# Architecting for the Cloud

## AWS Best Practices

*February 2016*

# Notices

This document is provided for informational purposes only. It represents AWS's current product offerings and practices as of the date of issue of this document, which are subject to change without notice. Customers are responsible for making their own independent assessment of the information in this document and any use of AWS's products or services, each of which is provided "as is" without warranty of any kind, whether express or implied. This document does not create any warranties, representations, contractual commitments, conditions or assurances from AWS, its affiliates, suppliers or licensors. The responsibilities and liabilities of AWS to its customers are controlled by AWS agreements, and this document is not part of, nor does it modify, any agreement between AWS and its customers.

# Contents

# Abstract

This whitepaper is intended for solutions architects and developers who are building solutions that will be deployed on Amazon Web Services (AWS). It provides architectural patterns and advice on how to design systems that are secure, reliable, high performing, and cost efficient. It includes a discussion on how to take advantage of attributes that are specific to the dynamic nature of cloud computing (elasticity, infrastructure automation, etc.). In addition, this whitepaper also covers general patterns, explaining how these are evolving and how they are applied in the context of cloud computing.

# Introduction

Migrating applications to AWS, even without significant changes (an approach known as "lift and shift"), provides organizations the benefits of a secured and cost-efficient infrastructure. However, to make the most of the elasticity and agility possible with cloud computing, engineers will have to evolve their architectures to take advantage of the AWS capabilities.

For new applications, AWS customers have been discovering cloud-specific IT architecture patterns, driving even more efficiency and scalability. Those new architectures can support anything from real-time analytics of Internet-scale data to applications with unpredictable traffic from thousands of connected Internet of Things (IoT) or mobile devices.

This paper will highlight the principles to consider whether you are migrating existing applications to AWS or designing new applications for the cloud.

This whitepaper assumes basic understanding of the AWS services and solutions. If you are new to AWS, please first see the About AWS webpage[1].

# The Cloud Computing Difference

This section reviews how cloud computing differs from a traditional environment and why those new best practices have emerged.

## IT Assets Become Programmable Resources

In a non-cloud environment you would have to provision capacity based on a guess of a theoretical maximum peak. This can result in periods where expensive resources are idle or occasions of insufficient capacity. With cloud computing, you can access as much or as little as you need, and dynamically scale to meet actual demand, while only paying for what you use.

On AWS, servers, databases, storage, and higher-level application components can be instantiated within seconds. You can treat these as temporary and disposable resources, free from the inflexibility and constraints of a fixed and finite IT infrastructure. This resets the way you approach change management, testing, reliability, and capacity planning.

## Global, Available, and Unlimited Capacity

Using the global infrastructure of AWS, you can deploy your application to the AWS Region[2] that best meets your requirements (e.g., proximity to your end users, compliance, or data residency constraints, cost, etc.). For global applications, you can reduce latency to end users around the world by using the Amazon CloudFront content delivery network. It is also much easier to operate production applications and databases across multiple data centers to achieve high availability and fault tolerance. Together with the virtually unlimited on-demand capacity that is available to AWS customers, you can think differently about how to enable future expansion via your IT architecture.

## Higher Level Managed Services

Apart from the compute resources of Amazon Elastic Compute Cloud (Amazon EC2), AWS customers also have access to a broad set of storage, database, analytics, application, and deployment services. Because these services are instantly available to developers, they reduce dependency on in-house specialized skills and allow organizations to deliver new solutions faster. These services are managed by AWS, which can lower operational complexity and cost. AWS

managed services are designed for scalability and high availability, so they can reduce risk for your implementations.

## Security Built In

On traditional IT, infrastructure security auditing would often be a periodic and manual process. The AWS cloud instead provides governance capabilities that enable continuous monitoring of configuration changes to your IT resources. Since AWS assets are programmable resources, your security policy can be formalized and embedded with the design of your infrastructure. With the ability to spin up temporary environments, security testing can now become part of your continuous delivery pipeline. Finally, solutions architects can leverage a plethora of native AWS security and encryption features that can help achieve higher levels of data protection and compliance.

# Design Principles

In this section, we provide design patterns and architectural options that can be applied in a wide variety of use cases.

## Scalability

Systems that are expected to grow over time need to be built on top of a scalable architecture. Such an architecture can support growth in users, traffic, or data size with no drop in performance. It should provide that scale in a linear manner where adding extra resources results in at least a proportional increase in ability to serve additional load. Growth should introduce economies of scale, and cost should follow the same dimension that generates business value out of that system. While cloud computing provides virtually unlimited on-demand capacity, your design needs to be able to take advantage of those resources seamlessly. There are generally two ways to scale an IT architecture: vertically and horizontally.

### Scaling Vertically

Scaling vertically takes place through an increase in the specifications of an individual resource (e.g., upgrading a server with a larger hard drive or a faster CPU). On Amazon EC2, this can easily be achieved by stopping an instance and resizing it to an instance type that has more RAM, CPU, IO, or networking capabilities. This way of scaling can eventually hit a limit and it is not always a

cost efficient or highly available approach. However, it is very easy to implement and can be sufficient for many use cases especially in the short term.

## Scaling Horizontally

Scaling horizontally takes place through an increase in the number of resources (e.g., adding more hard drives to a storage array or adding more servers to support an application). This is a great way to build Internet-scale applications that leverage the elasticity of cloud computing. Not all architectures are designed to distribute their workload to multiple resources, so let's examine some of the possible scenarios.

### *Stateless Applications*

When users or services interact with an application they will often perform a series of interactions that form a session. A stateless application is an application that needs no knowledge of previous interactions and stores no session information. Such an example could be an application that, given the same input, provides the same response to any end user. A stateless application can scale horizontally since any request can be serviced by any of the available compute resources (e.g., EC2 instances, AWS Lambda functions). With no session data to be shared, you can simply add more compute resources as needed. When that capacity is no longer required, any individual resource can be safely terminated (after running tasks have been drained). Those resources do not need to be aware of the presence of their peers – all that is required is a way to distribute the workload to them.

***How to distribute load to multiple nodes***

*Push model: A popular way to distribute a workload is through the use of a load balancing solution like the Elastic Load Balancing (ELB) service. Elastic Load Balancing routes incoming application requests across multiple EC2 instances. An alternative approach would be to implement a DNS round robin (e.g., with Amazon Route 53). In this case, DNS responses return an IP address from a list of valid hosts in a round robin fashion. While easy to implement, this approach does not always work well with the elasticity of cloud computing. This is because even if you can set low time to live (TTL) values for your DNS records, caching DNS resolvers are outside the control of Amazon Route 53 and might not always respect your settings.*

*Pull model: Asynchronous event-driven workloads do not require a load balancing solution because you can implement a pull model instead. In a pull model, tasks that need to be performed or data that need to be processed*

*could be stored as messages in a queue using Amazon Simple Queue Service (Amazon SQS) or as a streaming data solution like Amazon Kinesis. Multiple compute nodes can then pull and consume those messages, processing them in a distributed fashion.*

### *Stateless Components*

In practice, most applications need to maintain some kind of state information. For example, web applications need to track whether a user is signed in, or else they might present personalized content based on previous actions. An automated multi-step process will also need to track previous activity to decide what its next action should be. You can still make a portion of these architectures stateless by not storing anything in the local file system that needs to persist for more than a single request.

For example, web applications can use HTTP cookies to store information about a session at the client's browser (e.g., items in the shopping cart). The browser passes that information back to the server at each subsequent request so that the application does not need to store it. However, there are two drawbacks with this approach. First, the content of the HTTP cookies can be tampered with at the client side, so you should always treat them as untrusted data that needs to be validated. Second, HTTP cookies are transmitted with every request, which means that you should keep their size to a minimum (to avoid unnecessary latency).

Consider only storing a unique session identifier in a HTTP cookie and storing more detailed user session information server-side. Most programming platforms provide a native session management mechanism that works this way, however this is often stored on the local file system by default. This would result in a stateful architecture. A common solution to this problem is to store user session information in a database. Amazon DynamoDB is a great choice due to its scalability, high availability, and durability characteristics. For many platforms there are open source drop-in replacement libraries that allow you to store native sessions in Amazon DynamoDB[3].

