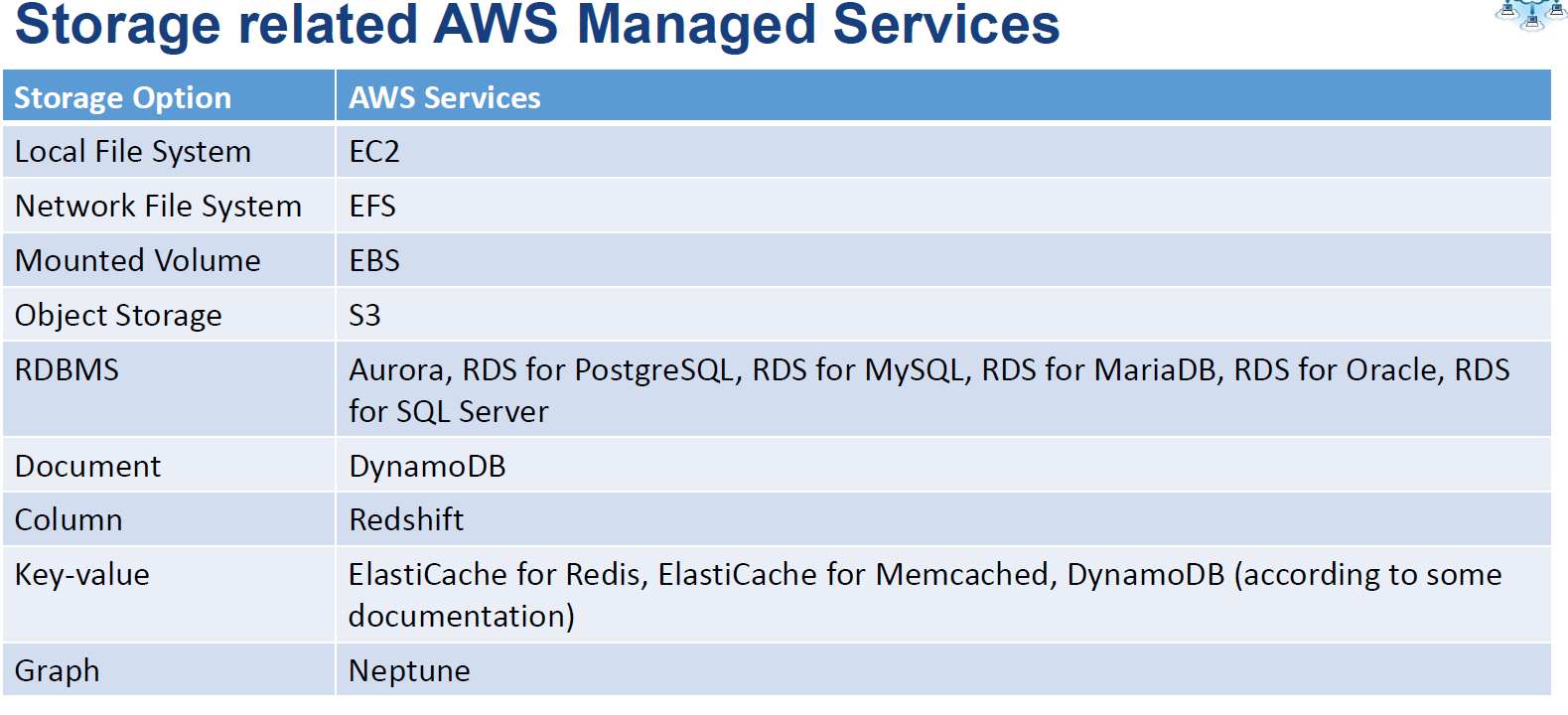
**AWS Analytics- AWS Data pipe Line & AWS EMR**

**Document Database- Mongo**

**Columnar Database-Apache CASANDRA, AWS Redshift, Apache HSBASE,B igQuery**

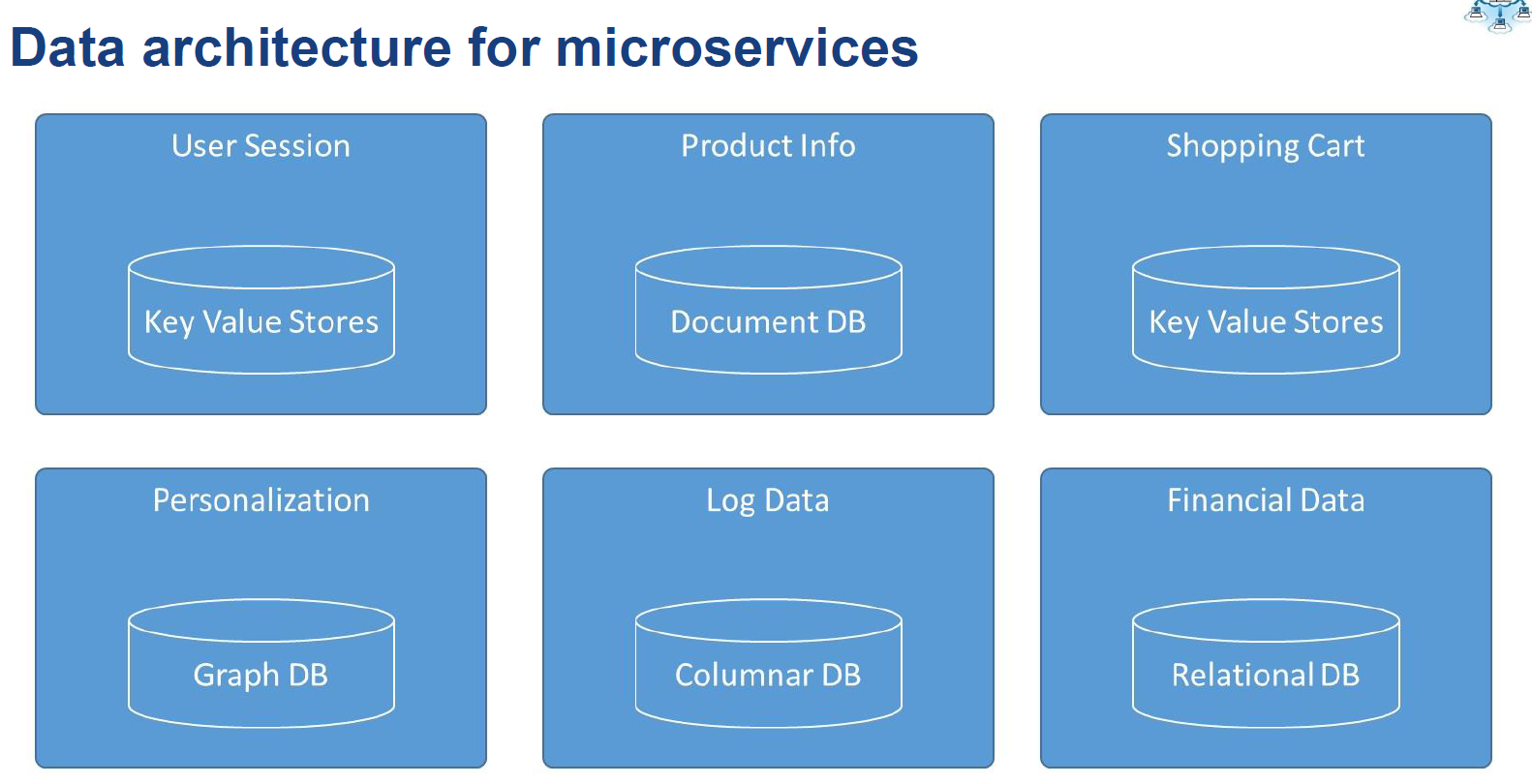
**Micro service Aggregation :If real-time business logic is applied to get aggregate data.It has lower dependency on api gateway but more chances to failover.**

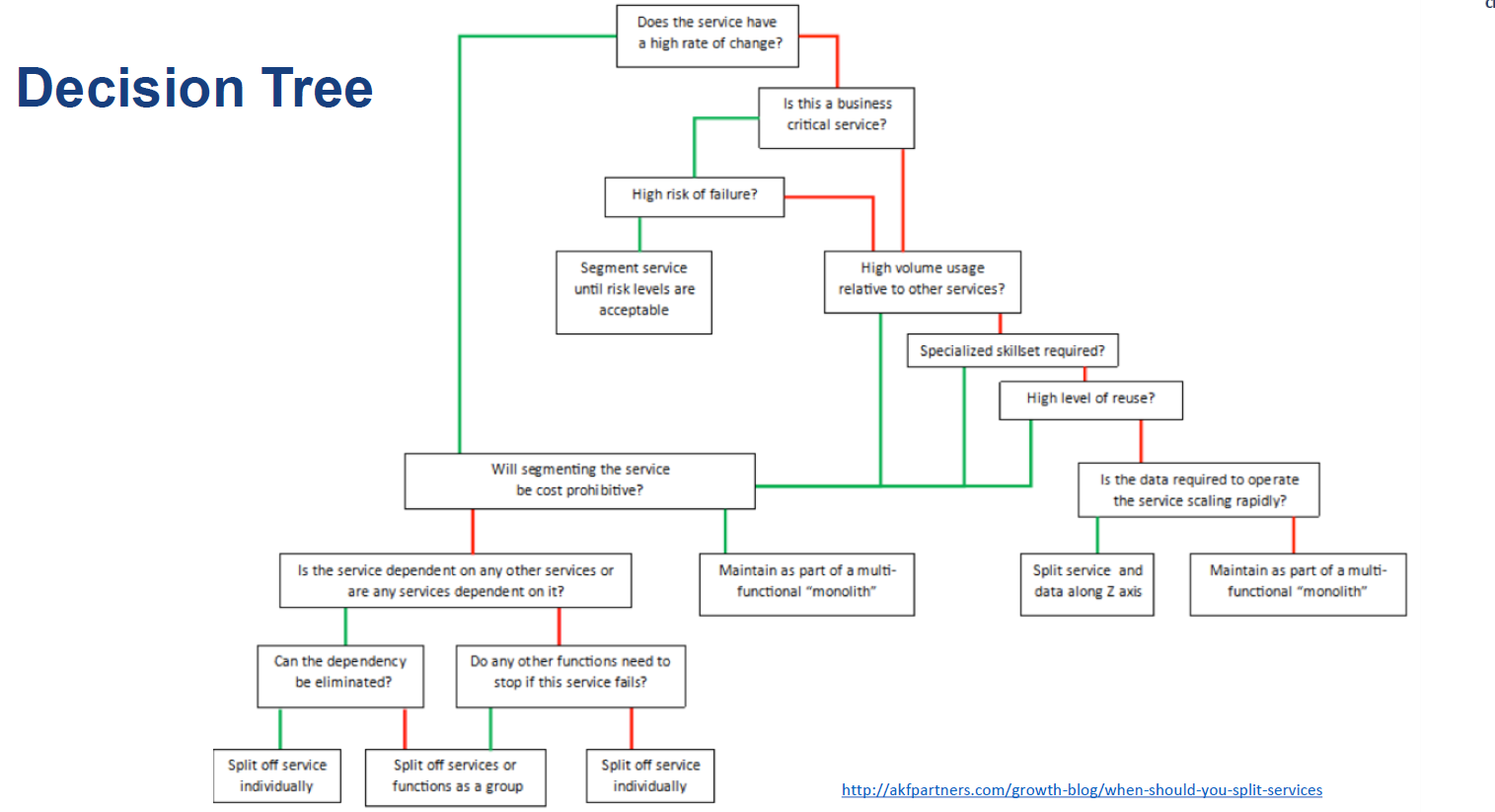
**Database Aggregation: Data is pre aggregated in operational data store ODS a document database.**

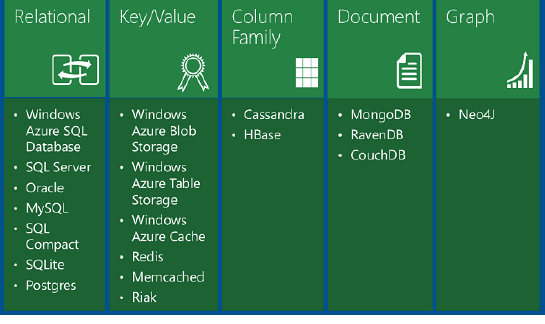


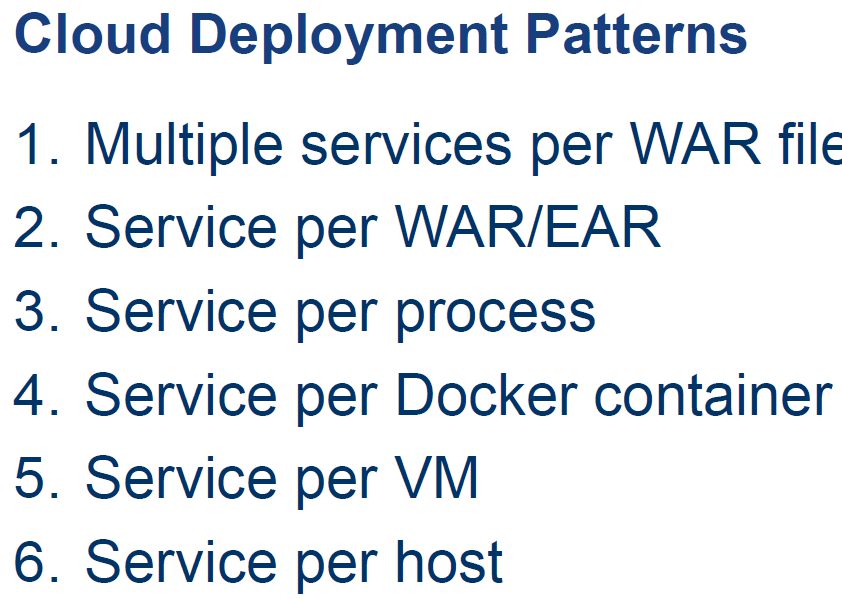
**Lift & Shift**

1. **Queue Services to replace by SQS**
2. **Replace Data store**
3. **Replace Security**
4. **Replace build Service**
5. **Replace Anaylytics**











**SaaS Advantages**

SaaS provides numerous advantages to employees and companies by greatly reducing the time and money spent on tedious tasks such as installing, managing, and upgrading software. This frees up a lot of time for technical staff to spend on more pressing matters and issues within the organization.

**SaaS Characteristics**

There are a few ways to help you determine when SaaS is being utilized:

* Managed from a central location
* Hosted on a remote server
* Accessible over the internet
* Users not responsible for hardware or software updates

**When to Use SaaS**

There are many different situations in which SaaS may be the most beneficial, including:

* If you are a startup or small company that needs to launch ecommerce quickly and don’t have time for server issues or software
* For short-term projects that require collaboration
* If you use applications that aren’t in-demand very often, such as tax software
* For applications that need both web and mobile access

### Examples of SaaS

Google Apps, Dropbox, Salesforce, Cisco WebEx, Concur, GoToMeeting

**PaaS Advantages**

No matter what size of company you may be in, there are numerous advantages for using PaaS:

* Makes the development and deployment of apps simple and cost-effective
* Scalable
* Highly available
* Gives developers the ability to create customized apps without the headache of maintaining the software
* Greatly reduces the amount of coding
* Automates business policy
* Allows easy migration to the hybrid model

**PaaS Characteristics**

PaaS has many characteristics that define it as a cloud service, including:

* It is built on virtualization technology, meaning resources can easily be scaled up or down as your business changes
* Provides a variety of services to assist with the development, testing, and deployment of apps
* Numerous users can access the same development application
* Web services and databases are integrated

**When to Use PaaS**

There are many situations that utilizing PaaS is beneficial or even necessary. If there are multiple developers working on the same development project, or if other vendors must be included as well, PaaS can provide great speed and flexibility to the entire process. PaaS is also beneficial if you wish to be able to create your own customized applications. This cloud service also can greatly reduce costs and it can simplify some challenges that come up if you are rapidly developing or deploying an app.

**Examples of PaaS**

AWS Elastic Beanstalk, Windows Azure, Heroku, Force.com, Google App Engine, Apache Stratos, OpenShift

**IaaS Advantages**

There are many benefits of choosing IaaS, such as that it:

* Is the most flexible cloud computing model
* Easily allows for automated deployment of storage, networking, servers, and processing power
* Hardware can be purchased based on consumption
* Gives clients complete control of their infrastructure
* Resources can be purchased as-needed
* Is highly scalable

**IaaS Characteristics**

Some characteristics to look for when considering IaaS are:

* Resources are available as a service
* The cost varies depending on consumption
* Services are highly scalable
* Typically includes multiple users on a single piece of hardware
* Provides complete control of the infrastructure to organizations
* Dynamic and flexible

**When to Use IaaS**

Just as with SaaS and PaaS, there are specific situations when it is the most advantageous to use IaaS. If you are a startup or a small company, IaaS is a great option so you don’t have to spend the time or money trying to create hardware and software. IaaS is also beneficial for large organizations who wish to have complete control over their applications and infrastructures, but are looking to only purchase what is actually consumed or needed. For rapidly growing companies, IaaS can be a good option as you don’t have to commit to a specific hardware or software as your needs change and evolve. It also helps if you are unsure what demands a new application will need as there is a lot of flexibility to scale up or down as needed.

**Examples of IaaS**

DigitalOcean, Linode, Rackspace, Amazon Web Services (AWS), Cisco Metapod, Microsoft Azure, Google Compute Engine (GCE)

**AWS Tools Available :**

As more businesses go through the [digital transformation](https://www.bmc.com/it-solutions/digital-transformation.html) process of switching to cloud platforms, [AWS](https://www.gartner.com/eventsguide/na/orlando-symposium/whats-ahead-for-aws-and-its-millions-of-customers/) has long established itself as a market leader among cloud service providers such as Azure and Google Cloud Platform. One thing that makes AWS so attractive for large businesses and developers alike is the number of tools available for those who choose to partner with Amazon for their enterprise needs.

Not only is there an abundant suite of cloud management tools accessible through AWS, but most reputable third-party vendors have also made their tools available for the service as well, making it a truly open source solution.

However, with such a huge selection of tools to choose from, it can be a little overwhelming for businesses and developers who need to know which are the best for their specific needs.

The following article will help you take the guesswork out of cloud management with AWS.

## Types of Management Tools

There are generally four types of management tools available to AWS consumers. These are either powered by Amazon or third-party vendors. The categories of tools include the following:

* Provisioning Tools
* Operations Management Tools
* Monitoring and Lodging Tools
* Managed Services and Configuration Tools

### Provisioning Tools

Suites of provisioning tools help you allocate and use resources in the most efficient way possible. Provisioning tools are also applications such as RightScale and CloudFormation (more on this below).

### Operations Management Tools

Operations management tools focus on allowing users to have total control of their Infrastructure-as-a-Service (IaaS) platform. And with confidence that they are meeting compliance and governance standards.

### Monitoring and Lodging Tools

Monitoring and lodging tools are established to make sure everything keeps running as it should. These tools send alerts to managers, automatically respond to certain conditions and monitor log files.

### Managed Services and Configuration Tools

These tools work with managed services like Chef and Puppet Enterprises Servers. They manage the process of configuring the environment and scaling the infrastructure.

## Tools Powered by Amazon

Here is an in-depth look at tools you can get directly from Amazon Web Services.

### AWS CloudFormation

[AWS CloudFormation](https://aws.amazon.com/cloudformation/) provides a common language to provision foundational assets in your cloud instance.

**Benefits:**

* Model your infrastructure from a single-source: a text file
* Standardize the infrastructure for your entire organization in a simplified way
* Provisions can be automated and deployed over and over again without being rebuilt
* Demystify infrastructure by treating it like what it is: code

Using a basic text file, CloudFormation enables you to model and provision each asset required.

### AWS Service Catalog

[AWS Service Catalog](https://aws.amazon.com/servicecatalog/) enables users to oversee a robust index of services primed for use on AWS.

**Benefits:**

* Ensuring your organization is compliant with industry standards
* Help users find IT services to deploy
* Manage IT services from one central point

With services that incorporate everything from virtual machine images, servers, applications and databases, AWS Service Catalog enables you to centrally administer programs. It empowers clients to rapidly deploy IT services they need, on-demand.

### Amazon CloudWatch

[Amazon CloudWatch](https://aws.amazon.com/cloudwatch/) provides monitoring administration services for AWS cloud resources and applications. Users benefit from Amazon CloudWatch ability to gather and track data analytics, screen log records, set alerts and respond to changes in your AWS assets.

**Benefits:**

* Amazon EC2 monitoring
* AWS resource monitoring
* Custom metrics monitoring
* Log monitoring and storage
* View data in visual reports
* React to resource changes
* Set alarms

Amazon CloudWatch can screen AWS assets, for example, Amazon EC2 occurrences, Amazon DynamoDB tables and Amazon RDS DB instances and custom metrics produced by your applications and services.