Other scenarios require storage of larger files (e.g., user uploads, interim results of batch processes, etc.). By placing those files in a shared storage layer like Amazon S3 or Amazon Elastic File System (Amazon EFS) you can avoid the introduction of stateful components. Another example is that of a complex multi-step workflow where you need to track the current state of each execution.

Amazon Simple Workflow Service (Amazon SWF) can be utilized to centrally store execution history and make these workloads stateless.

### *Stateful Components*

Inevitably, there will be layers of your architecture that you won't turn into stateless components. First, by definition, databases are stateful. (They will be covered separately in the Databases section.) In addition, many legacy applications were designed to run on a single server by relying on local compute resources. Other use cases might require client devices to maintain a connection to a specific server for prolonged periods of time. For example, real-time multiplayer gaming must offer multiple players a consistent view of the game world with very low latency. This is much simpler to achieve in a non-distributed implementation where participants are connected to the same server.

You might still be able to scale those components horizontally by distributing load to multiple nodes with "session affinity." In this model, you bind all the transactions of a session to a specific compute resource. You should be aware of the limitations of this model. Existing sessions do not directly benefit from the introduction of newly launched compute nodes. More importantly, session affinity cannot be guaranteed. For example, when a node is terminated or becomes unavailable, users bound to it will be disconnected and experience a loss of session-specific data (e.g., anything that is not stored in a shared resource like S3, EFS, a database, etc.).

***How to implement session affinity***

*For HTTP/S traffic, session affinity can be achieved through the "sticky sessions" feature of ELB[4]. Elastic Load Balancing will attempt to use the same server for that user for the duration of the session.*

*Another option, if you control the code that runs on the client, is to use client-side load balancing. This adds extra complexity but can be useful in scenarios where a load balancer does not meet your requirements. For example you might be using a protocol not supported by ELB or you might need full control on how users are assigned to servers (e.g., in a gaming scenario you might need to make sure game participants are matched and connect to the same server). In this model, the clients need a way of discovering valid server endpoints to directly connect to. You can use DNS for that, or you can build a simple discovery API to provide that information to the software running on the client. In the absence of a load balancer, the health checking mechanism will also need to be implemented on the client side. You should design your*

*client logic so that when server unavailability is detected, devices reconnect to another server with little disruption for the application.*

### *Distributed Processing*

Use cases that involve processing of very large amounts of data (e.g., anything that can't be handled by a single compute resource in a timely manner) require a distributed processing approach. By dividing a task and its data into many small fragments of work, you can execute each of them in any of a larger set of available compute resources.

***How to implement distributed processing***

*Offline batch jobs can be horizontally scaled by using a distributed data processing engine like Apache Hadoop. On AWS, you can use the Amazon Elastic MapReduce (Amazon EMR) service to run Hadoop workloads on top of a fleet of EC2 instances without the operational complexity. For real-time processing of streaming data, Amazon Kinesis partitions data in multiple shards that can then be consumed by multiple Amazon EC2 or AWS Lambda resources to achieve scalability.*

For more information on these types of workloads, you can refer to the "Big Data Analytics Options on AWS" whitepaper[5].

## Disposable Resources Instead of Fixed Servers

In a traditional infrastructure environment, you have to work with fixed resources due to the upfront cost and lead time of introducing new hardware. This would drive practices like manually logging in to servers to configure software or fix issues, hardcoding IP addresses, running tests or processing jobs sequentially etc.

When designing for AWS you have the opportunity to reset that mindset so that you take advantage of the dynamically provisioned nature of cloud computing. You can think of servers and other components as temporary resources. You can launch as many as you need, and use them only for as long as you need them.

Another issue with fixed, long-running servers is that of *configuration drift*. Changes and software patches applied through time can result in untested and heterogeneous configurations across different environments. This problem can be solved with the *immutable infrastructure* pattern. With this approach a server, once launched, is never updated throughout its lifetime. Instead, when

there is a problem or a need for an update the server is replaced with a new one that has the latest configuration. In this way, resources are always in a consistent (and tested) state and rollbacks become easier to perform.

## Instantiating Compute Resources

Whether you are deploying a new environment for testing, or increasing capacity of an existing system to cope with extra load, you will not want to manually set up new resources with their configuration and code. It is important that you make this an automated and repeatable process that avoids long lead times and is not prone to human error.

There are a few approaches on how to achieve an automated and repeatable process.

### *Bootstrapping*

When you launch an AWS resource like an Amazon EC2 instance or Amazon Relational Database (Amazon RDS) DB instance, you start with a default configuration. You can then execute automated bootstrapping actions. That is, scripts that install software or copy data to bring that resource to a particular state. You can parameterize configuration details that vary between different environments (e.g., production, test, etc.) so that the same scripts can be reused without modifications.

> ***Bootstrapping in practice***
>
> *You can use user data scripts and cloud-init[6] directives or AWS OpsWorks lifecycle events[7] to automatically set up new EC2 instances. You can use simple scripts, configuration management tools like Chef or Puppet. AWS OpsWorks natively supports Chef recipes or Bash/PowerShell scripts. In addition, through custom scripts and the AWS APIs, or through the use of AWS CloudFormation support for AWS Lambda-backed custom resources[8], it is possible to write provisioning logic that acts on almost any AWS resource.*

### *Golden Images*

Certain AWS resource types like Amazon EC2 instances, Amazon RDS DB instances, Amazon Elastic Block Store (Amazon EBS) volumes, etc., can be launched from a golden image: a snapshot of a particular state of that resource. When compared to the bootstrapping approach, a golden image results in faster start times and removes dependencies to configuration services or third-party repositories. This is important in auto-scaled environments where you want to be

able to quickly and reliably launch additional resources as a response to demand changes.

You can customize an Amazon EC2 instance and then save its configuration by creating an Amazon Machine Image (AMI)[9]. You can launch as many instances from the AMI as you need, and they will all include those customizations that you've made. Each time you want to change your configuration you will need to create a new golden image, so you will need to have a versioning convention to manage your golden images over time. We recommend that you use a script to bootstrap the EC2 instances that you use to create your AMIs. This will give you a flexible way to test and modify those images through time.

Alternatively, if you have an existing on-premises virtualized environment, you can use VM Import/Export from AWS to convert a variety of virtualization formats to an AMI. You can also find and use prebaked shared AMIs provided either by AWS or third parties in the AWS Community AMI catalog or the AWS Marketplace.

While golden images are most commonly used when launching new EC2 instances, they can also be applied to resources like Amazon RDS databases or Amazon EBS volumes. For example, when launching a new test environment you might want to prepopulate its database by instantiating it from a specific Amazon RDS snapshot, instead of importing the data from a lengthy SQL script.

> ***Containers***
>
> *Another option popular with developers is Docker—an open-source technology that allows you to build and deploy distributed applications inside software containers. Docker allows you to package a piece of software in a Docker Image, which is a standardized unit for software development, containing everything the software needs to run: code, runtime, system tools, system libraries, etc. AWS Elastic Beanstalk and the Amazon EC2 Container Service (Amazon ECS) support Docker and enable you to deploy and manage multiple Docker containers across a cluster of Amazon EC2 instances.*

### *Hybrid*

It is possible to use a combination of the two approaches, where some parts of the configuration are captured in a golden image, while others are configured dynamically through a bootstrapping action.

> ***The line between bootstrapping and golden image***
>
> *Items that do not change often or that introduce external dependencies will typically be part of your golden image. For example, your web server software that would otherwise have to be downloaded by a third-party repository each time you launch an instance is a good candidate.*
>
> *Items that change often or differ between your various environments can be set up dynamically through bootstrapping actions. For example, if you are deploying new versions of your application frequently, creating a new AMI for each application version might be impractical. You would also not want to hard code the database hostname configuration to your AMI because that would be different between the test and production environments. User data or tags can be used to allow you to use more generic AMIs that can be modified at launch time. For example, if you run web servers for various small businesses, they can all use the same AMI and retrieve their content from an Amazon S3 bucket location you specify in the user data at launch.*

AWS Elastic Beanstalk follows the hybrid model. It provides preconfigured run time environments (each initiated from its own AMI[10]) but allows you to run bootstrap actions (through configuration files called .ebextensions[11]) and configure environmental variables to parameterize the environment differences.

For a more detailed discussion of the different ways you can manage deployments of new resources please refer to the Overview of Deployment Options on AWS and Managing Your AWS Infrastructure at Scale whitepapers.

## Infrastructure as Code

The application of the principles we have discussed does not have to be limited to the individual resource level. Since AWS assets are programmable, you can apply techniques, practices, and tools from software development to make your whole infrastructure reusable, maintainable, extensible, and testable.

> ***AWS CloudFormation*** *templates give developers and systems administrators an easy way to create and manage a collection of related AWS resources, and provision and update them in an orderly and predictable fashion. You can describe the AWS resources, and any associated dependencies or run time parameters, required to run your application. Your CloudFormation templates can live with your application in your version control repository, allowing architectures to be reused and production environments to be reliably cloned for testing.*

## Automation

In a traditional IT infrastructure, you would often have to manually react to a variety of events. When deploying on AWS there is a lot of opportunity for automation, so that you improve both your system's stability and the efficiency of your organization:

- **AWS Elastic Beanstalk**[12] is the fastest and simplest way to get an application up and running on AWS. Developers can simply upload their application code and the service automatically handles all the details, such as resource provisioning, load balancing, auto scaling, and monitoring.

- **Amazon EC2 Auto recovery**[13]: You can create an Amazon CloudWatch alarm that monitors an Amazon EC2 instance and automatically recovers it if it becomes impaired. A recovered instance is identical to the original instance, including the instance ID, private IP addresses, Elastic IP addresses, and all instance metadata. However, this feature is only available for applicable instance configurations. Please refer to the Amazon EC2 documentation for an up-to-date description of those preconditions. In addition, during instance recovery, the instance is migrated through an instance reboot, and any data that is in-memory is lost.

- **Auto Scaling**[14]: With Auto Scaling, you can maintain application availability and scale your Amazon EC2 capacity up or down automatically according to conditions you define. You can use Auto Scaling to help ensure that you are running your desired number of healthy Amazon EC2 instances across multiple Availability Zones. Auto Scaling can also automatically increase the number of Amazon EC2 instances during demand spikes to maintain performance and decrease capacity during less busy periods to optimize costs.

- **Amazon CloudWatch Alarms**[15]: You can create a CloudWatch alarm that sends an Amazon Simple Notification Service (Amazon SNS) message when a particular metric goes beyond a specified threshold for a specified number of periods. Those Amazon SNS messages can automatically kick off the execution of a subscribed AWS Lambda function, enqueue a notification message to an Amazon SQS queue, or perform a POST request to an HTTP/S endpoint.

- **Amazon CloudWatch Events**[16]: The CloudWatch service delivers a near real-time stream of system events that describe changes in AWS resources. Using simple rules that you can set up in a couple of minutes, you can easily route each type of event to one or more targets: AWS Lambda functions, Amazon Kinesis streams, Amazon SNS topics, etc.

- **AWS OpsWorks Lifecycle events**[17]: AWS OpsWorks supports continuous configuration through lifecycle events that automatically update your instances' configuration to adapt to environment changes. These events can be used to trigger Chef recipes on each instance to perform specific configuration tasks. For example, when a new instance is successfully added to a Database server layer, the configure event could trigger a Chef recipe that updates the Application server layer configuration to point to the new database instance.

- **AWS Lambda Scheduled events**[18]: These events allow you to create a Lambda function and direct AWS Lambda to execute it on a regular schedule.

## Loose Coupling

As application complexity increases, a desirable attribute of an IT system is that it can be broken into smaller, loosely coupled components. This means that IT systems should be designed in a way that reduces interdependencies—a change or a failure in one component should not cascade to other components.

### Well-Defined Interfaces

A way to reduce interdependencies in a system is to allow the various components to interact with each other only through specific, technology-agnostic interfaces (e.g., RESTful APIs). In that way, technical implementation detail is hidden so that teams can modify the underlying implementation without affecting other components. As long as those interfaces maintain backwards compatibility, deployments of difference components are decoupled.

> ***Amazon API Gateway*** *is a fully managed service that makes it easy for developers to create, publish, maintain, monitor, and secure APIs at any scale. It handles all the tasks involved in accepting and processing up to hundreds of thousands of concurrent API calls, including traffic management, authorization and access control, monitoring, and API version management.*

### Service Discovery

Applications that are deployed as a set of smaller services will depend on the ability of those services to interact with each other. Because each of those services could be running across multiple compute resources there needs to be a way for each service to be addressed. For example, in a traditional infrastructure if your front end web service needed to connect with your back end web service, you could hardcode the IP address of the compute resource where this service was running. Although this approach can still work on cloud computing, if those

services are meant to be loosely coupled, they should be able to be consumed without prior knowledge of their network topology details. Apart from hiding complexity, this also allows infrastructure details to change at any time. Loose coupling is a crucial element if you want to take advantage of the elasticity of cloud computing where new resources can be launched or terminated at any point in time. In order to achieve that you will need some way of implementing service discovery.

***How to implement service discovery***

*For an Amazon EC2 hosted service a simple way to achieve service discovery is through the Elastic Load Balancing service. Because each load balancer gets its own hostname you now have the ability to consume a service through a stable endpoint. This can be combined with DNS and private Amazon Route53 zones, so that even the particular load balancer's endpoint can be abstracted and modified at any point in time.*

*Another option would be to use a service registration and discovery method to allow retrieval of the endpoint IP addresses and port number of any given service. Because service discovery becomes the glue between the components, it is important that it is highly available and reliable. If load balancers are not used, service discovery should also cater for things like health checking. Example implementations include custom solutions using a combination of tags, a highly available database and custom scripts that call the AWS APIs, or open source tools like Netflix Eureka, Airbnb Synapse, or HashiCorp Consul.*

## Asynchronous Integration

Asynchronous integration is another form of loose coupling between services. This model is suitable for any interaction that does not need an immediate response and where an acknowledgement that a request has been registered will suffice. It involves one component that generates events and another that consumes them. The two components do not integrate through direct point-to-point interaction but usually through an intermediate durable storage layer (e.g., an Amazon SQS queue or a streaming data platform like Amazon Kinesis).

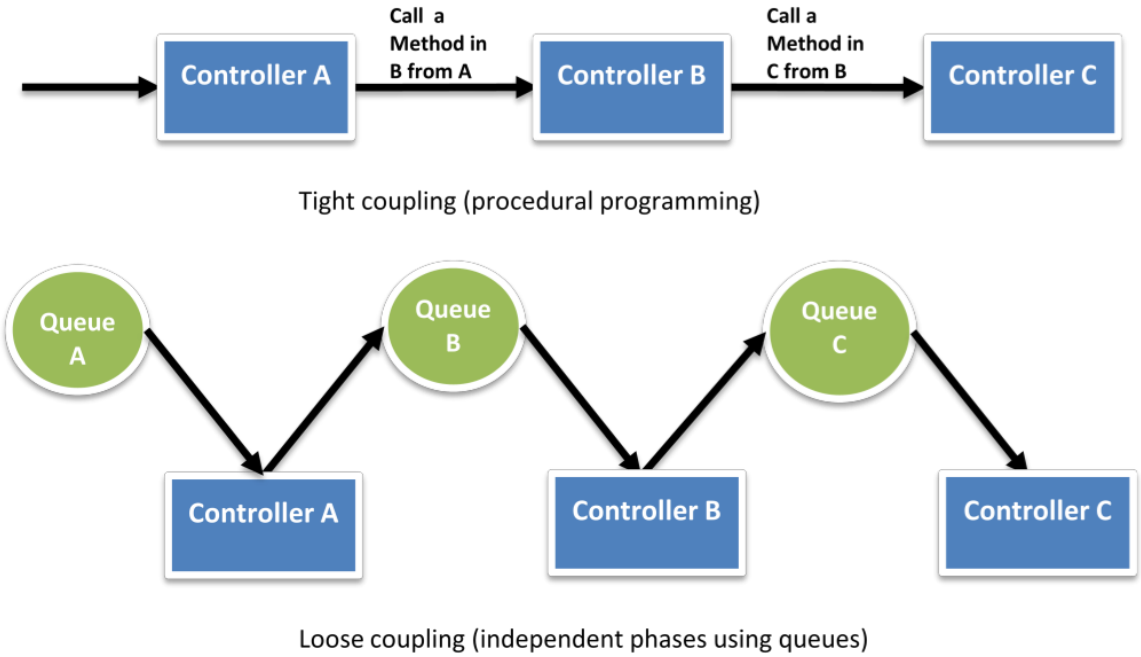


**Figure 1: Tight and Loose Coupling**

This approach decouples the two components and introduces additional resiliency. So, for example, if a process that is reading messages from the queue fails, messages can still be added to the queue to be processed when the system recovers. It also allows you to protect a less scalable back end service from front end spikes and find the right tradeoff between cost and processing lag. For example, you can decide that you don't need to scale your database to accommodate for an occasional peak of write queries as long as you eventually process those queries asynchronously with some delay. Finally, by moving slow operations off of interactive request paths you can also improve the end-user experience.