### AWS Systems Manager

[AWS Systems Manager](https://aws.amazon.com/systems-manager/) gives you full control of the framework on AWS. Systems Manager offers an impactful, easy-to-use UI so you can see operational information from various sources and automate tasks needed for smooth operation.

**Benefits:**

* Ensures security and compliance
* Includes management of hybrid environments
* Full visibility of resource groups and configurations lets you have greater control
* Perfect for automation, easy-to-use
* Detect problems more quickly

With Systems Manager, you can assemble assets by application, monitor operational system info and activate resources.

### AWS CloudTrail

An important operational tool, [AWS CloudTrail](https://aws.amazon.com/cloudtrail/) helps enterprise businesses achieve compliance and track user activity.

**Benefits:**

* User activity is recorded in a secure log
* Compliance audits become easier with pre-stored event logs generated by the system
* Find areas where your system is vulnerable and monitor or fix them
* Security automation

The services offers governance, compliance, operational and risk auditing of your account, to be exact. Cloud Trail provides a comprehensive list of actions taken throughout AWS and aligned services.

### AWS Config

[AWS Config](https://aws.amazon.com/config/) helps users monitor the many configurations of their AWS instance and services. This is a time-consuming process without the right tools. AWS Config offers assistance monitoring, assessing, auditing and evaluating configurations in one place.

**Benefits:**

* Monitor, record and assess system configurations
* Track relationships between resources to simplify change management within an organization
* Troubleshooting capabilities

This tool offers users the option to view changes, configuration history and relationships between asset configurations.

### AWS OpsWorks

[AWS OpsWorks](https://aws.amazon.com/opsworks/) lets you write small instances of code to automate configurations.

**Benefit:**

* Offers application and server management for Puppet, Chef and Stacks; Chef and Puppet are automation platforms that allow you to use code to automate the configurations of your servers.

Using instances of Chef and Puppet designed for AWS, developers can deploy code that keeps their configurations in check. OpsWorks has three offerings, [AWS Opsworks for Chef Automate](https://aws.amazon.com/opsworks/chefautomate/), [AWS OpsWorks for Puppet Enterprise](https://aws.amazon.com/opsworks/puppetenterprise/), and [AWS OpsWorks Stacks](https://aws.amazon.com/opsworks/stacks/).

### AWS Trusted Advisor

[AWS Trusted Advisor](https://aws.amazon.com/trustedadvisor/) increases the overall performance of your AWS environment. It does this by optimizing the instance, recalibrating things that reduce cost, increase security and more.

**Benefits:**

* Full access to a wide range of perks that optimize your AWS instance
* Increased security
* Fine-tuned performance
* Alerts and notifications

Trusted advisor is a provisioning resource that provides on-demand, real-time guidance to AWS users.

## Third-Party Tools

In addition to the tools created by AWS, a number of third-party vendors offer resources for provisioning, ops management, monitoring and configurations.

### RightScale

[RightScale](https://www.rightscale.com/) is a multi-use tool that offers assistance with operations management and provisioning. This tool is also used for monitoring governance and optimizing for cost.

**Benefits:**

* Speed up application delivery
* Increase efficiency
* Global reach
* Optimize investment strategy for the best return
* Risk reduction

This cloud management platform offers users the ability to manage all their clouds from one UI.

### SCALR

Similar to RightScale, [SCALR](https://www.scalr.com/) has a number of functions that are helpful for users in an AWS environment.

**Benefits:**

* Optimize for cost savings
* Gain productivity
* Reduce instances of vendor lock in
* Meets security and industry compliance needs
* Standardize usage across clouds

The aim of this service is to increase productivity, reduce cost, enhance security and prevent common concerns such as vendor-lock in. All the while, offering a flexible environment for users on a public, private or hybrid cloud.

### Hybridfox

[Hybridfox](https://code.google.com/archive/p/hybridfox/) is a popular Chrome add-on that works with a number of IaaS/PaaS providers, including AWS. It can be used with public and private clouds.

* Toggle between different cloud environments easily

Hybridfox perfect for users who have multiple cloud environments because it allows for switching between them seamlessly.

### Cloudability

[Cloudability](https://www.cloudability.com/cloudmgr/) is a full-service cloud suite that offers users migration assistance, configuration management and operations management.

**Benefits:**

* Offers migration assistance
* Cost control optimization
* Provides financial transparency
* Adheres to industry compliance and governance
* Optimized for speed

Cloudability helps to ensure governance and compliance needs are met, while offering a full service suites of services to AWS users.

### Ylastic

[Ylastic](http://ylastic.com/) is a cloud management service that focuses on managing user instances of AWS in an intuitive way and offering data analytic and backup options.

**Benefits:**

* Enhanced governance inventory uses snapshots to ensure compliance
* Track resource configuration
* Collects and monitors data
* Offers analytics in a number of easy-to-read dashboards

Ylastic touches operations management, configuration management, security, compliance and more.

## Final Thoughts: AWS Management Tools

As you can see, there are many tools for enterprise businesses to choose from in this growing market. Finding the right one is largely a matter of understanding your unique specifications and use case requirements. Trial and error is recommended to determine which tools you should use.

While the differences between some of these tools may seem small, something like red-flag resolution and alerts could make all the difference for enterprise business leaders. In many instances, it comes down to personal preference.

Overall, when purchasing any new services or applications, it’s important to first take inventory of the unique needs of your business, then decide on the right course of action.

Apart from choosing the right services, implementing an effective cloud management strategy is also of paramount importance. BMC offers multi-cloud management services simplified into one secure access point. BMC’s services include:

* Migration assistance
* Optimization for performance and cost; and
* Seamless service experience between cloud environments

**AWS & GOOGLE & AZURE COMPARISION:**

An increasing number of enterprises are making the switch to not only the cloud but multiple cloud platforms (multi-cloud) to support employee productivity, foster collaboration and drive business innovation. Indeed, the benefits of the multi-cloud environment are far-reaching and include reduced operational costs, higher accessibility and lower maintenance. All of these advantages are mission-critical to maintaining a competitive advantage in today’s digital world.

As more businesses go through the [digital transformation](https://www.bmc.com/it-solutions/digital-transformation.html) aspect of moving to cloud platforms, three of the biggest names in technology have answered the call. These are Amazon Web Services (AWS), Microsoft Azure and Google.

That being said, the process of choosing the right partners for your computing needs is not always easy. Let’s take a look at the benefits of each of these powerful cloud platforms and how they stack up against each other.

## AWS

With over 12 years of experience under their belt, AWS is one of the oldest in the market. AWS offers a number of features and benefits that make up their Infrastructure-as-a-Service (IaaS) platform. They’ve also more recently shifted to on-demand billing for service used, like their competitors, making AWS a much more attractive option for IT business leaders who continue to feel the pressure of shrinking budgets.

**Features**

There are four categories of offerings that AWS presents to clients. They are:

* Storage and content delivery
* Compute services
* Database; and
* Networking

AWS services are supported by a number of tried and true Amazon offerings that include Active Directory and AWS CloudHSM’s key storage, that enables you to easily generate and use your own encryption keys via the AWS Cloud.

**Benefits**

Choosing a partnership with AWS for your enterprise IaaS needs offers several benefits including the following:

* Widest range of cloud service offerings for enterprise businesses over competitors
* Increases business agility
* EC2 compute engine is customizable
* EC2 features a number of integrations like Elastic Beanstalk for container service and Lambda
* Services a number of high-profile, blue chip customers
* Geared toward open source developers
* Supports commonly used development languages
* Deploy anywhere

If you’re thinking about choosing AWS as your cloud platform, you’d be in good company. AWS has provided services for a number of major clients like Netflix, AirBnB and even the CIA.

## Azure

Microsoft launched Azure several years after Amazon, in 2011 and quickly built a leading reputation. If you’re an enterprise business considering [Azure](https://azure.microsoft.com/) for your cloud computing needs, here’s what you need to know.

**Features**

* Build websites in the cloud, using common programming languages
* Integrations include Windows Server and Linux Virtual Machine
* Migration assistance
* SQL database support
* Focal point is Virtual Machine capabilities supported by tools that include Cloud Services and Resource Manager
* Machine learning

**Benefits**

* Quick to deploy, operate and scale
* Increases business agility
* Has the bandwidth to take your business global
* Visual Studio development environment built in
* Supports integration
* Supported by secure login with Azure Single Sign-On
* Microsoft is no stranger to industry compliance standards
* Deploy anywhere

Microsoft’s culture and position as a long-time leader in technology means Azure, by design, caters to the needs of various industries and their unique specifications.

## Google Cloud

[Google](https://cloud.google.com/) is an obvious trailblazer in internet domination and deployment of web-based services. The Google Cloud Platform is the same age as Azure, but what does it have to offer? Keep reading to find out.

**Features**

Google’s cloud suite of tools includes:

* Google App Engine to build apps within the cloud
* Supports common coding languages
* Open source cloud environment
* Offers robust data analysis
* Google Cloud Bigtable noSQL database supported
* Emphasis on big data tool suites
* Allows users to create single-purpose functions that decrease the need for management
* CloudKMS offers security via encryption
* Cloud storage is a RESTful service for storing and retrieving data
* Migration assistance

**Benefits**

Google Cloud offers the following benefits to customers:

* Quick deployment and access to updates and functionality
* Increases business agility
* Continuous improvement allow Google to update the cloud without disruption to users
* Primed to facilitate collaboration
* Ownership of data
* Scalable
* Deploy anywhere

Google Cloud allows users to retain full ownership of all their data in Google Apps. Should they choose to migrate to another service, they can take their data with them.

## The Scoop: AWS vs Azure vs Google

Microsoft Azure generally has the lowest on-demand pricing while Amazon tends to come in somewhere around the middle among the three major players.

For more information on how to estimate your costs with each of the platforms, see below:

* [AWS Pricing](http://calculator.s3.amazonaws.com/index.html)
* [Azure Pricing](https://azure.microsoft.com/en-us/pricing/calculator/?&WT.srch=1&WT.mc_id=AID631184_SEM_nekFKka9&lnkd=Google_Azure_Brand&gclid=CjwKCAjwzoDXBRBbEiwAGZRIeAsqBfQq4Ur4_cmULhWpfmIwDaCqMyAcDwOzR-UH4tUO7HtE9Ml47hoC8tAQAvD_BwE)
* [Google Cloud Pricing](https://cloud.google.com/products/calculator/)

However, when it comes to choosing one cloud platform over another, cost should not generally be a major determining factor. Overall, the choice must depend on your enterprise needs as you get ready for the next generation cloud.

## AWS: Pros and Cons

AWS is a robust service that offers a lot in the way of benefits. But if you’re thinking about making the shift to AWS for cloud services here’s what you need to know to compare the offerings to the needs of your enterprise organization.

**Pros**:

* AWS offers a complete toolset with a wide breadth of functions available for users
* AWS has been perfecting cloud computing services since 2006
* This partner is seen as the gold standard for reliability, security, configuration option and monitoring
* The AWS cloud ecosystem and product offerings are viewed as a benefit to consumers
* AWS has more compute capacity than most of its competitors by 5x
* Multiple datacenter “regions”
* All major software vendors make their programs available on the AWS platform

**Cons**:

* Depending on the comfort level of enterprise businesses with AWS services, a steep learning curve may exist
* Enterprise level support must be purchased
* AWS was late to get on the on-demand billing for services bandwagon
* AWS doesn’t have a strong hybrid cloud strategy for those businesses that want to keep sensitive data in a private cloud
* The wide catalog of offerings can be overwhelming and difficult to navigate for some users

Despite some high profile outages in years past, today, AWS is widely considered a reliable product for consumers seeking enterprise cloud services today.

## Azure: Pros and Cons

Azure offers a familiar suite of resources that businesses can trust. Here’s the scoop on pros and cons of Microsoft Azure.

**Pros**:

* Strong presence in the enterprise community
* Flexible billing
* Platform-as-a-Service (PaaS) is a strong suit of Microsoft
* Reliability and scalability
* High-availability
* Cost-effective compared to the competition
* Easy to integrate for firms already running a lot of Microsoft software

**Cons**:

* Outages have occurred causing experts at [Gartner](https://www.forbes.com/sites/benkepes/2015/01/16/gartner-shines-a-light-on-the-amazon-versus-microsoft-cloud-race/#449992fd6e80) to suggest a good disaster recovery plan for Azure users
* Functions seem limited when compared against AWS
* Set up to support Windows; if you want to run something else Azure may not be the best option
* Requires enterprise management from customers

As mentioned above, the occurrence of outages isn’t [limited to Microsoft Azure](https://www.gartner.com/webinar/2809818). However, Microsoft has gone a long way to inspire confidence and trust in enterprise businesses due to its reputation for high quality service.

## Google Cloud: Pros and Cons

Google is a powerhouse on the technology scene, and their cloud services reflect that. If you’re thinking about choosing Google Cloud services, consider the following advantages and disadvantages.

**Pros**:

* Great reputation in open-source community
* Modern innovation, well-established in cloud computing
* Flexible pricing model
* Google Cloud is greener than some of the competition and working on getting credentials
* Second-to-none when it comes to building [containers](https://www.bmc.com/blogs/what-is-a-container-containerization-explained/)

**Cons**:

* Supporting large, enterprise projects has not been a focus for Google when compared to companies like Microsoft
* Outages have made some wonder if the product is reliable
* Range of functions lacks innovation, Google should focus on catching up with Microsoft and AWS

Google is a forward-thinking company with a lot to offer enterprise businesses. However, they have their hands in so many small project innovations, which means limited options for larger companies.

## Final Thoughts

Today, the clear market leader in functionality depth and breadth is AWS. This is partly because they have the maturity in the market and the experience to deliver. But competitors like Microsoft and Google aren’t too far behind.

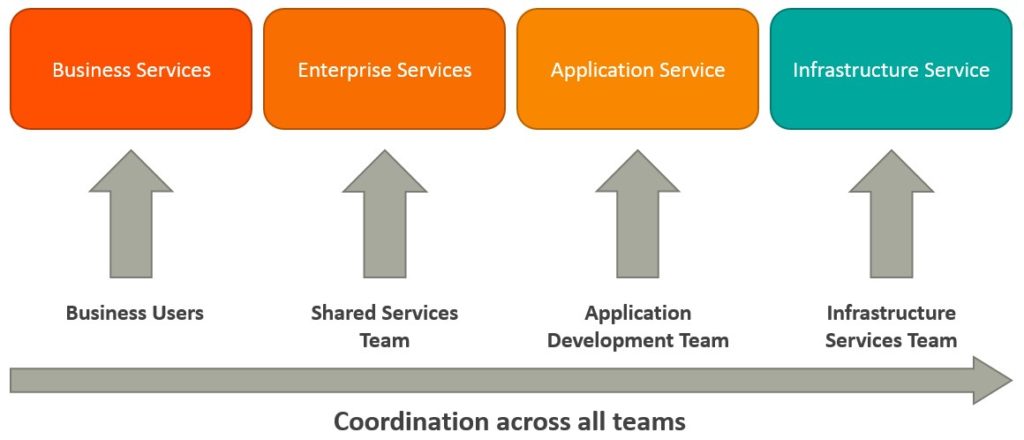
In particular, Microsoft continues to gain headway on AWS. This is especially true because they are a trusted name in enterprise computing with a reliable model for those that already use Microsoft products or languages across the enterprise organization. Google also has a lot to offer in terms of innovation, and might be a good fit for those in the open-source, cloud community that focus on smaller projects.

# Microservices vs SOA: What’s the Difference?

**33**

[Microservices Architecture (MSA)](https://www.bmc.com/blogs/microservices-architecture-introduction-microservices/) and [Service-Oriented Architecture (SOA)](https://www.bmc.com/blogs/service-oriented-architecture-overview/) both rely on services as the main component. But they vary greatly in terms of service characteristics.

SOA defines 4 basic service types as depicted below:



**Business services** are coarse-grained services that define core business operations. They are usually represented through XML, Web Services Definition Language (WSDL) or Business Process Execution Language (BPEL).

**Enterprise services** implement the functionality defined by business services. They rely on application services and infrastructure services to fulfill business requests.

**Application services** are fine-grained services that are bound to a specific application context. These services can be invoked directly through a dedicated user interface.

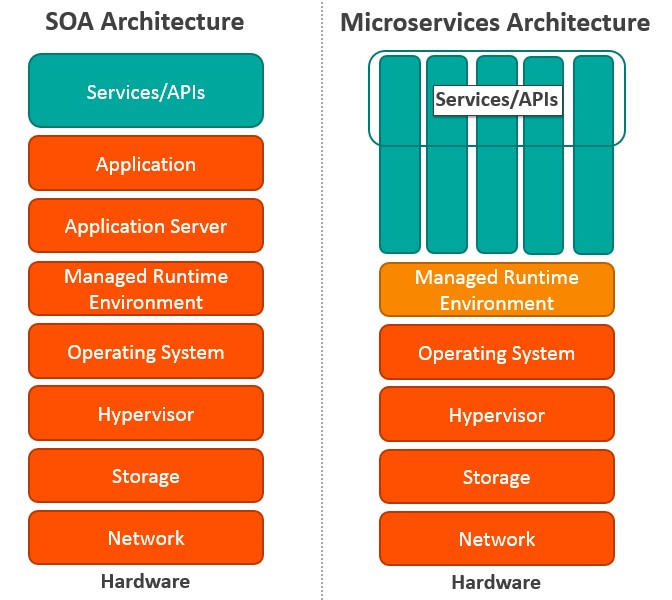
**Infrastructure services** implement non-functional tasks such as authentication, auditing, security, and logging. They can be invoked from either application services or enterprise services.

MSAs have limited service taxonomy. The architecture consists of 2 service types as depicted below.



**Functional services** support specific business operations. The services are accessed externally and they are usually not shared with any other service. As in SOA, infrastructure services implement tasks such as auditing, security, and logging. The services are not exposed to the outside world and they are available internally.

## Key differences between SOA and MSA



| **SOA** | **MSA** |
| --- | --- |
| Built on the idea of “share-as-much-as-possible” architecture approach | Built on the idea of “share-as-little-as-possible” architecture approach |
| More importance on business functionality reuse | More importance on the concept of “bounded context” |
| Common governance and standards | Relaxed governance, with more focus on people collaboration and freedom of choice |
| Uses enterprise service bus (ESB) for communication | Uses less elaborate and simple messaging system |
| Supports multiple message protocols | Uses lightweight protocols such as HTTP/REST & AMQP |
| Common platform for all services deployed to it | Application Servers not really used. Platforms such as Node.JS could be used |
| Multi-threaded with more overheads to handle I/O | Single-threaded usually with use of Event Loop features for non-locking I/O handling |
| Use of containers (Dockers, Linux Containers) less popular | Containers work very well in MSA |
| Maximizes application service reusability | More focused on decoupling |
| Uses traditional relational databases more often | Uses modern, non-relational databases |
| A systematic change requires modifying the monolith | A systematic change is to create a new service |
| DevOps / Continuous Delivery is becoming popular, but not yet mainstream | Strong focus on DevOps / Continuous Delivery |

## Let’s explore the differences in more detail:

* **Coordination**: In SOA, you need to coordinate with multiple groups to create business requests. But there is little or no coordination among services in MSA. If coordination is needed among service owners, it is done through small application development teams, and services can be quickly developed, tested and deployed.
* **Service granularity**: The prefix “micro” in Microservices refers to the granularity of the internal components. Service components within MSA are generally single purpose services that do one thing really well. Services usually include much more business functionality in SOA, and they are often implemented as complete subsystems.
* **Component sharing**: SOA enhances component sharing, whereas MSA tries to minimize on sharing through “bounded context.” A bounded context refers to the coupling of a component and its data as a single unit with minimal dependencies. As SOA relies on multiple services to fulfill a business request, systems built on SOA are likely to be slower than MSA.
* **Middleware vs API layer**: The messaging middleware in SOA offers a host of additional capabilities not found in MSA, including mediation and routing, message enhancement, message and protocol transformation. MSA has an API layer between services and service consumers.
* **Remote services**: SOA architectures rely on messaging (AMQP, MSMQ) and SOAP as primary remote access protocols. Most MSAs rely on two protocols – REST and simple messaging (JMS, MSMQ), and the protocol found in MSA is usually homogeneous.
* **Heterogeneous interoperability**: SOA promotes the propagation of multiple heterogeneous protocols through its messaging middleware component. MSA attempts to simplify the architecture pattern by reducing the number of choices for integration. If you would like to integrate several systems using different protocols in heterogeneous environment, you need to consider SOA. If all your services could be exposed and accessed through the same remote access protocol, then MSA is a better option.
* **Contract decoupling**: Contract decoupling is the holy grail of abstraction. It offers the greatest degree of decoupling between services and consumers. It is one of the fundamental capabilities offered within SOA. But MSA doesn’t support contract decoupling.