***Examples of asynchronous integration***

- *A front end application inserts jobs in a queue system like Amazon SQS. A back-end system retrieves those jobs and processes them at its own pace.*
- *An API generates events and pushes them into Amazon Kinesis streams. A back-end application processes these events in batches to create aggregated time-series data stored in a database.*
- *Multiple heterogeneous systems use Amazon SWF to communicate the flow of work between them without directly interacting with each other.*
- *AWS Lambda functions can consume events from a variety of AWS sources (e.g., Amazon DynamoDB update streams, Amazon S3 event notifications, etc.). In this case, you don't even need to worry about implementing a queuing or other asynchronous integration method because the service handles this for you.*

### Graceful Failure

Another way to increase loose coupling is to build applications in such a way that they handle component failure in a graceful manner. You can identify ways to reduce the impact to your end users and increase your ability to make progress on your offline procedures, even in the event of some component failure.

***Graceful failure in practice***

*A request that fails can be retried with an exponential backoff and Jitter strategy[19] or it could be stored in a queue for later processing. For front-end interfaces, it might be possible to provide alternative or cached content instead of failing completely when, for example, your database server becomes unavailable. The Amazon Route 53 DNS failover feature also gives you the ability to monitor your website and automatically route your visitors to a backup site if your primary site becomes unavailable. You can host your backup site as a static website on Amazon S3 or as a separate dynamic environment.*

## Services, Not Servers

Developing, managing, and operating applications—especially at scale—requires a wide variety of underlying technology components. With traditional IT infrastructure, companies would have to build and operate all those components.

AWS offers a broad set of compute, storage, database, analytics, application, and deployment services that help organizations move faster and lower IT costs.

Architectures that do not leverage that breadth (e.g., if they use only Amazon EC2) might not be making the most of cloud computing and might be missing an opportunity to increase developer productivity and operational efficiency.

## Managed Services

On AWS, there is a set of services that provide building blocks that developers can consume to power their applications. These managed services include databases, machine learning, analytics, queuing, search, email, notifications, and more. For example, with the Amazon Simple Queue Service (Amazon SQS) you can offload the administrative burden of operating and scaling a highly available messaging cluster, while paying a low price for only what you use. Not only that, Amazon SQS is inherently scalable. The same applies to Amazon S3 where you can store as much data as you want and access it when needed without having to think about capacity, hard disk configurations, replication, etc. In addition, Amazon S3 can also serve static assets of a web or mobile app, providing a highly available hosting solution that can scale automatically to meet traffic demands.

There are many other examples such as Amazon CloudFront for content delivery, ELB for load balancing, Amazon DynamoDB for NoSQL databases, Amazon CloudSearch for search workloads, Amazon Elastic Transcoder for video encoding, Amazon Simple Email Service (Amazon SES) for sending and receiving emails, and more[20].

## Serverless Architectures

Another approach that can reduce the operational complexity of running applications is that of the serverless architectures. It is possible to build both event-driven and synchronous services for mobile, web, analytics, and the Internet of Things (IoT) without managing any server infrastructure. These architectures can reduce costs because you are not paying for underutilized servers, nor are you provisioning redundant infrastructure to implement high availability.

> *You can upload your code to the **AWS Lambda** compute service and the service can run the code on your behalf using AWS infrastructure. With AWS Lambda, you are charged for every 100ms your code executes and the number of times your code is triggered. By using Amazon API Gateway, you can develop virtually infinitely scalable synchronous APIs powered by AWS Lambda. When combined with Amazon S3 for serving static content assets, this pattern can deliver a complete web application. For more details on this*

*type of architecture, please refer to the "AWS Serverless Multi-Tier Architectures" whitepaper[21].*

When it comes to mobile apps, there is one more way to reduce the surface of a server-based infrastructure. You can utilize Amazon Cognito, so that you don't have to manage a back end solution to handle user authentication, network state, storage, and sync. Amazon Cognito generates unique identifiers for your users. Those can be referenced in your access policies to enable or restrict access to other AWS resources on a per-user basis. Amazon Cognito provides temporary AWS credentials to your users, allowing the mobile application running on the device to interact directly with AWS Identity and Access Management (IAM)-protected AWS services. For example, using IAM you could restrict access to a folder within an Amazon S3 bucket to a particular end user.

For IoT applications, traditionally organizations have had to provision, operate, scale, and maintain their own servers as device gateways to handle the communication between connected devices and their services. AWS IoT provides a fully managed device gateway that scales automatically with your usage, without any operational overhead for you.

## Databases

With traditional IT infrastructure, organizations were often limited to the database and storage technologies they could use. There could be constraints based on licensing costs and the ability to support diverse database engines. On AWS, these constraints are removed by managed database services that offer enterprise performance at open source cost. As a result, it is not uncommon for applications to run on top of a polyglot data layer choosing the right technology for each workload.

***Determining the right database technology for each workload***

*The following questions can help you take decisions on which solutions to include in your architecture:*

- *Is this a read-heavy, write-heavy, or balanced workload? How many reads and writes per second are you going to need? How will those values change if the number of users increases?*
- *How much data will you need to store and for how long? How quickly do you foresee this will grow? Is there an upper limit in the foreseeable*

*future? What is the size of each object (average, min, max)? How are these objects going to be accessed?*

- *What are the requirements in terms of durability of data? Is this data store going to be your "source of truth"?*
- *What are your latency requirements? How many concurrent users do you need to support?*
- *What is your data model and how are you going to query the data? Are your queries relational in nature (e.g., JOINs between multiple tables)? Could you denormalize your schema to create flatter data structures that are easier to scale?*
- *What kind of functionality do you require? Do you need strong integrity controls or are you looking for more flexibility (e.g., schema-less data stores)? Do you require sophisticated reporting or search capabilities? Are your developers more familiar with relational databases than NoSQL?*

This section discusses the different categories of database technologies for you to consider.

## Relational Databases

Relational databases (often called RDBS or SQL databases) normalize data into well-defined tabular structures known as tables, which consist of rows and columns. They provide a powerful query language, flexible indexing capabilities, strong integrity controls, and the ability to combine data from multiple tables in a fast and efficient manner. Amazon Relational Database Service (Amazon RDS) makes it easy to set up, operate, and scale a relational database in the cloud.

### *Scalability*

Relational databases can scale vertically (e.g., by upgrading to a larger Amazon RDS DB instance or adding more and faster storage). In addition, consider the use of Amazon RDS for Aurora, which is a database engine designed to deliver much higher throughput compared to standard MySQL running on the same hardware. For read-heavy applications, you can also horizontally scale beyond the capacity constraints of a single DB instance by creating one or more read replicas.

> ***How to take advantage of read replicas***
>
> *Read replicas are separate database instances that are replicated asynchronously. As a result, they are subject to replication lag and might be missing some of the latest transactions. Application designers need to consider which queries have tolerance to slightly stale data. Those queries can be executed on a read replica, while the rest should run on the primary node. Read replicas can also not accept any write queries.*

Relational database workloads that need to scale their write capacity beyond the constraints of a single DB instance require a different approach called *data partitioning* or *sharding*. With this model, data is split across multiple database schemas each running in its own autonomous primary DB instance. Although Amazon RDS removes the operational overhead of running those instances, sharding introduces some complexity to the application. The application's data access layer will need to be modified to have awareness of how data is split so that it can direct queries to the right instance. In addition, any schema changes will have to be performed across multiple database schemas so it is worth investing some effort to automate this process.