Microservices are not invented. Enterprises such as Amazon, Netflix, and eBay used the divide and conquer strategy to functionally partition their monolithic applications into smaller units, and resolved many issues. Following the success of these companies, many other companies started adopting this as a common pattern to refactor their applications. Eventually the pattern was termed as Microservices Architecture. Nothing radically new has been introduced in MSA. MSA is the logical evolution of SOA and supports modern business use cases.

SOA is better suited for large and complex business application environments that require integration with many heterogeneous applications. However, workflow based applications that have a well defined processing flow are a bit difficult to implement using SOA patterns. Small applications are also not a good fit for SOA as they don’t need messaging middleware component. The MSA pattern is well suited for smaller and well partitioned web based systems. The lack of messaging middleware is one of the factors that makes it unfit for complex environments.

If you are developing an application, then MSA gives you greater control as a developer. If you are trying to orchestrate a number of business processes, SOA probably provides a better set of tools.

Also in the early stages of your business, you might find that MSA is a good choice. As the business grows, you may need capabilities such as complex request transformation and heterogeneous systems integration. In such situations, you may likely turn to SOA pattern to replace MSA.

Both SOA and MSA are the same set of standards used at different layers of an enterprise. The existence of MSA comes down to the success of SOA pattern. Hence, MSA pattern is a subset of SOA. Here the main focus is on the runtime autonomy of each service.

# What is Serverless Architecture? Serverless Architecture Explained

Many experts believe that serverless computing is the future, as mobile and Internet of Things (IoT) applications continue to fuel demand for serverless architecture, coupled with the growing need to integrate cloud applications with mobile and desktop apps.

Serverless architecture refers to the implementation of serverless code to create a number of design patterns that benefit businesses. It forms the foundation of serverless computing.

In this article, we will examine all things serverless architecture, including:

* Types of serverless architecture software
* Existing frameworks of serverless architecture
* Design patterns

## Types of Serverless Architecture Software

There are three primary services offered via software developed with serverless architecture.

### Function-as-a-Service

In the realm of pre-packaged services, Function-as-a-Service is also sometimes known as Framework-as-a-Service or FaaS, falls in between Software-as-a-Service and Platform-as-a-Service.

Think of FaaS as a ready-to-implement framework that can be easily tailored to the needs of an enterprise company. For further clarification, SaaS is ready to use out of the box while FaaS is not. However, FaaS does not require the resources to implement that you would need if you were using PaaS.

FaaS can be delivered in customizable templates, for instance, by industry vertical. FaaS uses containers to prime for rapid deployment of applications on all platforms. For instance, developers can stack containers for scalability or write one container for iOS development and another for desktop applications.

Consumers purchase FaaS from third-party vendors who handle server management. They are then charged for actual runtimes instead of pre-allocated units. Companies who use FaaS benefit from improved efficiency as fewer resources are spent on rapid development of applications.

### Backend-as-a-Service

Similar to FaaS, Backend-as-a-Service (or BaaS) is another serverless technology. Some will contend that BaaS takes it a step further as a NoOps offering. NoOps essentially refers to infrastructure that has been automated to the point that in-house developers have no hand in its operation.

Either way, here is an easy way to look at BaaS: imagine your enterprise organization is developing a mobile app to connect employees to important information on the go. You might develop the basic application framework in-house, and outsource the functionality. This includes backend processes like access cloud storage, syncing.and social collaboration.

A company’s ability to offer backend services that link a mobile application to the cloud is referred to as BaaS.

### Database

Database serverless frameworks access and automate your database functions. These are functions that both write and read from a database, as well as provide a response.

Serverless database frameworks offer companies room to grow globally, as multiple applications can be developed by region. Yet, they all run from one location powered by FaaS technology.

All [types of architecture](https://www.pwc.com/us/en/financial-services/publications/serverless-computing.html) benefit from the unique advantages of serverless computing, which can be summed up below:

* Lower operational cost
* Increased efficiency
* Better allocation of resources

## Notable Frameworks

As companies prepare for the future, they can expect to see a landscape of technology forever changed by serverless architecture. This is evidenced by the innovative frameworks that are already competing for consumer attention. Notable frameworks include:

### AWS Lambda

The most well-known, quintessential serverless framework is AWS Lambda, powered by Amazon. This framework allows developers to reap the benefits of development without the cost of a physical server infrastructure. Additionally, it is one of the first of its kind to gain mainstream popularity.

AWS Lambda is a full administrative solution. Amazon runs your code snippets in response to events and only charges for times when the code is being used. They handle all the backend and administrative functions.

Lambda supports:

* Node.js
* Java
* C#
* Python

### Google Cloud Functions

Google Cloud Functions offers serverless computing on Google’s open infrastructure. The platform supports Javascript and executes in Node.js. So, it offers a familiar environment for developers to code.

The platform benefits include the following:

* Focus on microservices development
* Allows users to connect cloud services and applications (BaaS)
* Rapid and efficient development environment
* Mobile ready for developers to drop in code

### IBM Cloud Functions

IBM Cloud Functions is a FaaS model based on its predecessor Apache OpenWhisk. It’s described as a lightweight serverless architecture model that allows developers to rapidly code on-demand, meaning they are only charged for what they use.

## Four Design Patterns

One of the big benefits of serverless architecture includes freeing up resources to focus on what the software is supposed to do and not what is needed to make this happen. As a result, frequently used design patterns have emerged.

At the 2017 [APIDays Conference](http://au.apidays.io/" \t "_blank), research on AWS Lambda revealed four general design patterns for server architecture that are commonly deployed. These are described below.

### Event-Driven Applications

If you do your research on serverless architecture, you’ll see the term event-driven a lot. It basically means that runtimes are prompted as a response to events. This simplistic architecture is among the most commonly used.

This form of architecture lends itself to DevOps developers who focus on writing patches that integrate two systems.

### Web Apps

In the use case of serverless web apps, processes run to determine who the user is and what information should be served to that user inside the application. In the case of Lambda, processing gets initiated in the API gateway to replace and enhance static content with custom content based on user details stored in a dynamic database.

### Mobile and IoT Apps

These cases work similar to web apps in this context. User authentication is deployed to determine who the user is, and custom content is served up based on user profile.

### Application Ecosystem

An application ecosystem for serverless applications is one where workflows and applications are created in a serverless environment. These unique ecosystems are powered by functions of both third-party providers and AWS. An ecosystem is created when the functions of a given serverless architecture are integrated with other functions that allow for customization of processes.

## The Future is Serverless

Serverless architecture is the way of the future for many types of simple applications. It’s an ideal approach for some enterprise businesses because it offers the following benefits:

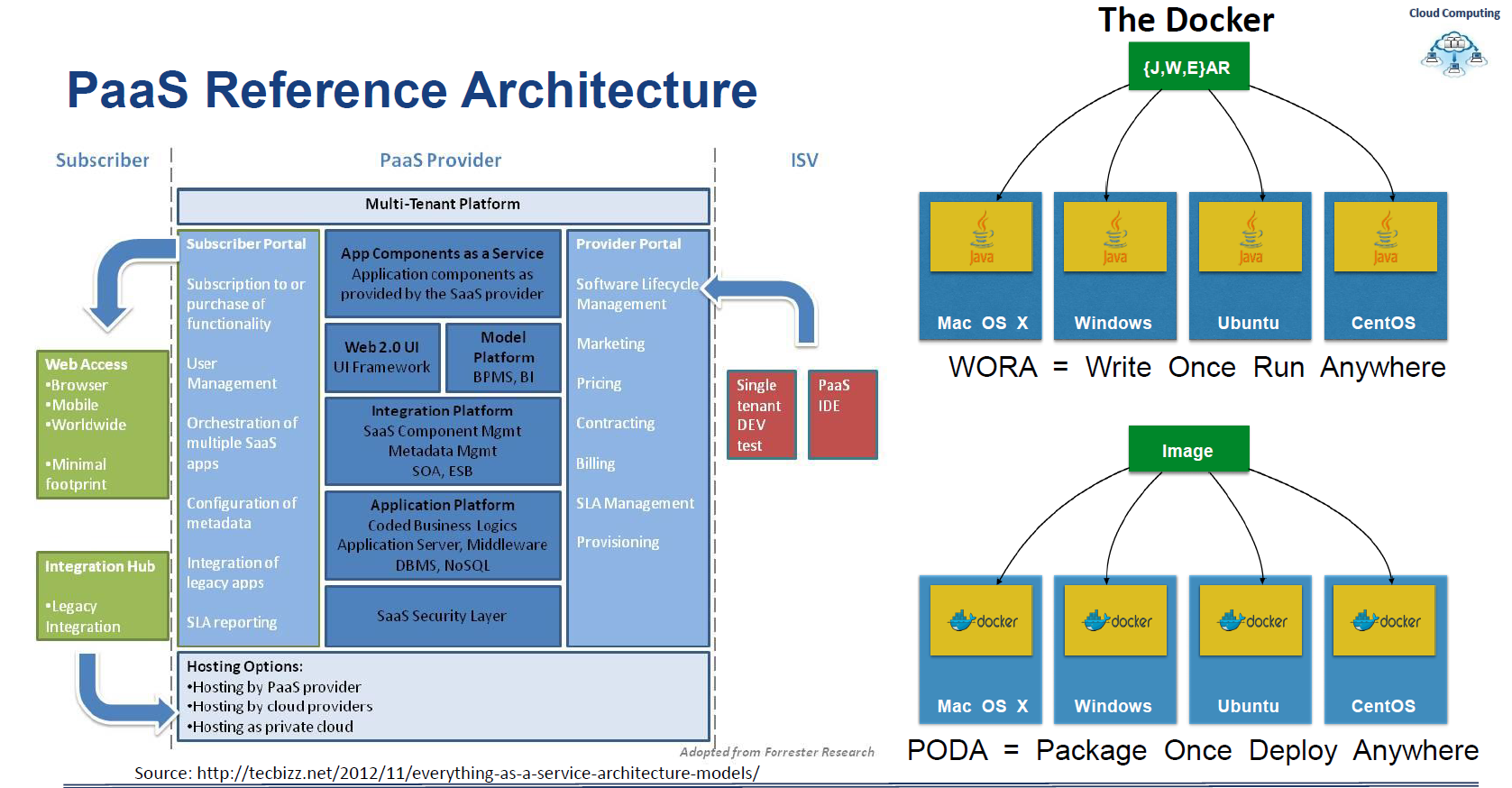
* Efficient use of resources
* Rapid deployment
* Cost-effective solutions
* Focus on coding
* Familiar programming environment and languages supported
* Increased scalability

For these reasons, it’s easy to see why many companies are making the switch. At the same time, it is important to understand that serverless architecture does have drawbacks that need to be considered before making any changes to your business model.

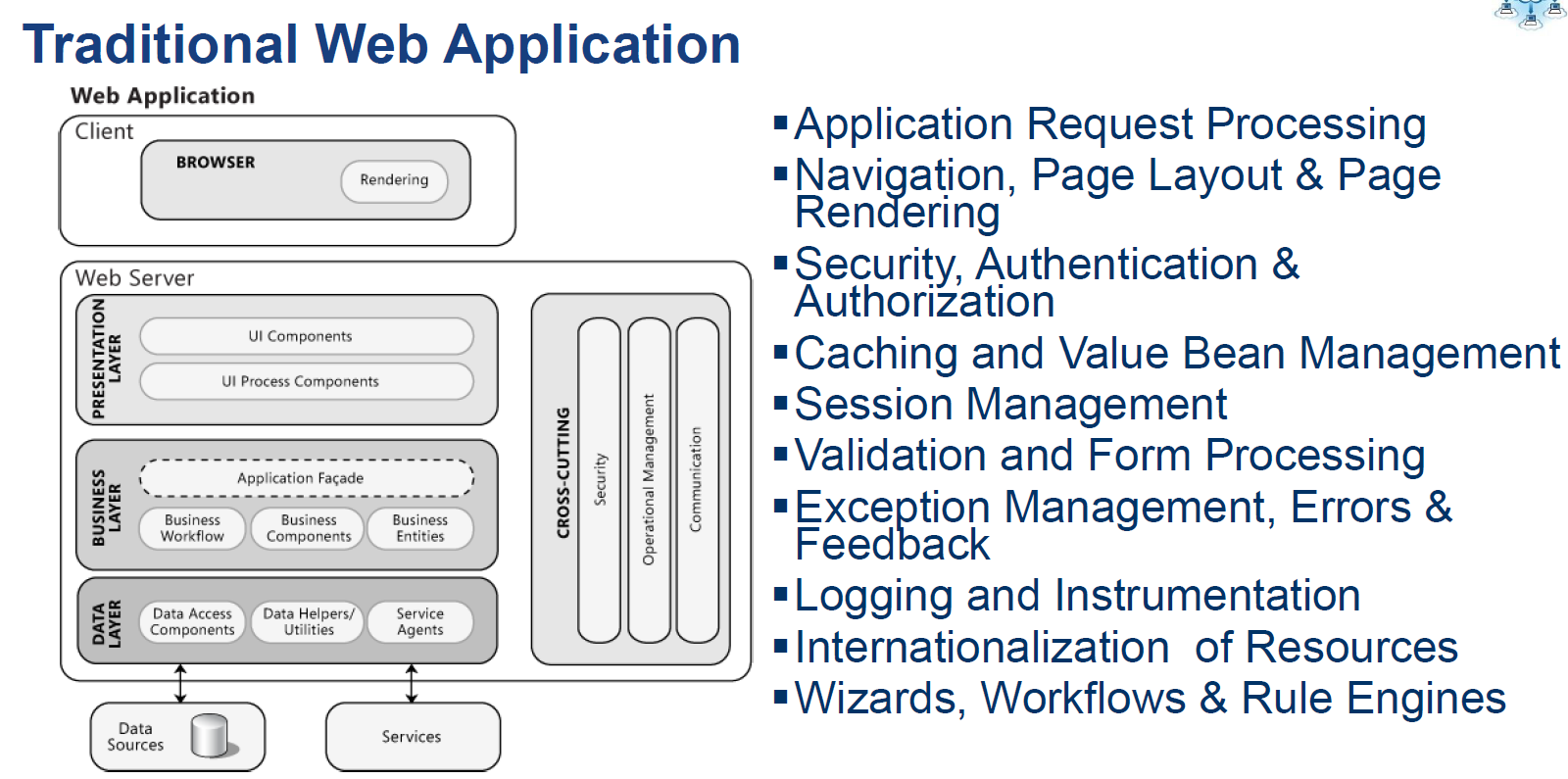
When you use a third party API the following concerns emerge:

* **Vendor control issues:** Developers give up control, especially in the case of BaaS systems.
* **Security concerns:** These arise as a result of leaving security in the hands of the vendor.
* **Contract lock-in:** This occurs when consumers are unable to break contracts with their vendors.
* **Dependent relationship:** Consumers are dependent on vendors for debugging and monitoring.
* **Limitations:** Limits exist on processes to sidestep the pitfalls of overcomplicated architecture.
* **Response Latency:** Because consumers are only billed for runtime, code is powered down between requests. Response times can vary as a result.

Because of the above-mentioned challenges, serverless architecture is often not typically ideal for high-performing, complex application builds.



**Enterprise Architecture :**



**Difference Between Traditional Apps & Cloud Apps:**

|  |  |
| --- | --- |
| **Sticky Session** | **Shared Session** |
| **VM Centric** | **App Centric (Include Docker)** |

**Microservice**

Microservices have become a popular architectural style for building cloud applications that are resilient, highly scalable, and able to evolve quickly. To be more than just a buzzword, however, microservices require a different approach to designing and building applications.

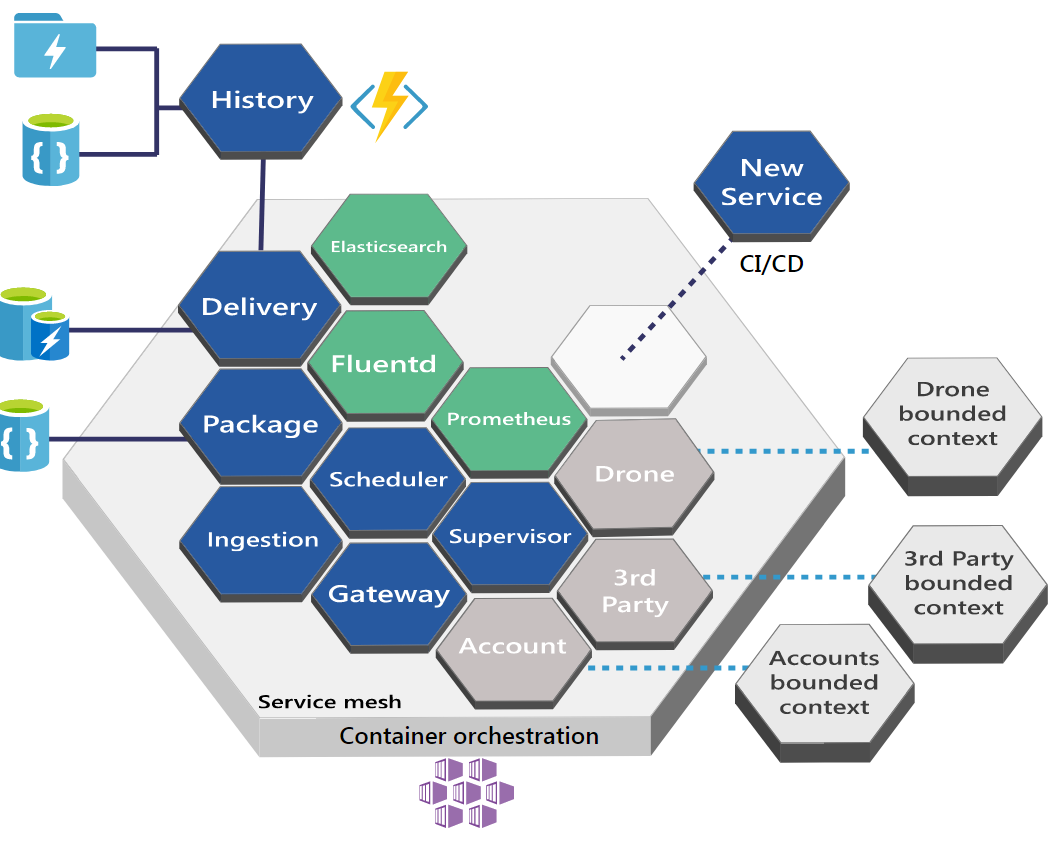
In this set of articles, we explore how to build and run a microservices architecture on Azure, using Kubernetes as a container orchestrator. Future articles will include Service Fabric. Topics include:

* Using Domain Driven Design (DDD) to design a microservices architecture.
* Choosing the right Azure technologies for compute, storage, messaging, and other elements of the design.
* Understanding microservices design patterns.
* Designing for resiliency, scalability, and performance.
* Building a CI/CD pipeline.

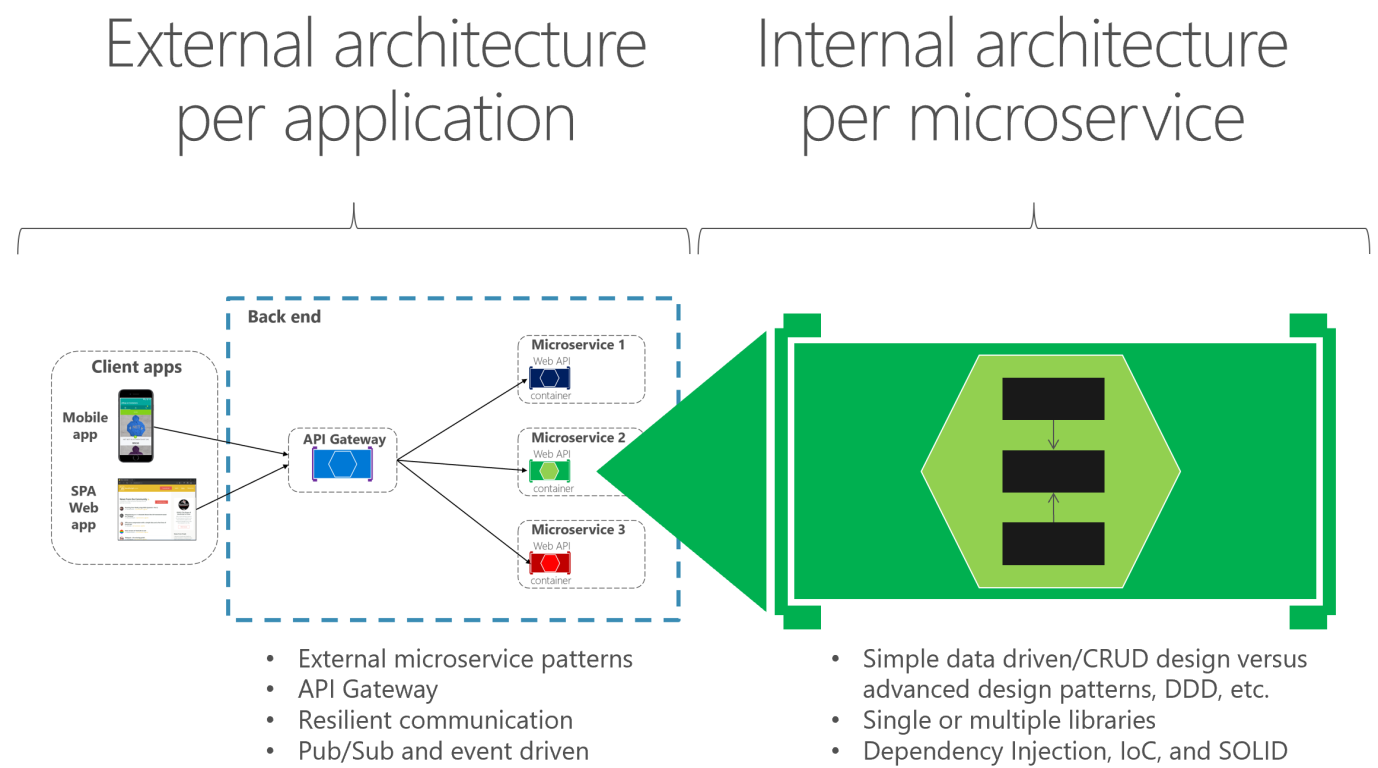
Throughout, we focus on an end-to-end scenario for a drone delivery service that lets customers schedule packages to be picked up and delivered via drone. A reference implementation for this project is available on GitHub.