### *High Availability*

For any production relational database, we recommend the use of the Amazon RDS Multi-AZ deployment feature, which creates a synchronously replicated standby instance in a different Availability Zone (AZ). In case of failure of the primary node, Amazon RDS performs an automatic failover to the standby without the need for manual administrative intervention. When a failover is performed, there is a short period during which the primary node is not accessible. Resilient applications can be designed for Graceful Failure by offering reduced functionality (e.g., read-only mode by utilizing read replicas).

### *Anti-Patterns*

If your application primarily indexes and queries data with no need for joins or complex transactions (especially if you expect a write throughput beyond the constraints of a single instance) consider a NoSQL database instead. If you have large binary files (audio, video, and image), it will be more efficient to store the actual files in the Amazon Simple Storage Service (Amazon S3) and only hold the metadata for the files in your database.

For more detailed relational database best practices refer to the Amazon RDS documentation[22].

## NoSQL Databases

NoSQL is a term used to describe databases that trade some of the query and transaction capabilities of relational databases for a more flexible data model that seamlessly scales horizontally. NoSQL databases utilize a variety of data models, including graphs, key-value pairs, and JSON documents. NoSQL databases are widely recognized for ease of development, scalable performance, high availability, and resilience. Amazon DynamoDB is a fast and flexible NoSQL database[23] service for applications that need consistent, single-digit millisecond latency at any scale. It is a fully managed cloud database and supports both document and key-value store models.

### *Scalability*

NoSQL database engines will typically perform data partitioning and replication to scale both the reads and the writes in a horizontal fashion. They do this transparently without the need of having the data partitioning logic implemented in the data access layer of your application. Amazon DynamoDB in particular manages table partitioning for you automatically, adding new partitions as your table grows in size or as read- and write-provisioned capacity changes.

In order to make the most of Amazon DynamoDB scalability when designing your application, refer to the Amazon DynamoDB best practices[24] section of the documentation.

### *High Availability*

The Amazon DynamoDB service synchronously replicates data across three facilities in an AWS region to provide fault tolerance in the event of a server failure or Availability Zone disruption.

### *Anti-Patterns*

If your schema cannot be denormalized and your application requires joins or complex transactions, consider a relational database instead. If you have large binary files (audio, video, and image), consider storing the files in Amazon S3 and storing the metadata for the files in your database.

When migrating or evaluating which workloads to migrate from a relational database to DynamoDB you can refer to the “Best Practices for Migrating from RDBMS to DynamoDB”[25] whitepaper for more guidance.

## Data Warehouse

A data warehouse is a specialized type of relational database, optimized for analysis and reporting of large amounts of data. It can be used to combine transactional data from disparate sources (e.g., user behavior in a web application, data from your finance and billing system, CRM, etc.) making them available for analysis and decision-making.

Traditionally, setting up, running, and scaling a data warehouse has been complicated and expensive. On AWS, you can leverage Amazon Redshift, a managed data warehouse service that is designed to operate at less than a tenth the cost of traditional solutions.

### *Scalability*

Amazon Redshift achieves efficient storage and optimum query performance through a combination of massively parallel processing (MPP), columnar data storage, and targeted data compression encoding schemes. It is particularly suited to analytic and reporting workloads against very large data sets. The Amazon Redshift MPP architecture enables you to increase performance by increasing the number of nodes in your data warehouse cluster.

### *High Availability*

Amazon Redshift has multiple features that enhance the reliability of your data warehouse cluster. We recommend that you deploy production workloads in multi-node clusters in which data written to a node is automatically replicated to other nodes within the cluster. Data is also continuously backed up to Amazon S3. Amazon Redshift continuously monitors the health of the cluster and automatically re-replicates data from failed drives and replaces nodes as necessary. Refer to the Amazon Redshift FAQ[26] for more information.

### *Anti-Patterns*

Because Amazon Redshift is a SQL-based relational database management system (RDBMS), it is compatible with other RDBMS applications and business intelligence tools. Although Amazon Redshift provides the functionality of a typical RDBMS, including online transaction processing (OLTP) functions, it is not designed for these workloads. If you expect a high concurrency workload that generally involves reading and writing all of the columns for a small number of records at a time you should instead consider using Amazon RDS or Amazon DynamoDB.

### Search

Applications that require sophisticated search functionality will typically outgrow the capabilities of relational or NoSQL databases. A search service can be used to index and search both structured and free text format and can support functionality that is not available in other databases, such as customizable result ranking, faceting for filtering, synonyms, stemming, etc.

On AWS, you have the choice between Amazon CloudSearch and Amazon Elasticsearch Service (Amazon ES). On the one hand, Amazon CloudSearch is a managed service that requires little configuration and will scale automatically. On the other hand, Amazon ES offers an open source API and gives you more control over the configuration details. Amazon ES has also evolved to become a lot more than just a search solution. It is often used as an analytics engine for use cases such as log analytics, real-time application monitoring, and click stream analytics.

#### *Scalability*

Both Amazon CloudSearch and Amazon ES use data partitioning and replication to scale horizontally. The difference is that with Amazon CloudSearch you do not need to worry about the number of partitions and replicas you will need because the service handles all that automatically for you.

#### *High Availability*

Both services provide features that store data redundantly across Availability Zones. For details, please refer to each service's documentation.

## Removing Single Points of Failure

Production systems typically come with defined or implicit objectives in terms of uptime. A system is highly available when it can withstand the failure of an individual or multiple components (e.g., hard disks, servers, network links etc.). You can think about ways to automate recovery and reduce disruption at every layer of your architecture. This section discusses high availability design patterns. For more details on the subject, refer to the "Building Fault Tolerant Applications" whitepaper[27].

## Introducing Redundancy

Single points of failure can be removed by introducing redundancy, which is having multiple resources for the same task. Redundancy can be implemented in either standby or active mode.

In *standby redundancy* when a resource fails, functionality is recovered on a secondary resource using a process called *failover*. The failover will typically require some time before it completes, and during that period the resource remains unavailable. The secondary resource can either be launched automatically only when needed (to reduce cost), or it can be already running idle (to accelerate failover and minimize disruption). Standby redundancy is often used for stateful components such as relational databases.

In *active redundancy*, requests are distributed to multiple redundant compute resources, and when one of them fails, the rest can simply absorb a larger share of the workload. Compared to standby redundancy, it can achieve better utilization and affect a smaller population when there is a failure.

## Detect Failure

You should aim to build as much automation as possible in both detecting and reacting to failure. You can use services like ELB and Amazon Route53 to configure health checks and mask failure by routing traffic to healthy endpoints. In addition, Auto Scaling can be configured to automatically replace unhealthy nodes. You can also replace unhealthy nodes using the Amazon EC2 auto-recovery[28] feature or services such as AWS OpsWorks and AWS Elastic Beanstalk. It won't be possible to predict every possible failure scenario on day one. Make sure you collect enough logs and metrics to understand normal system behavior. After you understand that, you will be able to set up alarms for manual intervention or automated response.

***Designing good health checks***

*Configuring the right health checks for your application will determine your ability to respond correctly and promptly to a variety of failure scenarios. Specifying the wrong health check can actually reduce your application's availability.*

*In a typical three-tier application, you configure health checks on the Elastic Load Balancing service. Design your health checks with the objective of reliably assessing the health of the back end nodes. A simple TCP health*

*check would not detect the scenario where the instance itself is healthy, but the web server process has crashed. Instead, you should assess whether the web server can return a HTTP 200 response for some simple request.*

*At this layer, it might not be a good idea to configure what is called a deep health check, which is a test that depends on other layers of your application to be successful (this could result in false positives). For example, if your health check also assesses whether the instance can connect to a back end database, you risk marking all of your web servers as unhealthy when that database node becomes shortly unavailable. A layered approach is often the best. A deep health check might be appropriate to implement at the Amazon Route53 level. By running a more holistic check that determines if that environment is able to actually provide the required functionality, you can configure Amazon Route53 to failover to a static version of your website until your database is up and running again.*

## Durable Data Storage

Your application and your users will create and maintain a variety of data. It is crucial that your architecture protects both data availability and integrity. Data replication is the technique that introduces redundant copies of data. It can help horizontally scale read capacity, but it also increase data durability and availability. Replication can take place in a few different modes.