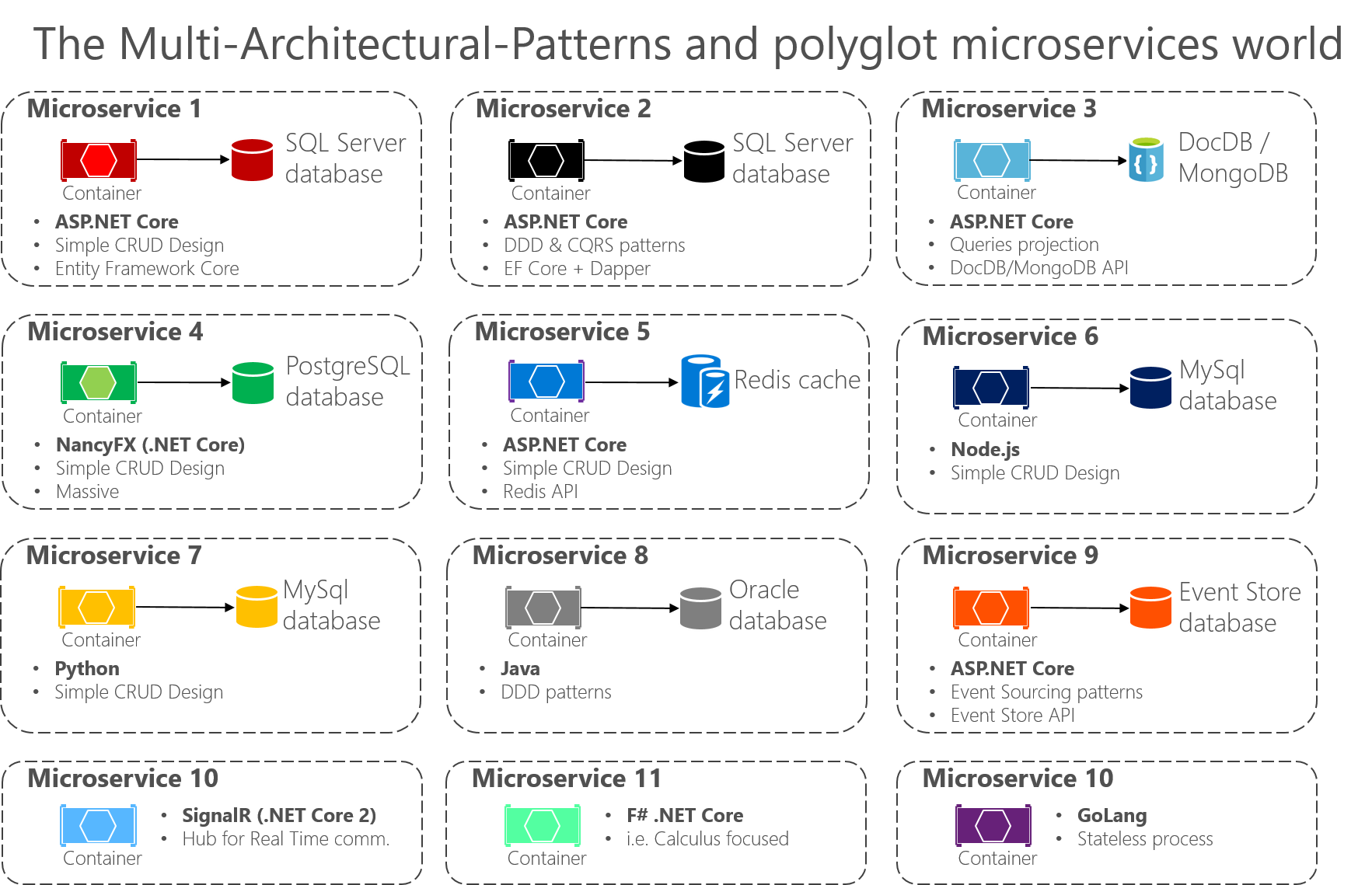
The reference implementation includes a number of different Azure and open source technologies:

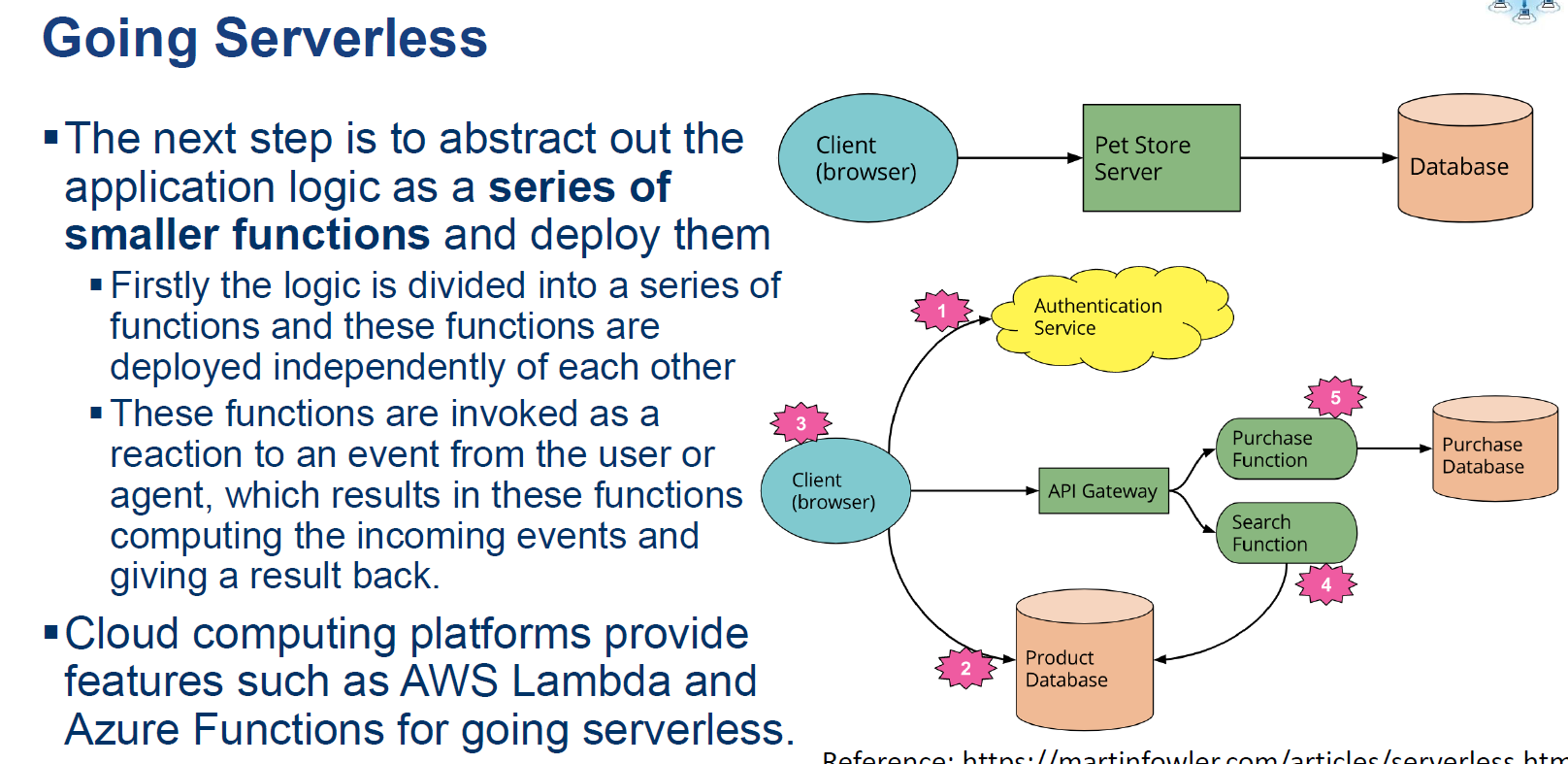
* Azure Container Service (Kubernetes) to run frontend and backend services.
* Azure Functions to run event driven services.
* Linkerd to manage inter-service communication.
* Prometheus to monitor system/application metrics.
* Fluentd and Elasticsearch to monitor application logs.
* Cosmos DB, Azure Data Lake Store, and Azure Redis Cache to store different types of data.