*Synchronous replication* only acknowledges a transaction after it has been durably stored in both the primary location and its replicas. It is ideal for protecting the integrity of data from the event of a failure of the primary node. Synchronous replication can also scale read capacity for queries that require the most up-to-date data (strong consistency). The drawback of synchronous replication is that the primary node is coupled to the replicas. A transaction can't be acknowledged before all replicas have performed the write. This can compromise performance and availability (especially in topologies that run across unreliable or high-latency network connections). For the same reason it is not recommended to maintain many synchronous replicas.

***Durability: No replacement for backups***

*Regardless of the durability of your solution, this is no replacement for backups. Synchronous replication will redundantly store all updates to your data—even those that are results of software bugs or human error. However, particularly for objects stored on Amazon S3, you can use versioning[29] to preserve, retrieve, and restore any of their versions. With versioning, you can recover from both unintended user actions and application failures.*

*Asynchronous replication* decouples the primary node from its replicas at the expense of introducing replication lag. This means that changes performed on the primary node are not immediately reflected on its replicas. Asynchronous replicas are used to horizontally scale the system's read capacity for queries that can tolerate that replication lag. It can also be used to increase data durability when some loss of recent transactions can be tolerated during a failover. For example, you can maintain an asynchronous replica of a database in a separate AWS region as a disaster recovery solution.

*Quorum-based replication* combines synchronous and asynchronous replication to overcome the challenges of large-scale distributed database systems. Replication to multiple nodes can be managed by defining a minimum number of nodes that must participate in a successful write operation. A detailed discussion of distributed data stores is beyond the scope of this document. You can refer to the Amazon Dynamo whitepaper[30] to learn more about a core set of principles that can result in an ultra-scalable and highly reliable database system.

***Data durability in practice***

*It is important to understand where each technology you are using fits in these data storage models. Their behavior during various failover or backup/restore scenarios should align to your recovery point objective (RPO) and your recovery time objective (RTO). You need to ascertain how much data you would expect to lose and how quickly you would be able to resume operations. For example, the Redis engine for Amazon ElastiCache supports replication with automatic failover, but the Redis engine's replication is asynchronous. During a failover, it is highly likely that some recent transactions would be lost. However, Amazon RDS, with its Multi AZ feature, is designed to provide synchronous replication to keep data on the standby node up-to-date with the primary.*

## Automated Multi-Data Center Resilience

Business critical applications will also need protection against disruption scenarios affecting a lot more than just a single disk, server, or rack. In a traditional infrastructure you would typically have a disaster recovery plan to allow a failover to a distant second data center, should there be a major disruption in the primary one. Because of the long distance between the two data centers, latency makes it impractical to maintain synchronous cross-data center copies of the data. As a result, a failover will most certainly lead to data loss or a very costly data recovery process. It becomes a risky and not always sufficiently tested procedure. Nevertheless, this is a model that provides excellent protection

against a low probability but huge impact risk—e.g., a natural catastrophe that brings down your whole infrastructure for a long time. You can refer to the AWS Disaster recovery whitepaper for more guidance on how to implement this approach on AWS[31].

A shorter interruption in a data center is a more likely scenario. For short disruptions, because the duration of the failure isn't predicted to be long, the choice to perform a failover is a difficult one and generally will be avoided. On AWS it is possible to adopt a simpler, more efficient protection from this type of failure. Each AWS region contains multiple distinct locations called Availability Zones. Each Availability Zone is engineered to be isolated from failures in other Availability Zones. An Availability Zone is a data center, and in some cases, an Availability Zone consists of multiple data centers. Availability Zones within a region provide inexpensive, low-latency network connectivity to other zones in the same region. This allows you to replicate your data across data centers in a synchronous manner so that failover can be automated and be transparent for your users.

It is also possible to implement active redundancy so that you don't pay for idle resources. For example, a fleet of application servers can be distributed across multiple Availability Zones and be attached to the Elastic Load Balancing service (ELB). When the Amazon EC2 instances of a particular Availability Zone fail their health checks, ELB will stop sending traffic to those nodes. When combined with Auto Scaling, the number of healthy nodes can automatically be rebalanced to the other Availability Zones with no manual intervention.

In fact, many of the higher level services on AWS are inherently designed according to the Multi-AZ principle. For example, Amazon RDS provides high availability and automatic failover support for DB instances using Multi-AZ deployments, while with Amazon S3 and Amazon DynamoDB your data is redundantly stored across multiple facilities.

### Fault Isolation and Traditional Horizontal Scaling

Though the active redundancy pattern is great for balancing traffic and handling instance or Availability Zone disruptions, it is not sufficient if there is something harmful about the requests themselves. For example, there could be scenarios where every instance is affected. If a particular request happens to trigger a bug that causes the system to fail over, then the caller may trigger a cascading failure by repeatedly trying the same request against all instances.

#### *Shuffle Sharding*

One fault-isolating improvement you can make to traditional horizontal scaling is called *sharding*. Similar to the technique traditionally used with data storage systems, instead of spreading traffic from all customers across every node, you can group the instances into shards. For example, if you have eight instances for your service, you might create four shards of two instances each (two instances for some redundancy within each shard) and distribute each customer to a specific shard. In this way, you are able to reduce the impact on customers in direct proportion to the number of shards you have. However, there will still be affected customers, so the key is to make the client fault tolerant. If the client can try every endpoint in a set of sharded resources, until one succeeds, you get a dramatic improvement. This technique is called *shuffle sharding* and is described in more detail in the relevant blog post[32].

## Optimize for Cost

Just by moving existing architectures into the cloud, organizations can reduce capital expenses and drive savings as a result of the AWS economies of scale. By iterating and making use of more AWS capabilities there is further opportunity to create cost-optimized cloud architectures. This section discusses the main principles of optimizing for cost with AWS cloud computing.

### Right Sizing

AWS offers a broad range of resource types and configurations to suit a plethora of use cases. For example, services like Amazon EC2, Amazon RDS, Amazon Redshift, and Amazon Elasticsearch Service (Amazon ES) give you a lot of choice of instance types. In some cases, you should select the cheapest type that suits your workload's requirements. In other cases, using fewer instances of a larger instance type might result in lower total cost or better performance. You should benchmark and select the right instance type depending on how your workload utilizes CPU, RAM, network, storage size, and I/O.

Similarly, you can reduce cost by selecting the right storage solution for your needs. For example, Amazon S3 offers a variety of storage classes, including Standard, Reduced Redundancy, and Standard-Infrequent Access. Other services, such as Amazon EC2, Amazon RDS, and Amazon ES support different Amazon Elastic Block Store (Amazon EBS) volume types (magnetic, general purpose SSD, provisioned IOPS SSD) that you should evaluate.

***Continuous monitoring and tagging***

*Cost optimization is an iterative process. Your application and its usage will evolve through time. In addition, AWS iterates frequently and regularly releases new options.*

*AWS provides tools[33] to help you identify those cost saving opportunities and keep your resources right-sized. To make those tools' outcomes easy to interpret you should define and implement a tagging policy for your AWS resources. You can make tagging a part of your build process and automate it with AWS management tools like AWS Elastic Beanstalk and AWS OpsWorks. You can also use the managed rules provided by AWS Config to assess whether specific tags are applied to your resources or not.*

## Elasticity

Another way you can save money with AWS is by taking advantage of the platform's elasticity. Plan to implement Auto Scaling for as many Amazon EC2 workloads as possible, so that you horizontally scale up when needed and scale down and automatically reduce your spend when you don't need all that capacity anymore. In addition, you can automate turning off non-production workloads when not in use[34]. Ultimately, consider which compute workloads you could implement on AWS Lambda so that you never pay for idle or redundant resources.

Where possible, replace Amazon EC2 workloads with AWS managed services that either don't require you to take any capacity decisions (e.g., ELB, Amazon CloudFront, Amazon SQS, Amazon Kinesis Firehose, AWS Lambda, Amazon SES, Amazon CloudSearch) or enable you to easily modify capacity as and when need (e.g., Amazon DynamoDB, Amazon RDS, Amazon Elasticsearch Service).

## Take Advantage of the Variety of Purchasing Options

Amazon EC2 On-Demand instance pricing gives you maximum flexibility with no long term commitments. There are two more ways to pay for Amazon EC2 instances that can help you reduce spend: Reserved Instances and Spot Instances.

### *Reserved Capacity*

Amazon EC2 Reserved Instances allow you to reserve Amazon EC2 computing capacity in exchange for a significantly discounted hourly rate compared to On-Demand instance pricing. This is ideal for applications with predictable minimum capacity requirements. You can take advantage of tools like the AWS

Trusted Advisor or Amazon EC2 usage reports to identify the compute resources that you use most of the time that you should consider reserving. Depending on your Reserved Instance purchases, the discounts will be reflected in the monthly bill. Note that there is technically no difference between an On-Demand EC2 instance and a Reserved Instance. The difference lies in the way you pay for instances that you reserve.

Reserved capacity options exist for other services as well (e.g., Amazon Redshift, Amazon RDS, Amazon DynamoDB, and Amazon CloudFront).

> ***Tip:*** *You should not commit to Reserved Instance purchases before sufficiently benchmarking your application in production. After you have purchased reserved capacity, you can use the Reserved Instance utilization reports to ensure you are still making the most of your reserved capacity.*

### *Spot Instances*

For less steady workloads, you can consider the use of Spot Instances. Amazon EC2 Spot Instances allow you to bid on spare Amazon EC2 computing capacity. Since Spot Instances are often available at a discount compared to On-Demand pricing, you can significantly reduce the cost of running your applications.

Spot Instances are ideal for workloads that have flexible start and end times. Your Spot Instance is launched when your bid exceeds the current Spot market price, and will continue run until you choose to terminate it, or until the Spot market price exceeds your bid. If the Spot market price increases above your bid price, your instance will be terminated automatically and you will not be charged for the partial hour that your instance has run.

As a result, Spot Instances are great for workloads that have tolerance to interruption. However, you can also use Spot Instances when you require more predictable availability:

**Bidding strategy:** You are charged the Spot market price (not your bid price) for as long as the Spot Instance runs. Your bidding strategy could be to bid much higher than that with the expectation that even if the market price occasionally spikes you would still be saving a lot of cost in the long term.

**Mix with On-Demand:** Consider mixing Reserved, On-Demand, and Spot Instances to combine a predictable minimum capacity with “opportunistic”

access to additional compute resources depending on the spot market price. This is a great way to improve throughput or application performance.

**Spot Blocks for Defined-Duration Workloads:** You can also bid for fixed duration Spot Instances. These have different hourly pricing but allow you to specify a duration requirement. If your bid is accepted your instance will continue to run until you choose to terminate it, or until the specified duration has ended; your instance will not be terminated due to changes in the Spot price (but of course, you should still design for fault tolerance because a Spot Instance can still fail like any other EC2 instance).

> ***Spot pricing best practices***
>
> *Spot Instances allow you to bid on multiple instance types simultaneously. Because prices fluctuate independently for each instance type in an Availability Zone, you can often get more compute capacity for the same price if your app is designed to be flexible about instance types. Test your application on different instance types when possible. Bid on all instance types that meet your requirements to further reduce costs.*

# Caching

Caching is a technique that stores previously calculated data for future use. This technique is used to improve application performance and increase the cost efficiency of an implementation. It can be applied at multiple layers of an IT architecture.

## Application Data Caching

Applications can be designed so that they store and retrieve information from fast, managed, in-memory caches. Cached information may include the results of I/O-intensive database queries or the outcome of computationally intensive processing. When the result set is not found in the cache, the application can calculate it or retrieve it from a database and store it in the cache for subsequent requests. When, however, a result set is found in the cache the application can use that directly, which improves latency for end users and reduces load on back end systems. Your application can control for how long each cached item will remain valid. In some cases, even a few seconds of caching for very popular objects can result in a dramatic decrease on the load for your database.

Amazon ElastiCache is a web service that makes it easy to deploy, operate, and scale an in-memory cache in the cloud. It supports two open-source in-memory

caching engines: Memcached and Redis. For more details on how to select the right engine for your workload, as well as a description of common ElastiCache design patterns please refer to the "Performance at Scale with Amazon ElastiCache"[35] whitepaper.

### Edge Caching

Copies of static content (e.g., images, css files, streaming of pre-recorded video) and dynamic content (e.g., html response, live video) can be cached at Amazon CloudFront, which is a content delivery network (CDN) consisting of multiple edge locations around the world. Edge caching allows content to be served by infrastructure that is closer to viewers, lowering latency and giving you the high, sustained data transfer rates needed to deliver large popular objects to end users at scale.

Requests for your content are carried back to Amazon S3 or your origin servers. If the origin is running on AWS then requests will be transferred over optimized network paths for a more reliable and consistent experience. Amazon CloudFront can be used to deliver your entire website, including non-cachable content. The benefit in that case is that Amazon CloudFront reuses existing connections between the Amazon CloudFront edge and the origin server reducing connection setup latency for each origin request. Other connection optimizations are also applied to avoid Internet bottlenecks and fully utilize available bandwidth between the edge location and the viewer. This means that Amazon CloudFront can speed-up the delivery of your dynamic content and provide your viewers with a consistent and reliable, yet personalized experience when navigating your web application. Amazon CloudFront also applies the same performance benefits to upload requests as those applied to the requests for downloading dynamic content.

## Security

Most of the security tools and techniques that you might already be familiar with in a traditional IT infrastructure can be used in the cloud. At the same time, AWS allows you to improve your security in a variety of ways. AWS is a platform that allows you to formalize the design of security controls in the platform itself. It simplifies system use for administrators and those running IT, and makes your environment much easier to audit in a continuous manner. This section gives you a high-level overview of AWS security best practices. For a detailed view on how you can achieve a high level of security governance please refer to the "Security at

Scale: Governance in AWS"[36] and the "AWS Security Best Practices"[37] whitepapers.

## Utilize AWS Features for Defense in Depth

AWS provides a wealth of features that can help architects build defense in depth. Starting at the network level you can build a VPC topology that isolates parts of the infrastructure through the use of subnets, security groups, and routing controls. Services like AWS WAF, a web application firewall, can help protect your web applications from SQL injection and other vulnerabilities in your application code. For access control, you can use IAM to define a granular set of policies and assign them to users, groups, and AWS resources. Finally, the AWS platform offers a breadth of options for protecting data, whether it is in transit or at rest with encryption[38]. An exhaustive list of all security features is beyond the scope of this document, but you can learn more by visiting the AWS Security page[39].

## Offload Security Responsibility to AWS

AWS operates under a shared security responsibility model, where AWS is responsible for the security of the underlying cloud infrastructure and you are responsible for securing the workloads you deploy in AWS. This way, you can reduce the scope of your responsibility and focus on your core competencies through the use of AWS managed services. For example, when you use services such as Amazon RDS, Amazon ElastiCache, Amazon CloudSearch, etc., security patches become the responsibility of AWS. This not only reduces operational overhead for your team, but it could also reduce your exposure to vulnerabilities.

## Reduce Privileged Access

When you treat servers as programmable resources, you can capitalize on that for benefits in the security space as well. When you can change your servers whenever you need to you can eliminate the need for guest operating system access to production environments. If an instance experiences an issue you can automatically or manually terminate and replace it. However, before you replace instances you should collect and centrally store logs on your instances that can help you recreate issues in your development environment and deploy them as fixes through your continuous deployment process. This is particularly important in an elastic compute environment where servers are temporary. You can use Amazon CloudWatch Logs to collect this information. Where you don't have access and you need it, you can implement just-in-time access by using an API

action to open up the network for management only when necessary. You can integrate these requests for access with your ticketing system, so that access requests are tracked and dynamically handled only after approval.

Another common source of security risk is the use of service accounts. In a traditional environment, service accounts would often be assigned long-term credentials stored in a configuration file. On AWS, you can instead use IAM roles to grant permissions to applications running on Amazon EC2 instances through the use of short-term credentials. Those credentials are automatically distributed and rotated. For mobile applications, the use of Amazon Cognito allows client devices to get controlled access to AWS resources via temporary tokens. For AWS Management Console users you can similarly provide federated access through temporary tokens instead of creating IAM users in your AWS account. In that way, an employee who leaves your organization and is removed from your organization's identity directory will also lose access to your AWS account.