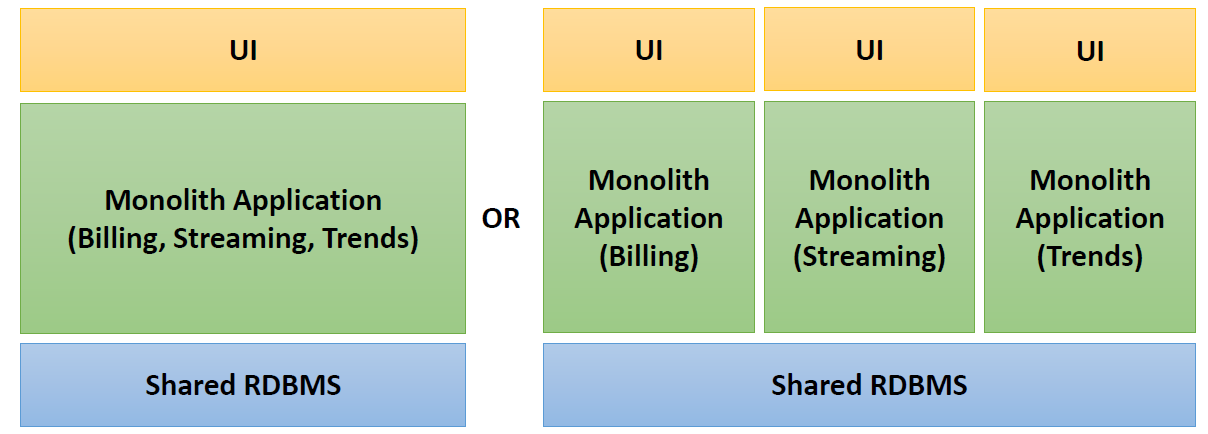


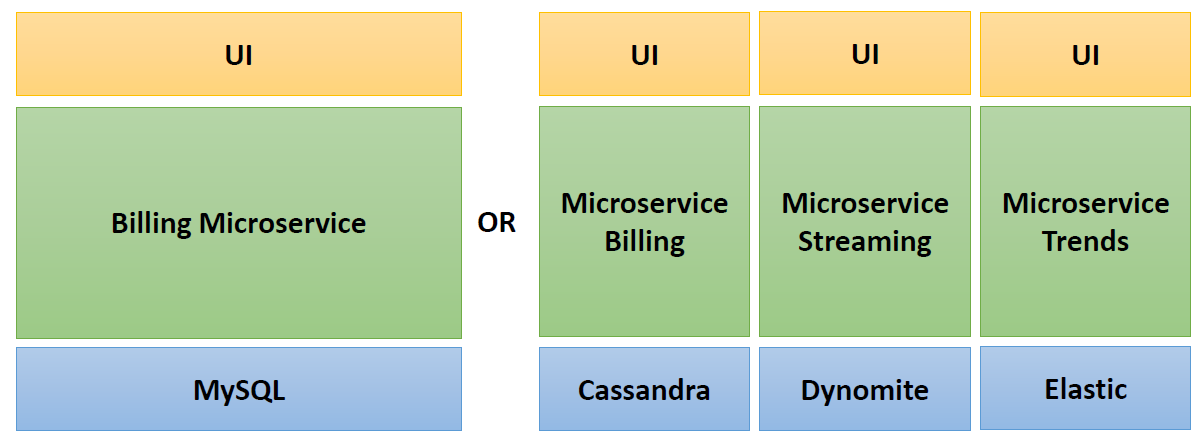
****

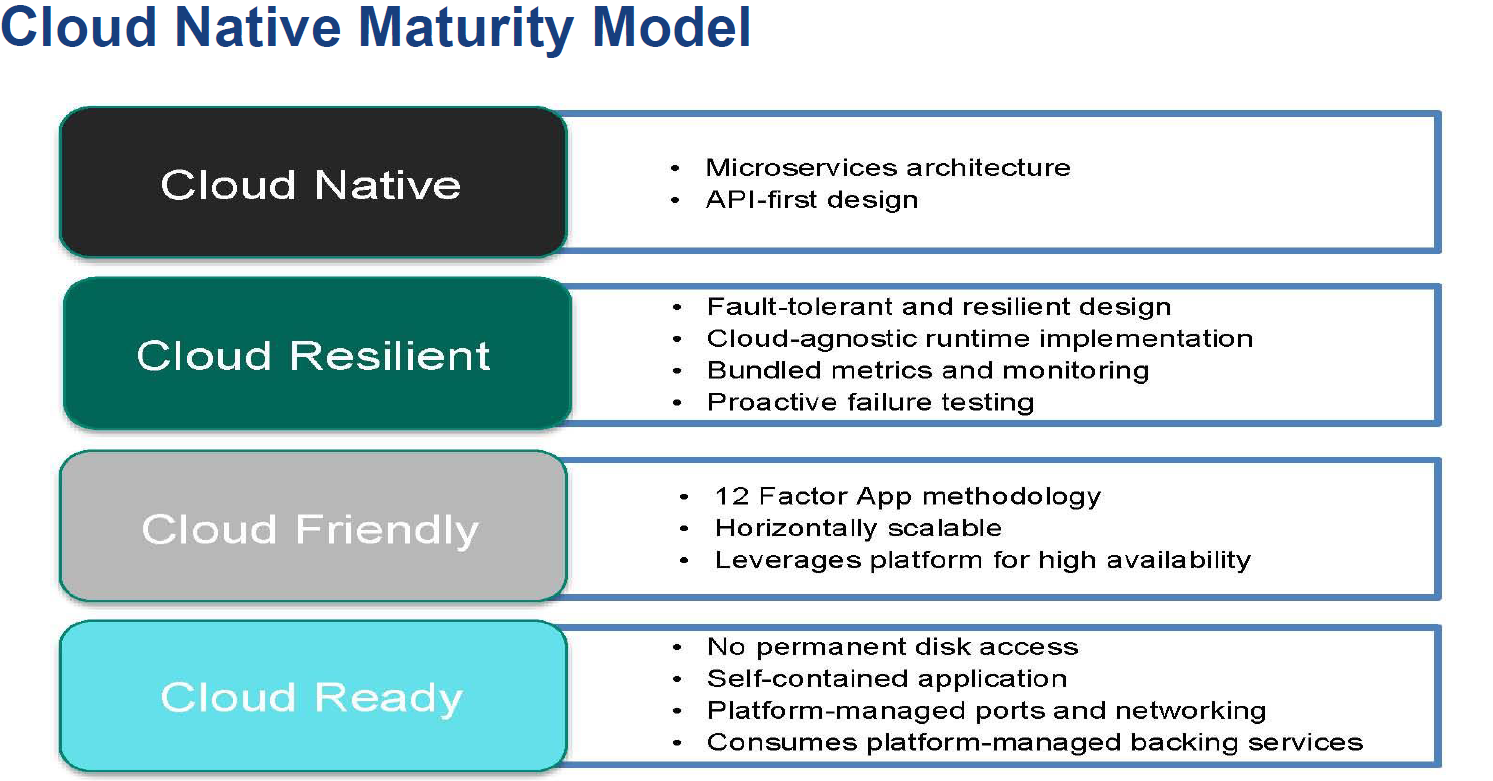
****

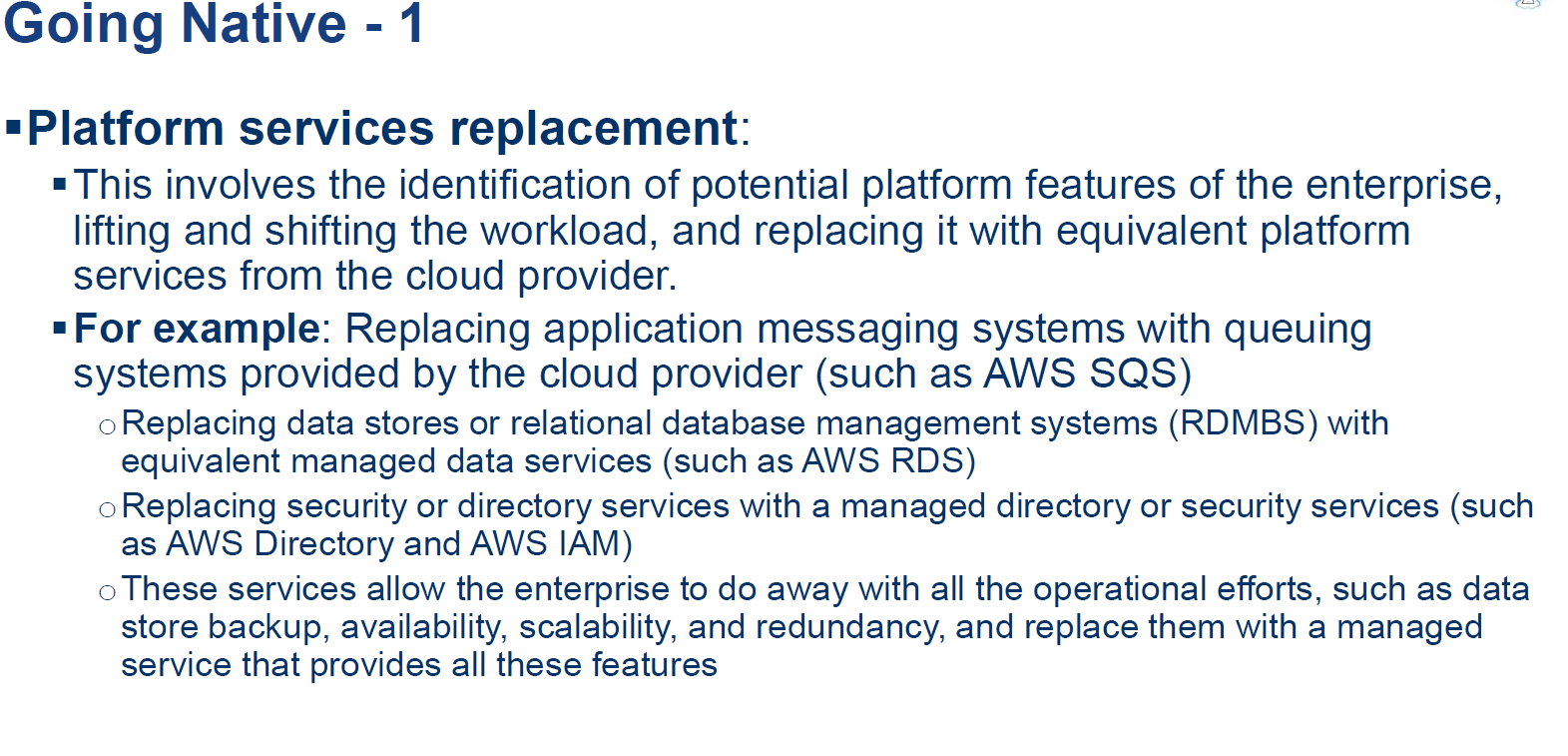
****

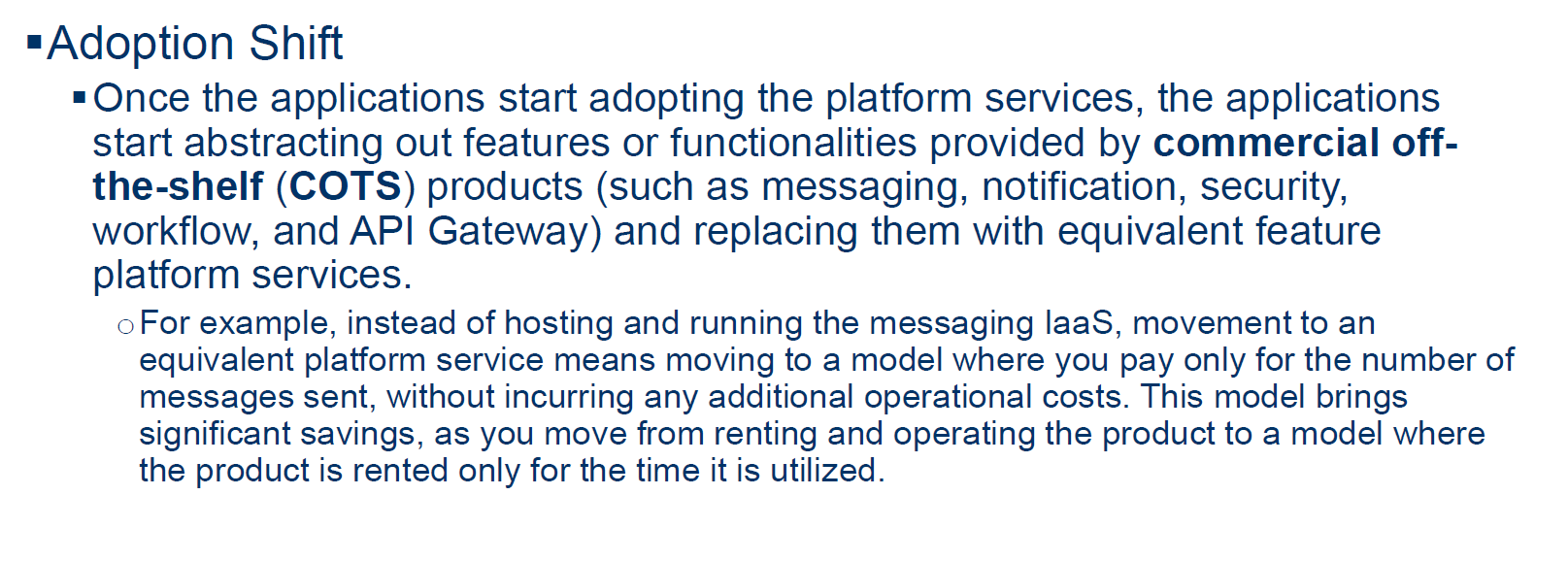


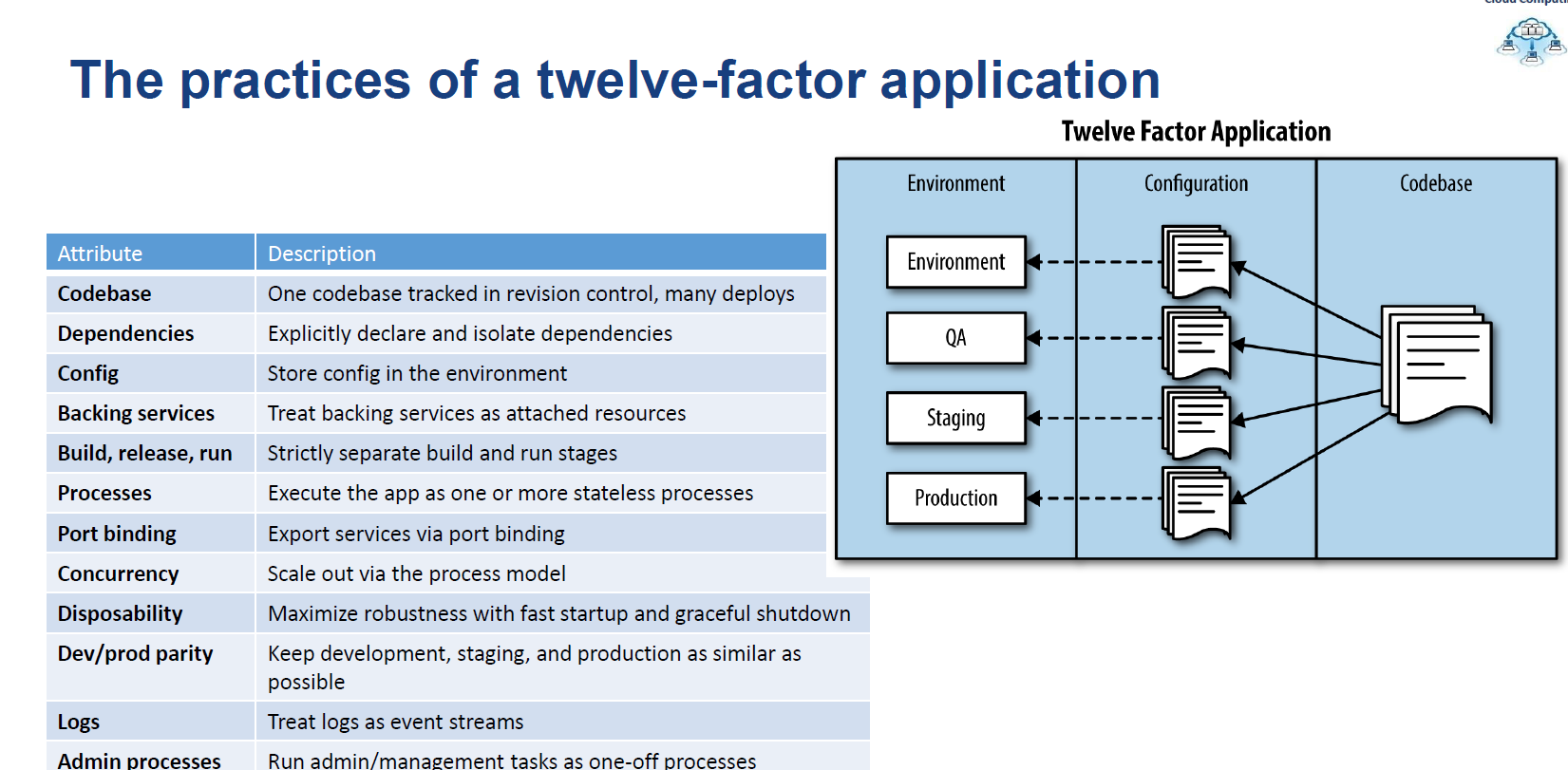


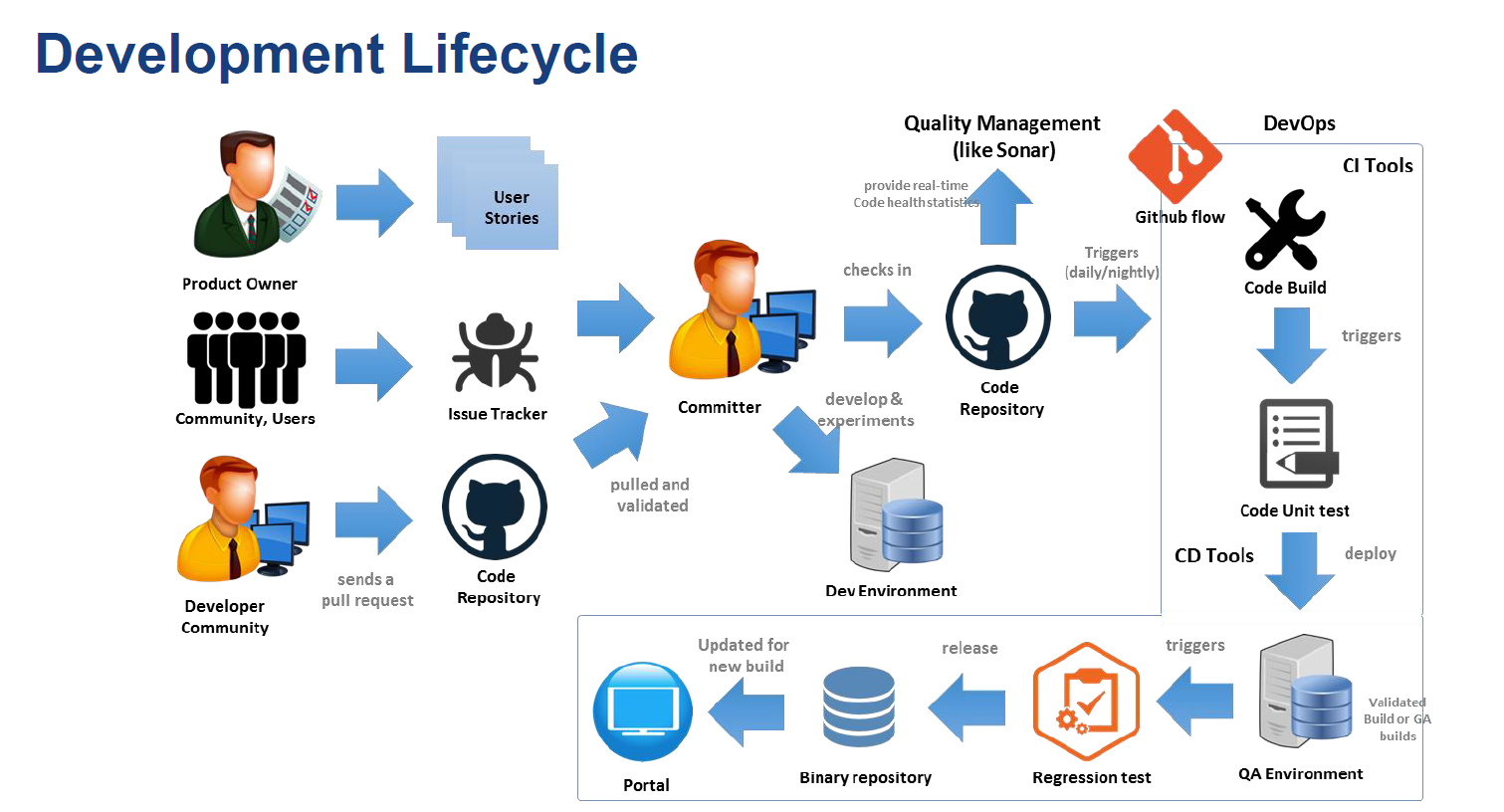












**When we need to scale out application layer how we can achieve without affecting availability.**

# Designing resilient applications for Azure

In a distributed system, failures will happen. Hardware can fail. The network can have transient failures. Rarely, an entire service or region may experience a disruption, but even those must be planned for.

Building a reliable application in the cloud is different than building a reliable application in an enterprise setting. While historically you may have purchased higher-end hardware to scale up, in a cloud environment you must scale out instead of scaling up. Costs for cloud environments are kept low through the use of commodity hardware. Instead of focusing on preventing failures and optimizing "mean time between failures," in this new environment the focus shifts to "mean time to restore." The goal is to minimize the effect of a failure.

This article provides an overview of how to build resilient applications in Microsoft Azure. It starts with a definition of the term resiliency and related concepts. Then it describes a process for achieving resiliency, using a structured approach over the lifetime of an application, from design and implementation to deployment and operations.

## What is resiliency?

**Resiliency** is the ability of a system to recover from failures and continue to function. It's not about avoiding failures, but responding to failures in a way that avoids downtime or data loss. The goal of resiliency is to return the application to a fully functioning state following a failure.

Two important aspects of resiliency are high availability and disaster recovery.

* **High availability** (HA) is the ability of the application to continue running in a healthy state, without significant downtime. By "healthy state," we mean the application is responsive, and users can connect to the application and interact with it.
* **Disaster recovery** (DR) is the ability to recover from rare but major incidents: non-transient, wide-scale failures, such as service disruption that affects an entire region. Disaster recovery includes data backup and archiving, and may include manual intervention, such as restoring a database from backup.

One way to think about HA versus DR is that DR starts when the impact of a fault exceeds the ability of the HA design to handle it.

When you design resiliency, you must understand your availability requirements. How much downtime is acceptable? This is partly a function of cost. How much will potential downtime cost your business? How much should you invest in making the application highly available? You also have to define what it means for the application to be available. For example, is the application "down" if a customer can submit an order but the system cannot process it within the normal timeframe? Also consider the probability of a particular type of outage occurring, and whether a mitigation strategy is cost-effective.

Another common term is **business continuity** (BC), which is the ability to perform essential business functions during and after adverse conditions, such as a natural disaster or a downed service. BC covers the entire operation of the business, including physical facilities, people, communications, transportation, and IT. This article focuses on cloud applications, but resilience planning must be done in the context of overall BC requirements.

**Data backup** is a critical part of DR. If the stateless components of an application fail, you can always redeploy them. But if data is lost, the system can't return to a stable state. Data must be backed up, ideally in a different region in case of a region-wide disaster.

Backup is distinct from **data replication**. Data replication involves copying data in near-real-time, so that the system can fail over quickly to a replica. Many databases systems support replication; for example, SQL Server supports SQL Server Always On Availability Groups. Data replication can reduce how long it takes to recover from an outage, by ensuring that a replica of the data is always standing by. However, data replication won't protect against human error. If data gets corrupted because of human error, the corrupted data just gets copied to the replicas. Therefore, you still need to include long-term backup in your DR strategy.

## Process to achieve resiliency

Resiliency is not an add-on. It must be designed into the system and put into operational practice. Here is a general model to follow:

1. **Define** your availability requirements, based on business needs.
2. **Design** the application for resiliency. Start with an architecture that follows proven practices, and then identify the possible failure points in that architecture.
3. **Implement** strategies to detect and recover from failures.
4. **Test** the implementation by simulating faults and triggering forced failovers.
5. **Deploy** the application into production using a reliable, repeatable process.
6. **Monitor** the application to detect failures. By monitoring the system, you can gauge the health of the application and respond to incidents if necessary.
7. **Respond** if there are incidents that require manual interventions.

In the remainder of this article, we discuss each of these steps in more detail.

## Defining your resiliency requirements

Resiliency planning starts with business requirements. Here are some approaches for thinking about resiliency in those terms.

### Decompose by workload

Many cloud solutions consist of multiple application workloads. The term "workload" in this context means a discrete capability or computing task, which can be logically separated from other tasks, in terms of business logic and data storage requirements. For example, an e-commerce app might include the following workloads:

* Browse and search a product catalog.
* Create and track orders.
* View recommendations.

These workloads might have different requirements for availability, scalability, data consistency, disaster recovery, and so forth. Again, these are business decisions.

Also consider usage patterns. Are there certain critical periods when the system must be available? For example, a tax-filing service can't go down right before the filing deadline, a video streaming service must stay up during a big sports event, and so on. During the critical periods, you might have redundant deployments across several regions, so the application could fail over if one region failed. However, a multi-region deployment is more expensive, so during less critical times, you might run the application in a single region.

### RTO and RPO

Two important metrics to consider are the recovery time objective and recovery point objective.

* **Recovery time objective** (RTO) is the maximum acceptable time that an application can be unavailable after an incident. If your RTO is 90 minutes, you must be able to restore the application to a running state within 90 minutes from the start of a disaster. If you have a very low RTO, you might keep a second deployment continually running on standby, to protect against a regional outage.
* **Recovery point objective** (RPO) is the maximum duration of data loss that is acceptable during a disaster. For example, if you store data in a single database, with no replication to other databases, and perform hourly backups, you could lose up to an hour of data.

RTO and RPO are business requirements. Conducting a risk assessment can help you define the application's RTO and RPO. Another common metric is **mean time to recover** (MTTR), which is the average time that it takes to restore the application after a failure. MTTR is an empirical fact about a system. If MTTR exceeds the RTO, then a failure in the system will cause an unacceptable business disruption, because it won't be possible to restore the system within the defined RTO.

### SLAs

In Azure, the [Service Level Agreement](https://azure.microsoft.com/support/legal/sla/) (SLA) describes Microsoft’s commitments for uptime and connectivity. If the SLA for a particular service is 99.9%, it means you should expect the service to be available 99.9% of the time.

**Note**

The Azure SLA also includes provisions for obtaining a service credit if the SLA is not met, along with specific definitions of "availability" for each service. That aspect of the SLA acts as an enforcement policy.

You should define your own target SLAs for each workload in your solution. An SLA makes it possible to evaluate whether the architecture meets the business requirements. For example, if a workload requires 99.99% uptime, but depends on a service with a 99.9% SLA, that service cannot be a single-point of failure in the system. One remedy is to have a fallback path in case the service fails, or take other measures to recover from a failure in that service.

The following table shows the potential cumulative downtime for various SLA levels.

| **SLA** | **Downtime per week** | **Downtime per month** | **Downtime per year** |
| --- | --- | --- | --- |
| 99% | 1.68 hours | 7.2 hours | 3.65 days |
| 99.9% | 10.1 minutes | 43.2 minutes | 8.76 hours |
| 99.95% | 5 minutes | 21.6 minutes | 4.38 hours |
| 99.99% | 1.01 minutes | 4.32 minutes | 52.56 minutes |
| 99.999% | 6 seconds | 25.9 seconds | 5.26 minutes |

Of course, higher availability is better, everything else being equal. But as you strive for more 9s, the cost and complexity to achieve that level of availability grows. An uptime of 99.99% translates to about 5 minutes of total downtime per month. Is it worth the additional complexity and cost to reach five 9s? The answer depends on the business requirements.

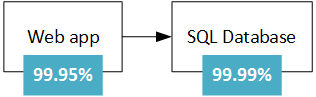
Here are some other considerations when defining an SLA:

* To achieve four 9's (99.99%), you probably can't rely on manual intervention to recover from failures. The application must be self-diagnosing and self-healing.
* Beyond four 9's, it is challenging to detect outages quickly enough to meet the SLA.
* Think about the time window that your SLA is measured against. The smaller the window, the tighter the tolerances. It probably doesn't make sense to define your SLA in terms of hourly or daily uptime.

### Composite SLAs

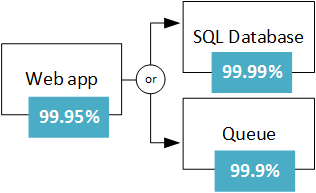
Consider an App Service web app that writes to Azure SQL Database. At the time of this writing, these Azure services have the following SLAs:

* App Service Web Apps = 99.95%
* SQL Database = 99.99%



What is the maximum downtime you would expect for this application? If either service fails, the whole application fails. In general, the probability of each service failing is independent, so the composite SLA for this application is 99.95% × 99.99% = 99.94%. That's lower than the individual SLAs, which isn't surprising, because an application that relies on multiple services has more potential failure points.

On the other hand, you can improve the composite SLA by creating independent fallback paths. For example, if SQL Database is unavailable, put transactions into a queue, to be processed later.



With this design, the application is still available even if it can't connect to the database. However, it fails if the database and the queue both fail at the same time. The expected percentage of time for a simultaneous failure is 0.0001 × 0.001, so the composite SLA for this combined path is:

* Database OR queue = 1.0 − (0.0001 × 0.001) = 99.99999%

The total composite SLA is:

* Web app AND (database OR queue) = 99.95% × 99.99999% = ~99.95%

But there are tradeoffs to this approach. The application logic is more complex, you are paying for the queue, and there may be data consistency issues to consider.

**SLA for multi-region deployments**. Another HA technique is to deploy the application in more than one region, and use Azure Traffic Manager to fail over if the application fails in one region. For a two-region deployment, the composite SLA is calculated as follows.

Let N be the composite SLA for the application deployed in one region. The expected chance that the application will fail in both regions at the same time is (1 − N) × (1 − N). Therefore,

* Combined SLA for both regions = 1 − (1 − N)(1 − N) = N + (1 − N)N

Finally, you must factor in the [SLA for Traffic Manager](https://azure.microsoft.com/support/legal/sla/traffic-manager/v1_0/). At the time of this writing, the SLA for Traffic Manager SLA is 99.99%.

* Composite SLA = 99.99% × (combined SLA for both regions)

Also, failing over is not instantaneous and can result in some downtime during a failover. See [Traffic Manager endpoint monitoring and failover](https://docs.microsoft.com/en-us/azure/traffic-manager/traffic-manager-monitoring).

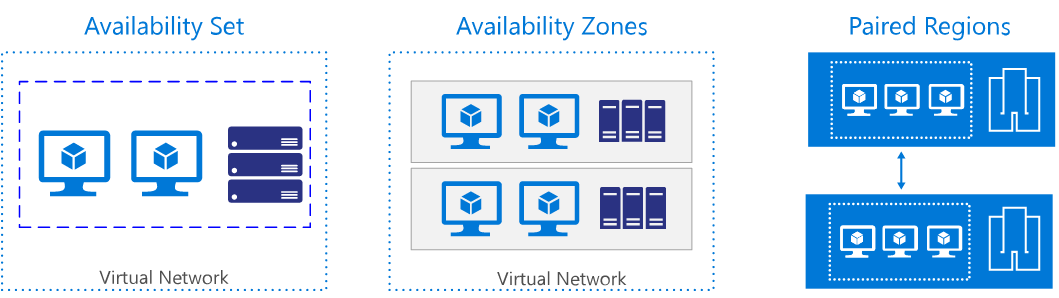
The calculated SLA number is a useful baseline, but it doesn't tell the whole story about availability. Often, an application can degrade gracefully when a non-critical path fails. Consider an application that shows a catalog of books. If the application can't retrieve the thumbnail image for the cover, it might show a placeholder image. In that case, failing to get the image does not reduce the application's uptime, although it affects the user experience.

## Redundancy and designing for failure

Failures can vary in the scope of their impact. Some hardware failures, such as a failed disk, may affect a single host machine. A failed network switch could affect a whole server rack. Less common are failures that disrupt a whole data center, such as loss of power in a data center. Rarely, an entire region could become unavailable.

One of the main ways to make an application resilient is through redundancy. But you need to plan for this redundancy when you design the application. Also, the level of redundancy that you need depends on your business requirements — not every application needs redundancy across regions to guard against a regional outage. In general, there is a tradeoff between greater redundancy and reliability versus higher cost and complexity.

Azure has a number of features to make an application redundant at every level of failure, from an individual VM to an entire region.



**Single VM**. Azure provides an uptime SLA for single VMs. Although you can get a higher SLA by running two or more VMs, a single VM may be reliable enough for some workloads. For production workloads, we recommend using two or more VMs for redundancy.

**Availability sets**. To protect against localized hardware failures, such as a disk or network switch failing, deploy two or more VMs in an availability set. An availability set consists of two or more fault domains that share a common power source and network switch. VMs in an availability set are distributed across the fault domains, so if a hardware failure affects one fault domain, network traffic can still be routed the VMs in the other fault domains. For more information about Availability Sets, see [Manage the availability of Windows virtual machines in Azure](https://docs.microsoft.com/en-us/azure/virtual-machines/windows/manage-availability).

**Availability zones**. An Availability Zone is a physically separate zone within an Azure region. Each Availability Zone has a distinct power source, network, and cooling. Deploying VMs across availability zones helps to protect an application against datacenter-wide failures.

**Paired regions**. To protect an application against a regional outage, you can deploy the application across multiple regions, using Azure Traffic Manager to distribute internet traffic to the different regions. Each Azure region is paired with another region. Together, these form a [regional pair](https://docs.microsoft.com/en-us/azure/best-practices-availability-paired-regions). With the exception of Brazil South, regional pairs are located within the same geography in order to meet data residency requirements for tax and law enforcement jurisdiction purposes.

When you design a multi-region application, take into account that network latency across regions is higher than within a region. For example, if you are replicating a database to enable failover, use synchronous data replication within a region, but asynchronous data replication across regions.

|  | **Availability Set** | **Availability Zone** | **Paired region** |
| --- | --- | --- | --- |
| Scope of failure | Rack | Datacenter | Region |
| Request routing | Load Balancer | Cross-zone Load Balancer | Traffic Manager |
| Network latency | Very low | Low | Mid to high |
| Virtual network | VNet | VNet | Cross-region VNet peering |

## Designing for resiliency

During the design phase, you should perform a failure mode analysis (FMA). The goal of an FMA is to identify possible points of failure, and define how the application will respond to those failures.

* How will the application detect this type of failure?
* How will the application respond to this type of failure?
* How will you log and monitor this type of failure?

For more information about the FMA process, with specific recommendations for Azure, see [Azure resiliency guidance: Failure mode analysis](https://docs.microsoft.com/en-us/azure/architecture/resiliency/failure-mode-analysis).

### Example of identifying failure modes and detection strategy

**Failure point:** Call to an external web service / API.

| **Failure mode** | **Detection strategy** |
| --- | --- |
| Service is unavailable | HTTP 5xx |
| Throttling | HTTP 429 (Too Many Requests) |
| Authentication | HTTP 401 (Unauthorized) |
| Slow response | Request times out |

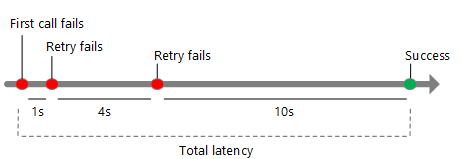
## Resiliency strategies

This section provides a survey of some common resiliency strategies. Most of these are not limited to a particular technology. The descriptions in this section summarize the general idea behind each technique, with links to further reading.

### Retry transient failures

Transient failures can be caused by momentary loss of network connectivity, a dropped database connection, or a timeout when a service is busy. Often, a transient failure can be resolved simply by retrying the request. For many Azure services, the client SDK implements automatic retries, in a way that is transparent to the caller; see [Retry service specific guidance](https://docs.microsoft.com/en-us/azure/architecture/best-practices/retry-service-specific).

Each retry attempt adds to the total latency. Also, too many failed requests can cause a bottleneck, as pending requests accumulate in the queue. These blocked requests might hold critical system resources such as memory, threads, database connections, and so on, which can cause cascading failures. To avoid this, increase the delay between each retry attempt, and limit the total number of failed requests.



For more information, see [Retry Pattern](https://docs.microsoft.com/en-us/azure/architecture/patterns/retry).

### Load balance across instances

For scalability, a cloud application should be able to scale out by adding more instances. This approach also improves resiliency, because unhealthy instances can be removed from rotation.

For example:

* Put two or more VMs behind a load balancer. The load balancer distributes traffic to all the VMs. See [Run load-balanced VMs for scalability and availability](https://docs.microsoft.com/en-us/azure/architecture/reference-architectures/virtual-machines-windows/multi-vm).
* Scale out an Azure App Service app to multiple instances. App Service automatically balances load across instances. See [Basic web application](https://docs.microsoft.com/en-us/azure/architecture/reference-architectures/app-service-web-app/basic-web-app).
* Use [Azure Traffic Manager](https://azure.microsoft.com/services/traffic-manager/) to distribute traffic across a set of endpoints.

### Replicate data

Replicating data is a general strategy for handling non-transient failures in a data store. Many storage technologies provide built-in replication, including Azure SQL Database, Cosmos DB, and Apache Cassandra.

It's important to consider both the read and write paths. Depending on the storage technology, you might have multiple writable replicas, or a single writable replica and multiple read-only replicas.

To maximize availability, replicas can be placed in multiple regions. However, this increases the latency when replicating the data. Typically, replicating across regions is done asynchronously, which implies an eventual consistency model and potential data loss if a replica fails.

### Degrade gracefully

If a service fails and there is no failover path, the application may be able to degrade gracefully while still providing an acceptable user experience. For example:

* Put a work item on a queue, to be handled later.
* Return an estimated value.
* Use locally cached data.
* Show the user an error message. (This option is better than having the application stop responding to requests.)

### Throttle high-volume users

Sometimes a small number of users create excessive load. That can have an impact on other users, reducing the overall availability of your application.

When a single client makes an excessive number of requests, the application might throttle the client for a certain period of time. During the throttling period, the application refuses some or all of the requests from that client (depending on the exact throttling strategy). The threshold for throttling might depend on the customer's service tier.

Throttling does not imply the client was necessarily acting maliciously, only that it exceeded its service quota. In some cases, a consumer might consistently exceed their quota or otherwise behave badly. In that case, you might go further and block the user. Typically, this is done by blocking an API key or an IP address range.

For more information, see [Throttling Pattern](https://docs.microsoft.com/en-us/azure/architecture/patterns/throttling).

### Use a circuit breaker

The Circuit Breaker pattern can prevent an application from repeatedly trying an operation that is likely to fail. This is similar to a physical circuit breaker, a switch that interrupts the flow of current when a circuit is overloaded.

The circuit breaker wraps calls to a service. It has three states:

* **Closed**. This is the normal state. The circuit breaker sends requests to the service, and a counter tracks the number of recent failures. If the failure count exceeds a threshold within a given time period, the circuit breaker switches to the Open state.
* **Open**. In this state, the circuit breaker immediately fails all requests, without calling the service. The application should use a mitigation path, such as reading data from a replica or simply returning an error to the user. When the circuit breaker switches to Open, it starts a timer. When the timer expires, the circuit breaker switches to the Half-open state.
* **Half-open**. In this state, the circuit breaker lets a limited number of requests go through to the service. If they succeed, the service is assumed to be recovered, and the circuit breaker switches back to the Closed state. Otherwise, it reverts to the Open state. The Half-Open state prevents a recovering service from suddenly being inundated with requests.

For more information, see [Circuit Breaker Pattern](https://msdn.microsoft.com/library/dn589784.aspx).

### Use load leveling to smooth out spikes in traffic

Applications may experience sudden spikes in traffic, which can overwhelm services on the backend. If a backend service cannot respond to requests quickly enough, it may cause requests to queue (back up), or cause the service to throttle the application.

To avoid this, you can use a queue as a buffer. When there is a new work item, instead of calling the backend service immediately, the application queues a work item to run asynchronously. The queue acts as a buffer that smooths out peaks in the load.

For more information, see [Queue-Based Load Leveling Pattern](https://docs.microsoft.com/en-us/azure/architecture/patterns/queue-based-load-leveling).

### Isolate critical resources

Failures in one subsystem can sometimes cascade, causing failures in other parts of the application. This can happen if a failure causes some resources, such as threads or sockets, not to get freed in a timely manner, leading to resource exhaustion.

To avoid this, you can partition a system into isolated groups, so that a failure in one partition does not bring down the entire system. This technique is sometimes called the Bulkhead pattern.

Examples:

* Partition a database (for example, by tenant) and assign a separate pool of web server instances for each partition.
* Use separate thread pools to isolate calls to different services. This helps to prevent cascading failures if one of the services fails. For an example, see the Netflix [Hystrix library](http://techblog.netflix.com/2012/11/hystrix.html).
* Use [containers](https://en.wikipedia.org/wiki/Operating-system-level_virtualization) to limit the resources available to a particular subsystem.

### Apply compensating transactions

A compensating transaction is a transaction that undoes the effects of another completed transaction.

In a distributed system, it can be very difficult to achieve strong transactional consistency. Compensating transactions are a way to achieve consistency by using a series of smaller, individual transactions that can be undone at each step.

For example, to book a trip, a customer might reserve a car, a hotel room, and a flight. If any of these steps fails, the entire operation fails. Instead of trying to use a single distributed transaction for the entire operation, you can define a compensating transaction for each step. For example, to undo a car reservation, you cancel the reservation. In order to complete the whole operation, a coordinator executes each step. If any step fails, the coordinator applies compensating transactions to undo any steps that were completed.

For more information, see [Compensating Transaction Pattern](https://msdn.microsoft.com/library/dn589804.aspx).

## Testing for resiliency

Generally, you can't test resiliency in the same way that you test application functionality (by running unit tests and so on). Instead, you must test how the end-to-end workload performs under failure conditions which only occur intermittently.

Testing is an iterative process. Test the application, measure the outcome, analyze and address any failures that result, and repeat the process.

**Fault injection testing**. Test the resiliency of the system during failures, either by triggering actual failures or by simulating them. Here are some common failure scenarios to test:

* Shut down VM instances.
* Crash processes.
* Expire certificates.
* Change access keys.
* Shut down the DNS service on domain controllers.
* Limit available system resources, such as RAM or number of threads.
* Unmount disks.
* Redeploy a VM.

Measure the recovery times and verify that your business requirements are met. Test combinations of failure modes as well. Make sure that failures don't cascade, and are handled in an isolated way.

This is another reason why it's important to analyze possible failure points during the design phase. The results of that analysis should be inputs into your test plan.

**Load testing**. Load test the application using a tool such as [Visual Studio Team Services](https://www.visualstudio.com/features/vso-cloud-load-testing-vs.aspx) or [Apache JMeter](http://jmeter.apache.org/). Load testing is crucial for identifying failures that only happen under load, such as the backend database being overwhelmed or service throttling. Test for peak load, using production data or synthetic data that is as close to production data as possible. The goal is to see how the application behaves under real-world conditions.

## Resilient deployment

Once an application is deployed to production, updates are a possible source of errors. In the worst case, a bad update can cause downtime. To avoid this, the deployment process must be predictable and repeatable. Deployment includes provisioning Azure resources, deploying application code, and applying configuration settings. An update may involve all three, or a subset.

The crucial point is that manual deployments are prone to error. Therefore, it's recommended to have an automated, idempotent process that you can run on demand, and re-run if something fails.

* Use Resource Manager templates to automate provisioning of Azure resources.
* Use [Azure Automation Desired State Configuration](https://docs.microsoft.com/en-us/azure/automation/automation-dsc-overview) (DSC) to configure VMs.
* Use an automated deployment process for application code.

Two concepts related to resilient deployment are infrastructure as code and immutable infrastructure.

* **Infrastructure as code** is the practice of using code to provision and configure infrastructure. Infrastructure as code may use a declarative approach or an imperative approach (or a combination of both). Resource Manager templates are an example of a declarative approach. PowerShell scripts are an example of an imperative approach.
* **Immutable infrastructure** is the principle that you shouldn’t modify infrastructure after it’s deployed to production. Otherwise, you can get into a state where ad hoc changes have been applied, so it's hard to know exactly what changed, and hard to reason about the system.

Another question is how to roll out an application update. We recommend techniques such as blue-green deployment or canary releases, which push updates in highly controlled way to minimize possible impacts from a bad deployment.

* [Blue-green deployment](http://martinfowler.com/bliki/BlueGreenDeployment.html) is a technique where an update is deployed into a production environment separate from the live application. After you validate the deployment, switch the traffic routing to the updated version. For example, Azure App Service Web Apps enables this with staging slots.
* [Canary releases](http://martinfowler.com/bliki/CanaryRelease.html) are similar to blue-green deployments. Instead of switching all traffic to the updated version, you roll out the update to a small percentage of users, by routing a portion of the traffic to the new deployment. If there is a problem, back off and revert to the old deployment. Otherwise, route more of the traffic to the new version, until it gets 100% of the traffic.

Whatever approach you take, make sure that you can roll back to the last-known-good deployment, in case the new version is not functioning. Also, if errors occur, the application logs must indicate which version caused the error.

## Monitoring and diagnostics

Monitoring and diagnostics are crucial for resiliency. If something fails, you need to know that it failed, and you need insights into the cause of the failure.

Monitoring a large-scale distributed system poses a significant challenge. Think about an application that runs on a few dozen VMs — it's not practical to log into each VM, one at a time, and look through log files, trying to troubleshoot a problem. Moreover, the number of VM instances is probably not static. VMs get added and removed as the application scales in and out, and occasionally an instance may fail and need to be reprovisioned. In addition, a typical cloud application might use multiple data stores (Azure storage, SQL Database, Cosmos DB, Redis cache), and a single user action may span multiple subsystems.

You can think of the monitoring and diagnostics process as a pipeline with several distinct stages:

* **Instrumentation**. The raw data for monitoring and diagnostics comes from a variety of sources, including application logs, web server logs, OS performance counters, database logs, and diagnostics built into the Azure platform. Most Azure services have a diagnostics feature that you can use to determine the cause of problems.
* **Collection and storage**. Raw instrumentation data can be held in various locations and with various formats (e.g., application trace logs, IIS logs, performance counters). These disparate sources are collected, consolidated, and put into reliable storage.
* **Analysis and diagnosis**. After the data is consolidated, it can be analyzed to troubleshoot issues and provide an overall view of application health.
* **Visualization and alerts**. In this stage, telemetry data is presented in such a way that an operator can quickly notice problems or trends. Example include dashboards or email alerts.

Monitoring is not the same as failure detection. For example, your application might detect a transient error and retry, resulting in no downtime. But it should also log the retry operation, so that you can monitor the error rate, in order to get an overall picture of application health.

Application logs are an important source of diagnostics data. Best practices for application logging include:

* Log in production. Otherwise, you lose insight where you need it most.
* Log events at service boundaries. Include a correlation ID that flows across service boundaries. If a transaction flows through multiple services and one of them fails, the correlation ID will help you pinpoint why the transaction failed.
* Use semantic logging, also known as structured logging. Unstructured logs make it hard to automate the consumption and analysis of the log data, which is needed at cloud scale.
* Use asynchronous logging. Otherwise, the logging system itself can cause the application to fail by causing requests to back up, as they block while waiting to write a logging event.
* Application logging is not the same as auditing. Auditing may be done for compliance or regulatory reasons. As such, audit records must be complete, and it's not acceptable to drop any while processing transactions. If an application requires auditing, this should be kept separate from diagnostics logging.

For more information about monitoring and diagnostics, see [Monitoring and diagnostics guidance](https://docs.microsoft.com/en-us/azure/architecture/best-practices/monitoring).

## Manual failure responses

Previous sections have focused on automated recovery strategies, which are critical for high availability. However, sometimes manual intervention is needed.

* **Alerts**. Monitor your application for warning signs that may require proactive intervention. For example, if you see that SQL Database or Cosmos DB consistently throttles your application, you might need to increase your database capacity or optimize your queries. In this example, even though the application might handle the throttling errors transparently, your telemetry should still raise an alert so that you can follow up.
* **Manual failover**. Some systems cannot fail over automatically and require a manual failover.
* **Operational readiness testing**. If your application fails over to a secondary region, you should perform an operational readiness test before you fail back to the primary region. The test should verify that the primary region is healthy and ready to receive traffic again.
* **Data consistency check**. If a failure happens in a data store, there may be data inconsistencies when the store becomes available again, especially if the data was replicated.
* **Restoring from backup**. For example, if SQL Database experiences a regional outage, you can geo-restore the database from the latest backup.

Document and test your disaster recovery plan. Evaluate the business impact of application failures. Automate the process as much as possible, and document any manual steps, such as manual failover or data restoration from backups. Regularly test your disaster recovery process to validate and improve the plan.

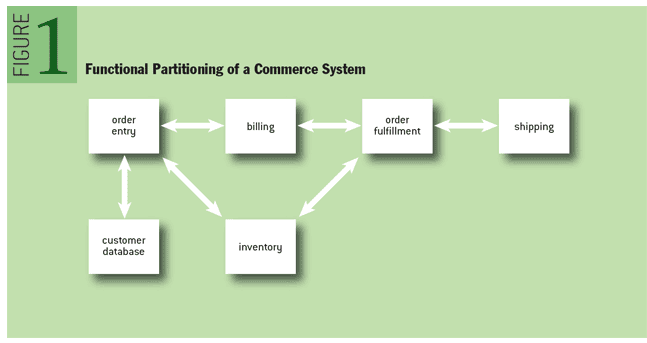
## Summary

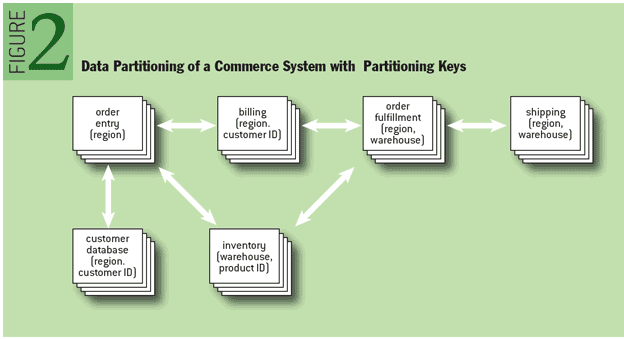
This article discussed resiliency from a holistic perspective, emphasizing some of the unique challenges of the cloud. These include the distributed nature of cloud computing, the use of commodity hardware, and the presence of transient network faults.