## Security as Code

Traditional security frameworks, regulations, and organizational policies define security requirements related to things such as firewall rules, network access controls, internal/external subnets, and operating system hardening. You can implement these in an AWS environment as well, but you now have the opportunity to capture them all in a script that defines a "Golden Environment." This means you can create an AWS CloudFormation script that captures your security policy and reliably deploys it. Security best practices can now be reused among multiple projects and become part of your continuous integration pipeline. You can perform security testing as part of your release cycle, and automatically discover application gaps and drift from your security policy.

Additionally, for greater control and security, AWS CloudFormation templates can be imported as "products" into AWS Service Catalog[40]. This enables centralized management of resources to support consistent governance, security, and compliance requirements, while enabling users to quickly deploy only the approved IT services they need. You apply IAM permissions to control who can view and modify your products, and you define constraints to restrict the ways that specific AWS resources can be deployed for a product.

### Real-Time Auditing

Testing and auditing your environment is key to moving fast while staying safe. Traditional approaches that involve periodic (and often manual or sample-based) checks are not sufficient, especially in agile environments where change is constant. On AWS, it is possible to implement continuous monitoring and automation of controls to minimize exposure to security risks. Services like AWS Config, Amazon Inspector, and AWS Trusted Advisor continually monitor for compliance or vulnerabilities giving you a clear overview of which IT resources are in compliance, and which are not. With AWS Config rules you will also know if some component was out of compliance even for a brief period of time, making both point-in-time and period-in-time audits very effective. You can implement extensive logging for your applications (using Amazon CloudWatch Logs) and for the actual AWS API calls by enabling AWS CloudTrail[41]. AWS CloudTrail is a web service that records API calls to supported AWS services in your AWS account and delivers a log file to your Amazon S3 bucket. Logs can then be stored in an immutable manner and automatically processed to either notify or even take action on your behalf, protecting your organization from non-compliance. You can use AWS Lambda, Amazon EMR, the Amazon Elasticsearch Service, or third-party tools from the AWS Marketplace to scan logs to detect things like unused permissions, overuse of privileged accounts, usage of keys, anomalous logins, policy violations, and system abuse.

## Conclusion

This whitepaper provides guidance for designing architectures that make the most of the AWS platform by covering important principles and design patterns: from how to select the right database for your application, to architecting applications that can scale horizontally and with high availability. As each use case is unique, you will have to evaluate how those can be applied to your implementation. The topic of cloud computing architectures is broad and continuously evolving. Going forward you can stay updated through the wealth of material available on the AWS website and the training and certification offerings of AWS.

# Contributors

The following individual contributed to this document:

- Andreas Chatzakis, Manager, AWS Solutions Architecture

# Further Reading

For more architecture examples, you can refer to the AWS Architecture Center[42]. For applications already running on AWS we recommend you also go through the "AWS Well Architected Framework" whitepaper[43] that complements this document by providing a structured evaluation model. Finally, to validate your operational readiness you can also refer to the comprehensive AWS Operational Checklist[44].

# Notes

[1] About AWS: https://aws.amazon.com/about-aws/

[2] The AWS global infrastructure: https://aws.amazon.com/about-aws/global-infrastructure/

[3] For example there is the PHP Amazon DynamoDB session handler (http://docs.aws.amazon.com/aws-sdk-php/v3/guide/service/dynamodb-session-handler.html) and the Tomcat Amazon DynamoDB session handler (http://docs.aws.amazon.com/AWSSdkDocsJava/latest//DeveloperGuide/java-dg-tomcat-session-manager.html)

[4] ELB sticky sessions http://docs.aws.amazon.com/ElasticLoadBalancing/latest/DeveloperGuide/elb-sticky-sessions.html

[5] "Big Data Analytics Options on AWS" whitepaper https://do.awsstatic.com/whitepapers/Big_Data_Analytics_Options_on_AWS.pdf

[6] Bootstrapping with user data scripts and cloud-init: http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-metadata.html

[7] AWS Opsworks Lifecycle events http://docs.aws.amazon.com/opsworks/latest/userguide/workingcookbook-events.html

[8] AWS Lambda-backed custom CloudFormation resources: http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/template-custom-resources-lambda.html

[9] Amazon Machine Images http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html

[10] AMIs for the AWS Elastic Beanstalk run times: http://docs.aws.amazon.com/elasticbeanstalk/latest/dg/concepts.platforms.html

[11] AWS Elastic Beanstalk customization with configuration files: http://docs.aws.amazon.com/elasticbeanstalk/latest/dg/ebextensions.html

[12] AWS Elastic Beanstalk: https://aws.amazon.com/elasticbeanstalk/

[13] Amazon EC2 auto recovery: http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-recover.html

[14] Auto Scaling: https://aws.amazon.com/autoscaling/

[15] Amazon CloudWatch alarms: http://docs.aws.amazon.com/AmazonCloudWatch/latest/DeveloperGuide/AlarmThatSendsEmail.html

[16] Amazon CloudWatch events: http://docs.aws.amazon.com/AmazonCloudWatch/latest/DeveloperGuide/WhatIsCloudWatchEvents.html

[17] AWS OpsWorks lifecycle http://docs.aws.amazon.com/opsworks/latest/userguide/workingcookbook-events.html

[18] AWS Lambda scheduled events: http://docs.aws.amazon.com/lambda/latest/dg/with-scheduled-events.html

[19] Exponential Backoff and Jitter http://www.awsarchitectureblog.com/2015/03/backoff.html

[20] You can see the full list of AWS products here: http://aws.amazon.com/products/

[21] "AWS Serverless Multi-Tier Architectures" whitepaper https://d0.awsstatic.com/whitepapers/AWS_Serverless_Multi-Tier_Archiectures.pdf

[22] Best Practices for Amazon RDS: http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/CHAP_BestPractices.html

[23] NoSQL databases on AWS https://aws.amazon.com/nosql/

[24] Best practices for Amazon DynamoDB: http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/BestPractices.html

[25] Best practices for Migrating from RDBMS to Amazon DynamoDB: https://d0.awsstatic.com/whitepapers/migration-best-practices-rdbms-to-dynamodb.pdf

[26] Amazon Redshift FAQ: https://aws.amazon.com/redshift/faqs/

[27] "Building Fault Tolerant Applications" whitepaper: https://d0.awsstatic.com/whitepapers/aws-building-fault-tolerant-applications.pdf

[28] Recover your instance: http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-recover.html

[29] Amazon S3 versioning: http://docs.aws.amazon.com/AmazonS3/latest/dev/Versioning.html

[30] "Dynamo: Amazon's Highly Available Key-value Store"http://www.allthingsdistributed.com/2007/10/amazons_dynamo.html

[31] "Using Amazon Web Services for Disaster Recovery" https://media.amazonwebservices.com/AWS_Disaster_Recovery.pdf

[32] Shuffle sharding http://www.awsarchitectureblog.com/2014/04/shuffle-sharding.html

[33] Monitoring Your Usage and Costs http://docs.aws.amazon.com/awsaccountbilling/latest/aboutv2/monitoring-costs.html

[34] Create Alarms that stop or terminate an instance http://docs.aws.amazon.com/AmazonCloudWatch/latest/DeveloperGuide/UsingAlarmActions.html

[35] "Performance at Scale with Amazon ElastiCache:" https://d0.awsstatic.com/whitepapers/performance-at-scale-with-amazon-elasticache.pdf

[36] "Security at Scale: Governance in AWS" https://d0.awsstatic.com/whitepapers/compliance/AWS_Security_at_Scale_Governance_in_AWS_Whitepaper.pdf

[37] "AWS Security Best Practices": https://d0.awsstatic.com/whitepapers/aws-security-best-practices.pdf

[38] Securing data at rest with encryption: https://d0.awsstatic.com/whitepapers/aws-securing-data-at-rest-with-encryption.pdf

[39] AWS Security: http://aws.amazon.com/security

[40] AWS Service Catalog: https://aws.amazon.com/servicecatalog/

[41] "Security at Scale: Logging in AWS" https://do.awsstatic.com/whitepapers/compliance/AWS_Security_at_Scale_Logging_in_AWS_Whitepaper.pdf

[42] AWS Architecture Center https://aws.amazon.com/architecture

[43] "AWS Well Architected Framework": http://do.awsstatic.com/whitepapers/architecture/AWS_Well-Architected_Framework.pdf

[44] AWS Operational Checklist http://media.amazonwebservices.com/AWS_Operational_Checklists.pdf