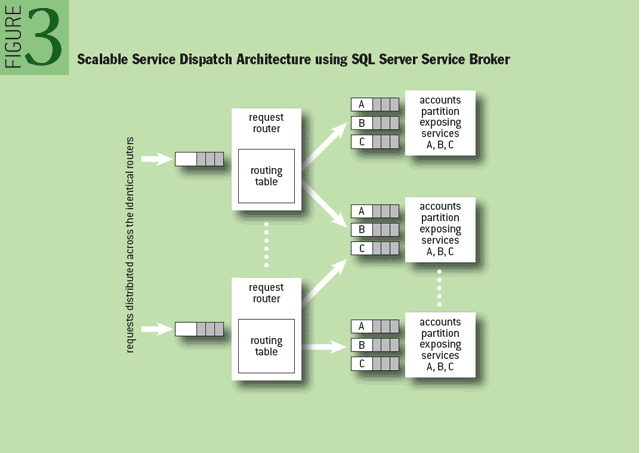
Here are the major points to take away from this article:

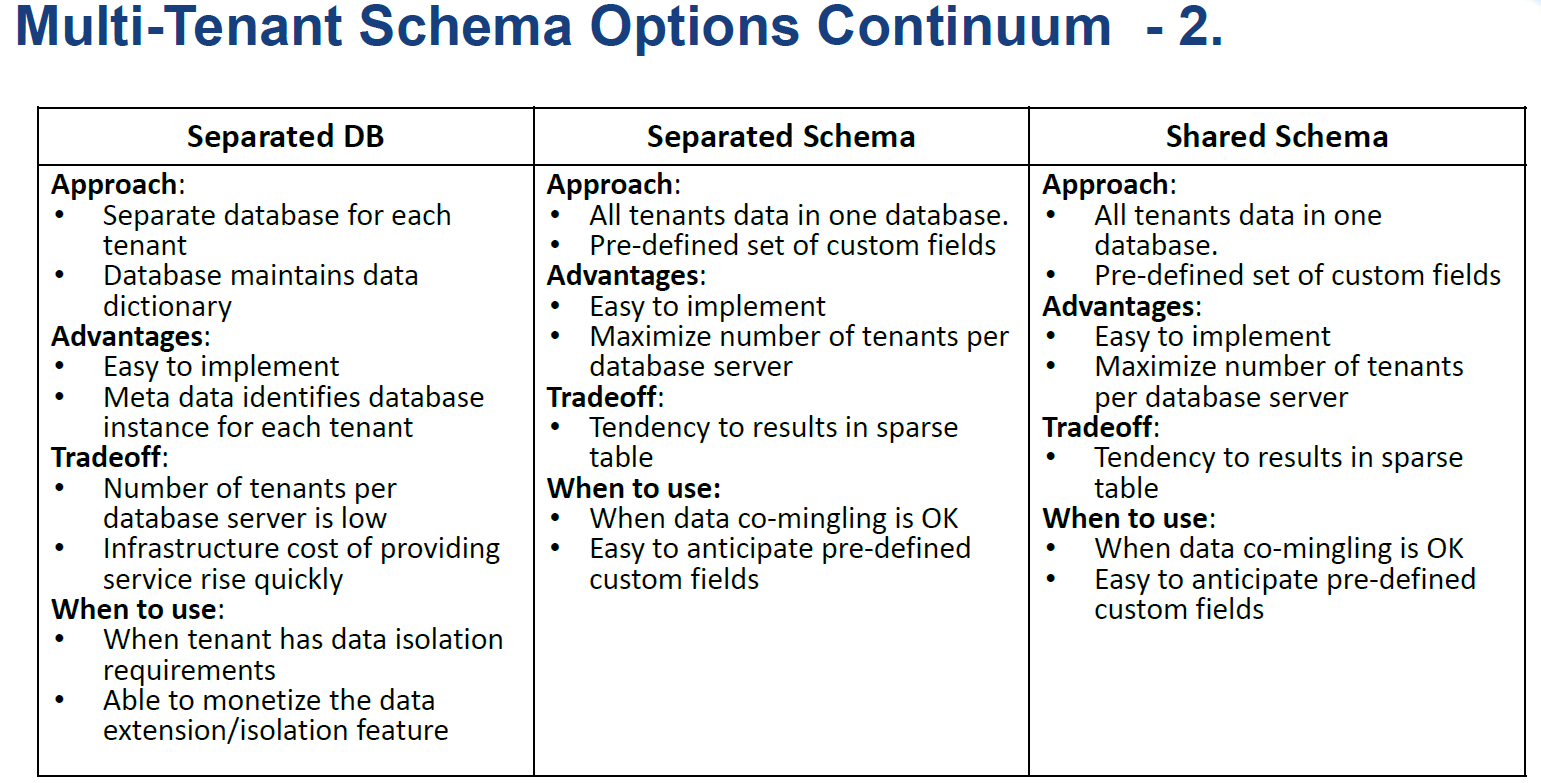
* Resiliency leads to higher availability, and lower mean time to recover from failures.
* Achieving resiliency in the cloud requires a different set of techniques from traditional on-premises solutions.
* Resiliency does not happen by accident. It must be designed and built in from the start.
* Resiliency touches every part of the application lifecycle, from planning and coding to operations.
* Test and monitor!

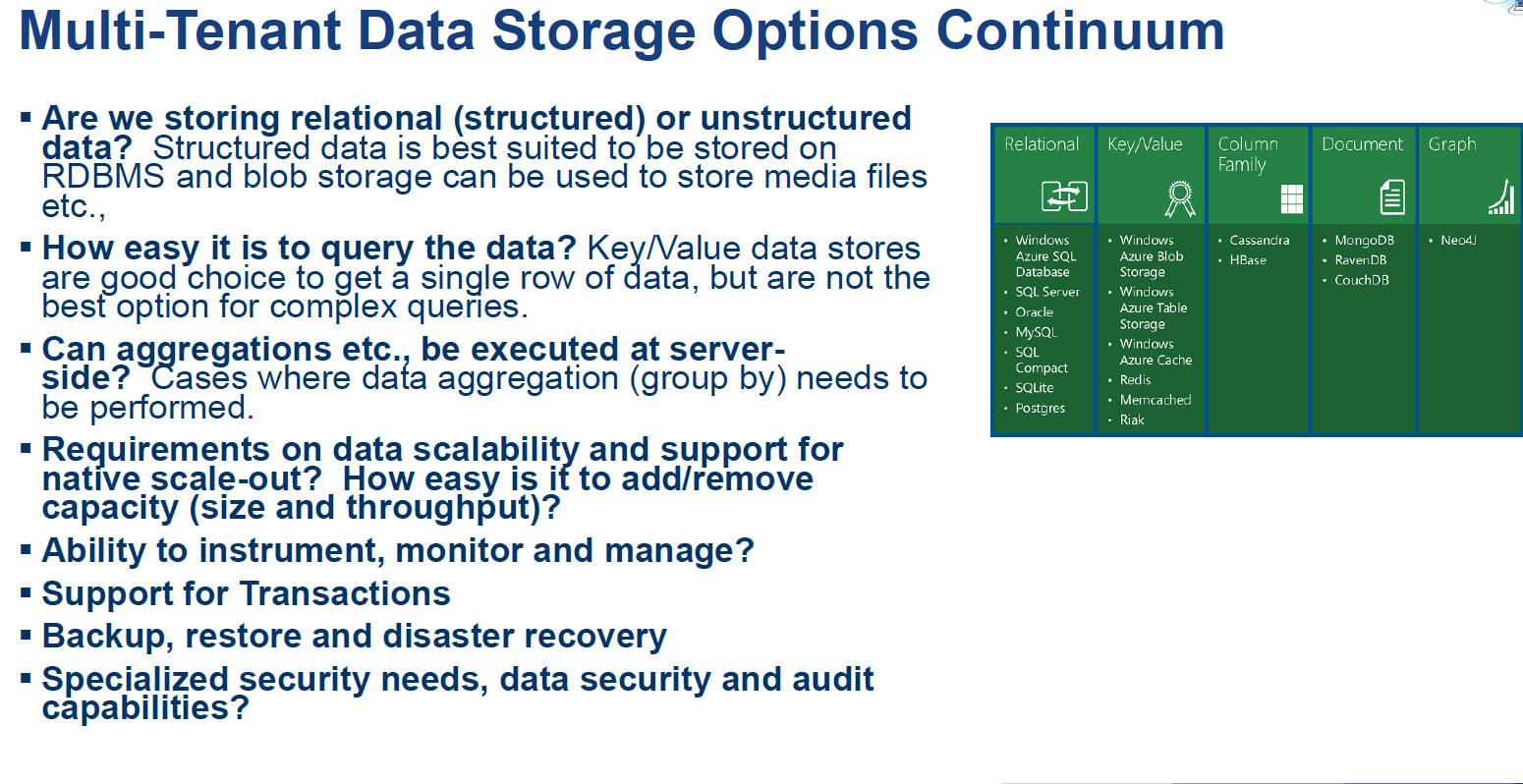
Scaling from Data Storage SQL

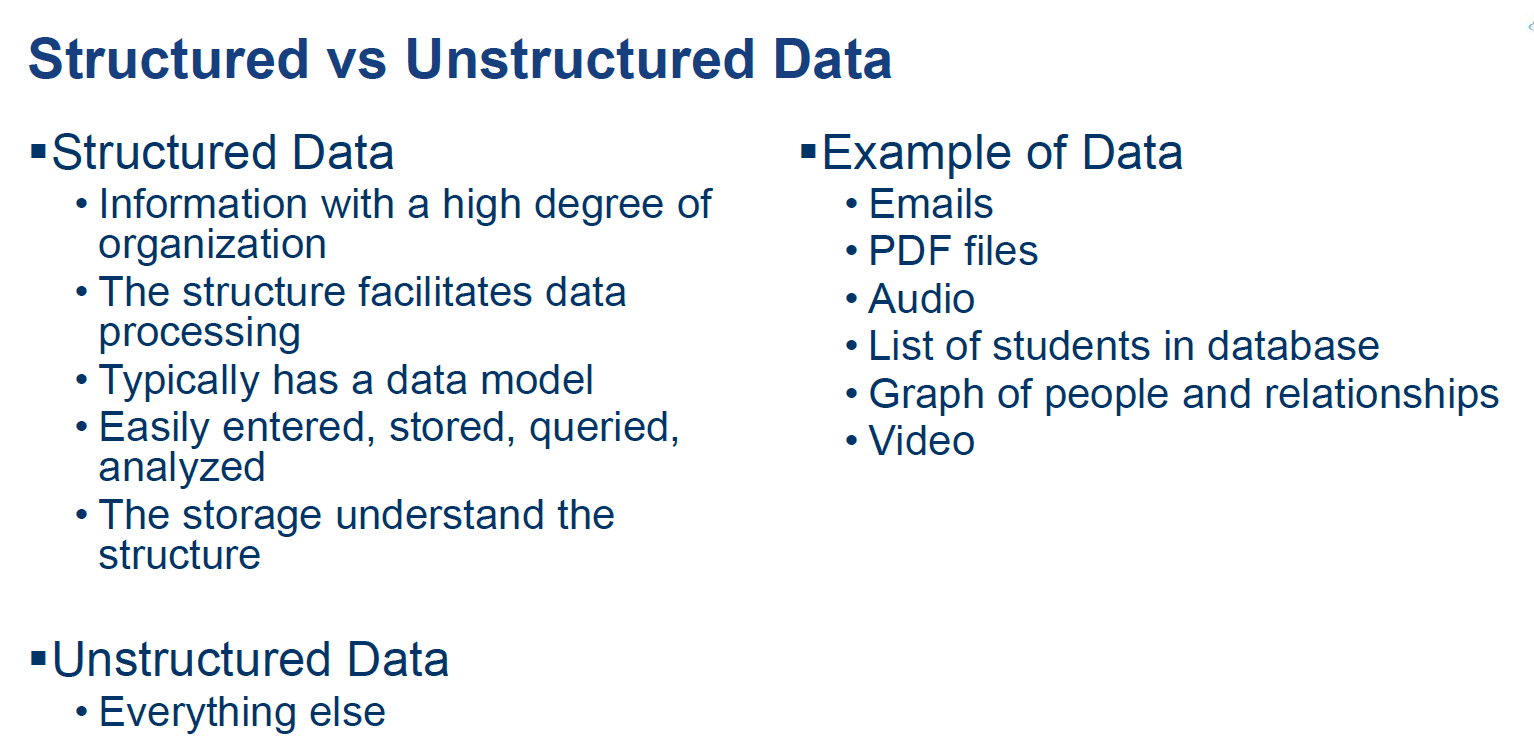












**Columnar Database :**

While a relational database is optimized for storing rows of data, typically for transactional applications, a columnar database is optimized for fast retrieval of columns of data, typically in analytical applications. Column-oriented storage for database tables is an important factor in analytic query performance because it [drastically reduces the overall disk I/O requirements](https://docs.aws.amazon.com/redshift/latest/dg/c_columnar_storage_disk_mem_mgmnt.html) and reduces the amount of data you need to load from disk.

columnar databases are good for:

* Queries that involve only a few columns
* Aggregation queries against vast amounts of data
* Column-wise compression

But are not so good at:

* Incremental data loading
* Online Transaction Processing (OLTP) usage
* Queries against only a few rows

**Difference between Key Value Database & Document DB:**

Key-value databases and document databases are quite similar. Key-value databases are the simplest of the NoSQL databases: The basic data structure is a dictionary or map. You can store a value, such as an integer, string, a JSON structure, or an array, along with a key used to reference that value.

Document databases organize documents into groups called collections, which are analogous to the tables in relational databases. By contrast, key-value databases store all key-value pairs together in a single namespace, which is analogous to a relational schema.

Key-value pairs of similar types, such as IDs and names, are stored with dissimilar value pairs, such as IDs and customer orders. This sounds like a potential problem, but whether it is depends on the search capabilities of the key-value database. When searching using freeform text to find the ID associated with a given name, you can search across all value types—not just names. However, unless search queries are properly crafted, this approach could end up being less efficient than using a document database.

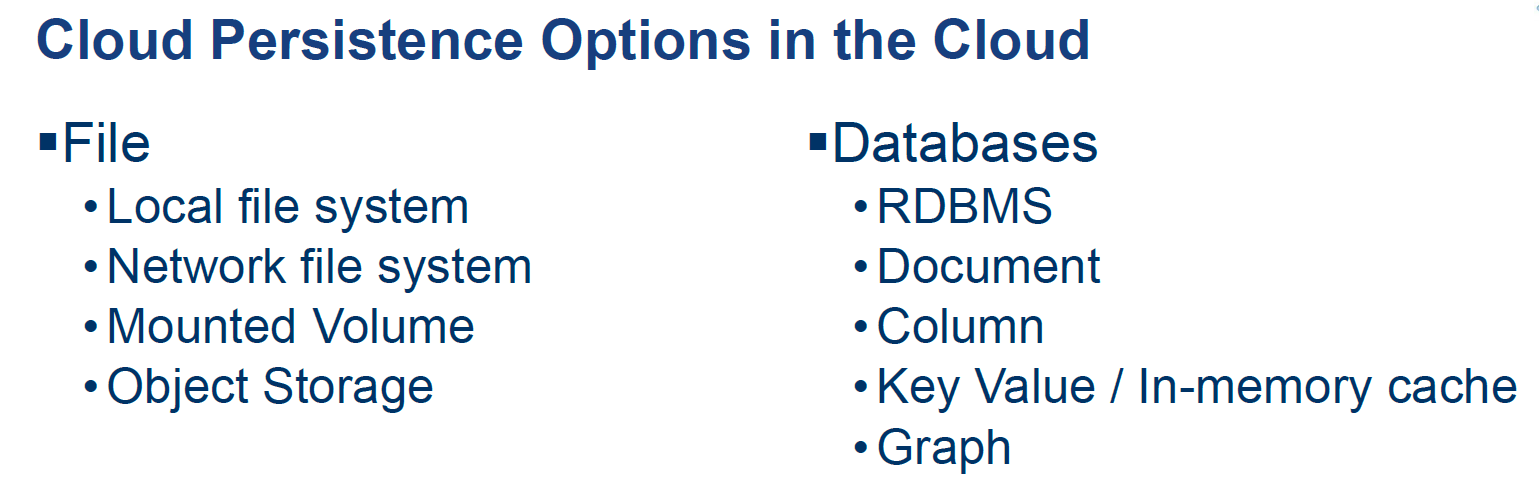
Document database collections allow developers to apply a high level of organization to their databases. For example, a document database for an e-commerce site might include collections for customers, orders, and products:

* • *Customers* would include fields such as name, shipping address, and billing address.
* • *Orders* would include fields such as product and shipping address data.
* • *Products* would include fields such as department and price data.

Document databases can also provide better performance when working with complex data sets. Separating collection entities by type, such as orders and customer profiles, can help with performance. Similarly, large collections, such as a collection of products, can be partitioned to improve query performance. Partitioning splits collections over multiple servers, allocating a subset of work to each server.

Document databases also support indexing, which can improve query performance by using filter criteria, such as searching for all orders placed in the last 10 days. However, it's often best to limit the use of indexes, confining them to fields that are commonly used in filtering query results. Having too many indexes can slow write operations. In the end, using indexes is a tradeoff. You may get faster query responses at the expense of slower write operations, plus the cost of additional space to store index data.

For each of these classifications of databases, the actual implementations will vary from vendor to vendor with some offering different scheme and querying capabilities as well as other fields.   
There is a lot of cross over between the different types and generally speaking, they're all build to be distributed and scaled horizontally.  
  
This will give you a general overview of the differences:  
  
**Key-Value** stores store keys value pairs, generally in buckets, exactly like a hash table data structure...each key must be unique.  They are extremely fast for writing, and extremely fast for reading and updating...if you have the key.  They are slow on multiple updates and if you have to query the entire store.    
You see Key-value stores used a lot as caching stores because of their fast reads. (see: **Redis, Riak, memcached, Azure's tablestore**, etc)  
  
**Column stores** seem to store data in related rows, but they actually serialize the data into columns.    
With a row based database you would have:  
ID,firstname, lastname, websitename  
1:bart, loews, quora  
2:jim, finnegan, beginagain  
3:don, quixote, windmill  
The column store stores columns together, like this:  
1:bart,2:jim,3:don  
1:loews,2:finnegan,3:quixote  
1:quora,2:beginagain,3:windmill  
This allows for much faster querying and processing of data while storing data that's somewhat related (druid boasts billions of records per second).    
These are used in a lot of high powered big-data analysis where speed is critical. (**Cassandra, Google's BigTable, Druid**)  
  
**Document Stores**store data in "documents", typically XML or JSON documents.  They're typically schemaless, so each document can contain any data that you want them to have and you can change it on the fly.  Documents contain key value pairs, which can be any sort of value, array, or even another document.  For example:  
{  
FirstName: "Bart",  
LastName: "Loews",  
Children: [ {  
                    FirstName:"Tadd",  
                    Age: 4},  
                    {  
                      FirstName:"Todd",  
                      Age:4}  
                    ],  
Age: 35,  
Address:{  
               number:1234,  
               street: "Fake road",  
               City: "Fake City",  
               state: "VA",  
               Country: "USA"  
               }  
}  
Document stores allow you to play with your data and store it however you see fit. They have fast writing, good query times based on indexing, but the main advantage is the schema flexibility and nestibility.  
You could see these being used for pretty much any thing, some vendors will let you perform relational joins between documents, but generally, for the best flexibility you'll have to build in any join logic into your application.  
Some examples include: **MongoDB, OrientDB, CouchDB, Azure's DocumentDB, AWS DynamoDB, RethinkDB.**  
**Graph Stores**focus on relationships.  The best way to picture a graph store is like a mathematical graph where you have edges and vertices.  With a graph store data is stored as Nodes and Relationships.  A Node is basically a noun, a person, place, thing, entity, etc.  A relationship is a one or two way connection between two nodes.  A node could be people and a relationship could be a two way friendship. You could have two nodes, a link to a website, and a user, and the relationship could be a one way "like" from the user to the website.  
  
You can usually apply metadata documents to nodes and relationships, as well as labels. You can label a node as a user or a website or a pet or whatever. You can add data like name, age, gender to user nodes, it's schemaless and very flexible.  
  
Graph stores excel at determining relationships between nodes and finding patterns.  For example, you can use a graph store to determine friends of friends or friends of friends of friends or friends of friends to the nth degree.  You could figure out people you're not friends with, you can find people who are friends of friends who like the same things you do, you can find people who have worked with friends of friends who like the same things you do who don't have pets.    
  
They handle these crazy queries quickly.  
  
I'm unaware of who uses them, but they'd be great for social networking and for retail marketing purposes.  
  
Examples of this include: **neo4j and orientDB.**



**Difference Between EFS & EBS:**

storage options, which may be used for temporary storing data while/before processing it:

* SNS
* SQS
* Kinesis stream
* DynamoDB, SimpleDB

The costs above are just samples. There can be differences by region, and it can change at any point. Also there are extra costs for data transfer (out to the internet). However they **show a ratio between the prices of the services**.

There are a lot more differences between these services:

**EFS is:**

* [Generally Available](https://aws.amazon.com/about-aws/whats-new/2016/06/amazon-elastic-file-system-efs-is-now-generally-available/)(out of preview), but may not yet be available in your region
* Network filesystem (that means it may have bigger latency but it can be shared across several instances; even between regions)
* It is expensive compared to EBS (~10x more) but it gives extra features.
* It's a highly available service.
* It's a managed service
* You can attach the EFS storage to an EC2 Instance
* Can be accessed by multiple EC2 instances simultaneously
* Since 2016.dec.20 it's possible to attach your EFS storage directly to [on-premise servers via Direct Connect.](https://aws.amazon.com/blogs/aws/amazon-efs-update-on-premises-access-via-direct-connect-vpc/) ()

**EBS is:**

* A block storage (so you need to format it). This means you are able to choose which type of file system you want.
* As it's a block storage, you can use Raid 1 (or 0 or 10) with multiple block storages
* It is really fast
* It is relatively cheap
* With the new announcements from Amazon, you can store up to 16TB data per storage on SSD-s.
* You can snapshot an EBS (while it's still running) for backup reasons
* But it only exists in a particular region. Although you can migrate it to another region, you cannot just access it across regions (only if you share it via the EC2; but that means you have a file server)
* You need an EC2 instance to attach it to
* [New feature](https://aws.amazon.com/blogs/aws/amazon-ebs-update-new-elastic-volumes-change-everything/) (2017.Feb.15): You can now increase volume size, adjust performance, or change the volume type while the volume is in use. You can continue to use your application while the change takes effect.

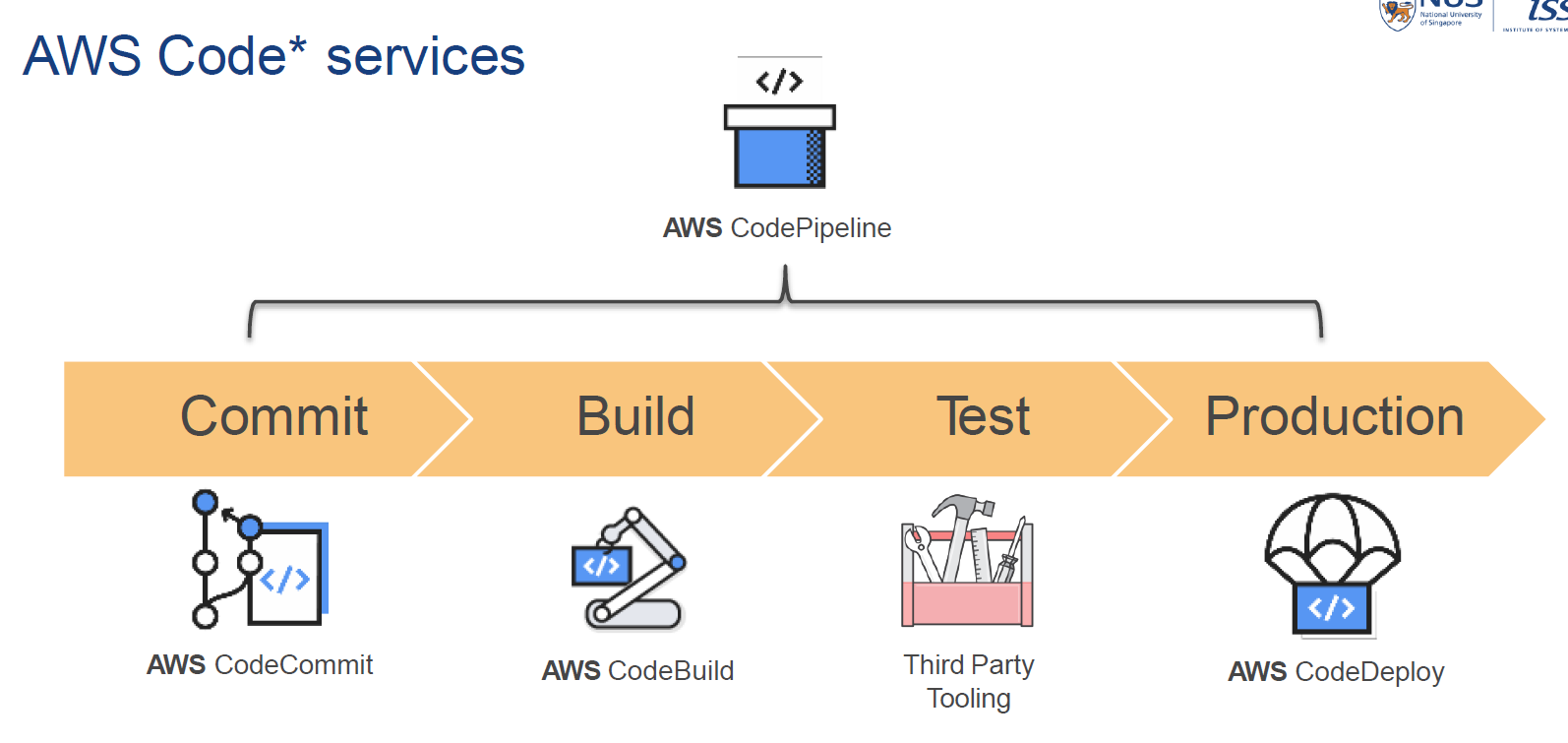
**S3 is:**

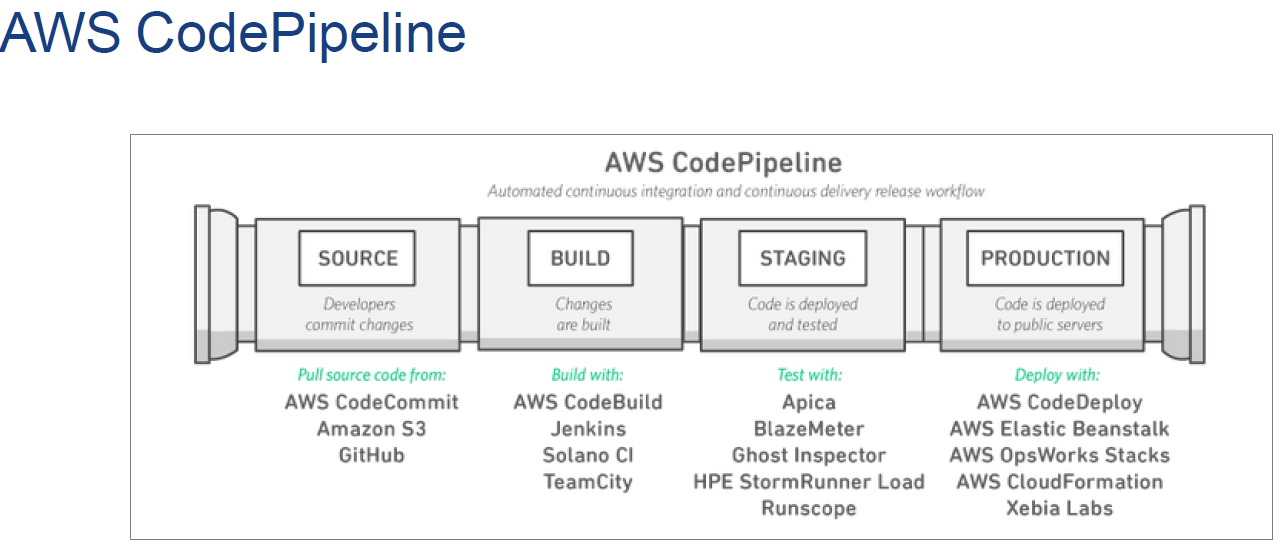
* An object store (not a file system).
* You can store files and "folders" but can't have locks, permissions etc like you would with a traditional file system
* This means, by default you can't just mount S3 and use it as your webserver
* But it's perfect for storing your images and videos for your website
* Great for short term archiving (e.g. a few weeks). It's good for long term archiving too, but Glacier is more cost efficient.
* Great for storing logs
* You can access the data from every region (extra costs may apply)
* Highly Available, Redundant. Basically data loss is not possible (99.999999999% durability, 99.9 uptime SLA)
* Much cheaper than EBS.
* You can serve the content directly to the internet, you can even have a full (static) website working direct from S3, without an EC2 instance

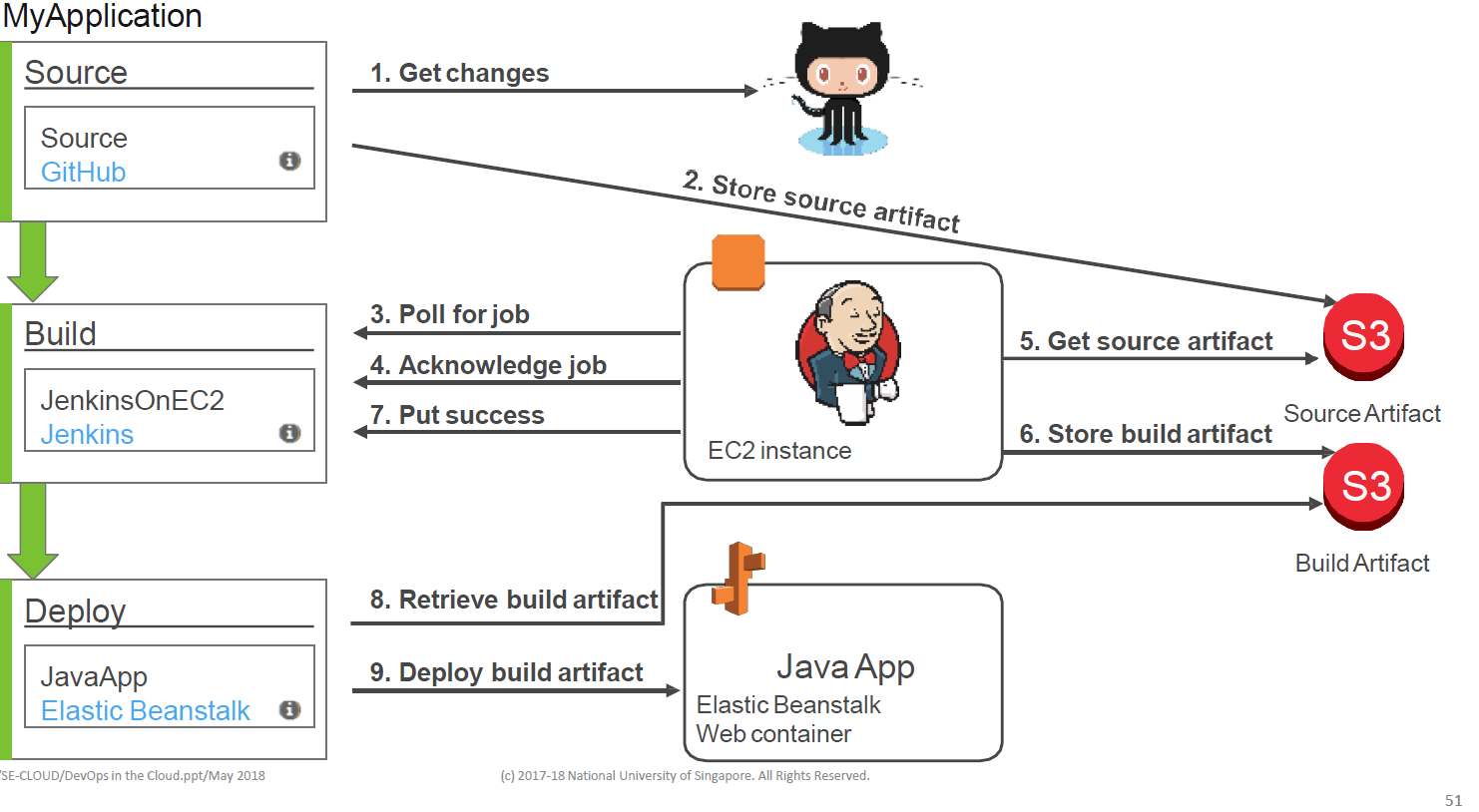
**Glacier is:**

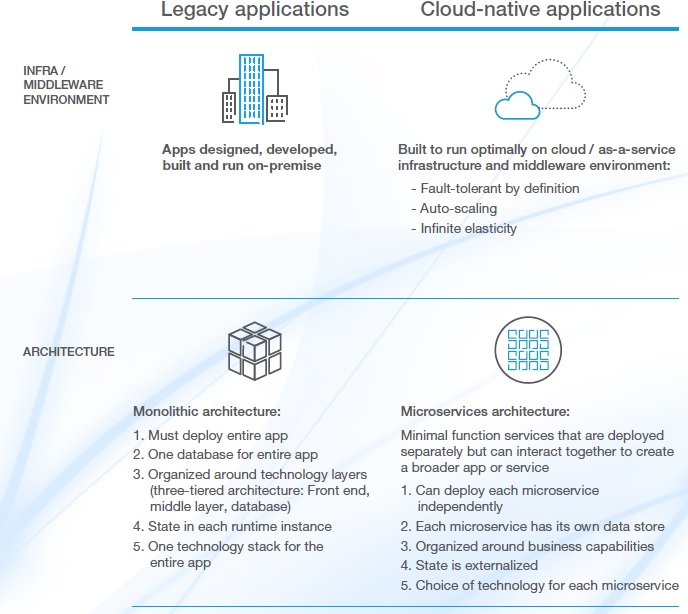
* Long term archive storage
* Extremely cheap to store
* Potentially very expensive to retrieve
* Takes up to 4 hours to "read back" your data (so only store items you know you won't need to retrieve for a long time)

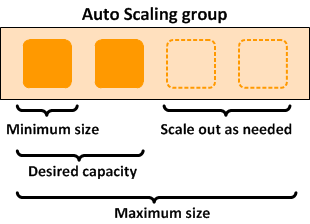
As it got mentioned in JDL's comment, there are several interesting aspects in terms of pricing. For example Glacier, S3, EFS allocates the storage for you based on your usage, while at EBS you need to predefine the allocated storage. Which means, you need to over estimate. ( However it's easy to add more storage to your EBS volumes, it requires some engineering, which means you always "overpay" your EBS storage, which makes it even more expensive.)

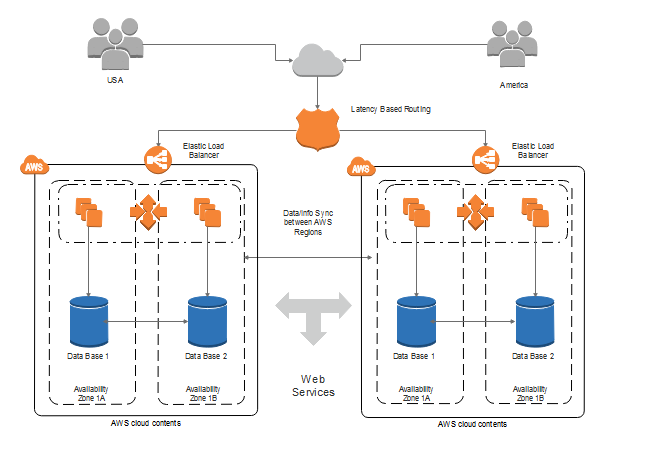


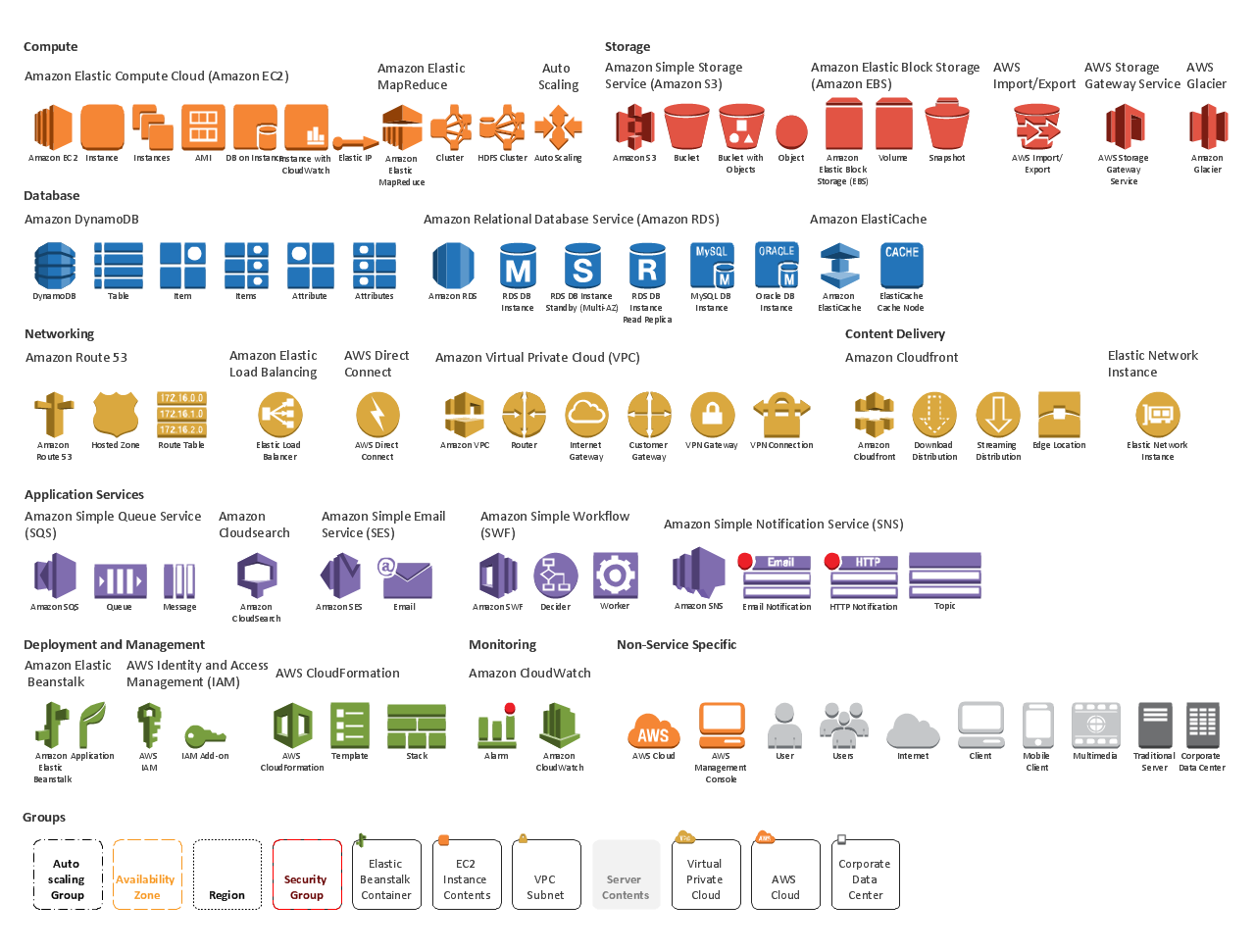


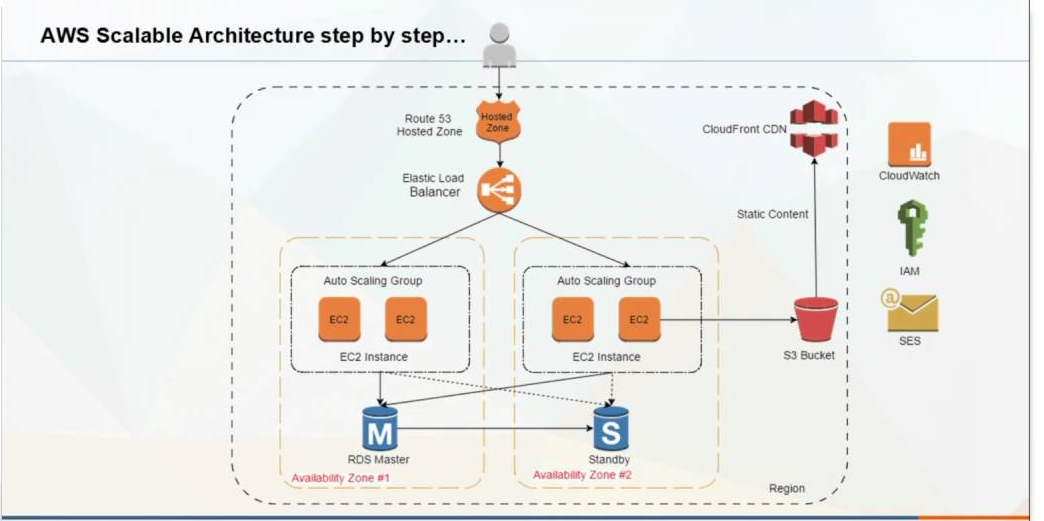


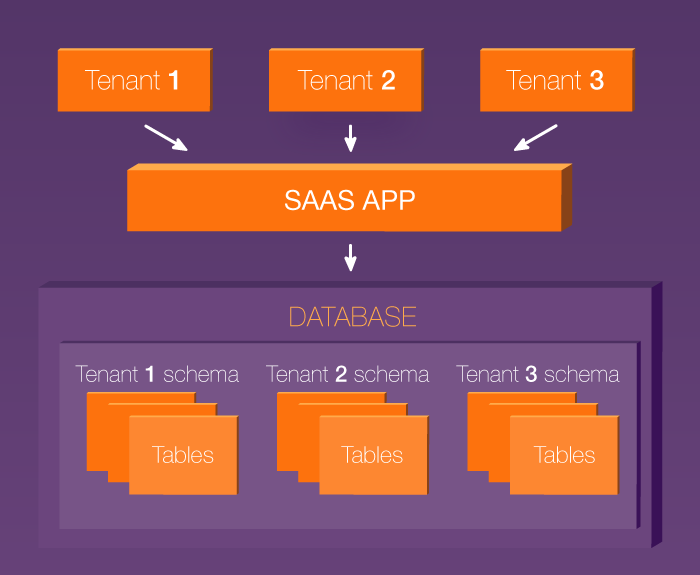
****

****

****

****



****

****