SWINBURNE UNIVERSITY OF TECHNOLOGY

Cloud Computing Architecture

Lecture 08
- Highly Available Applications
- Automating Infrastructure

includes material from
ACA Module 3 & 4: High Availability – Section I & II
ACA Module 5: Automating your Infrastructure

Reminders

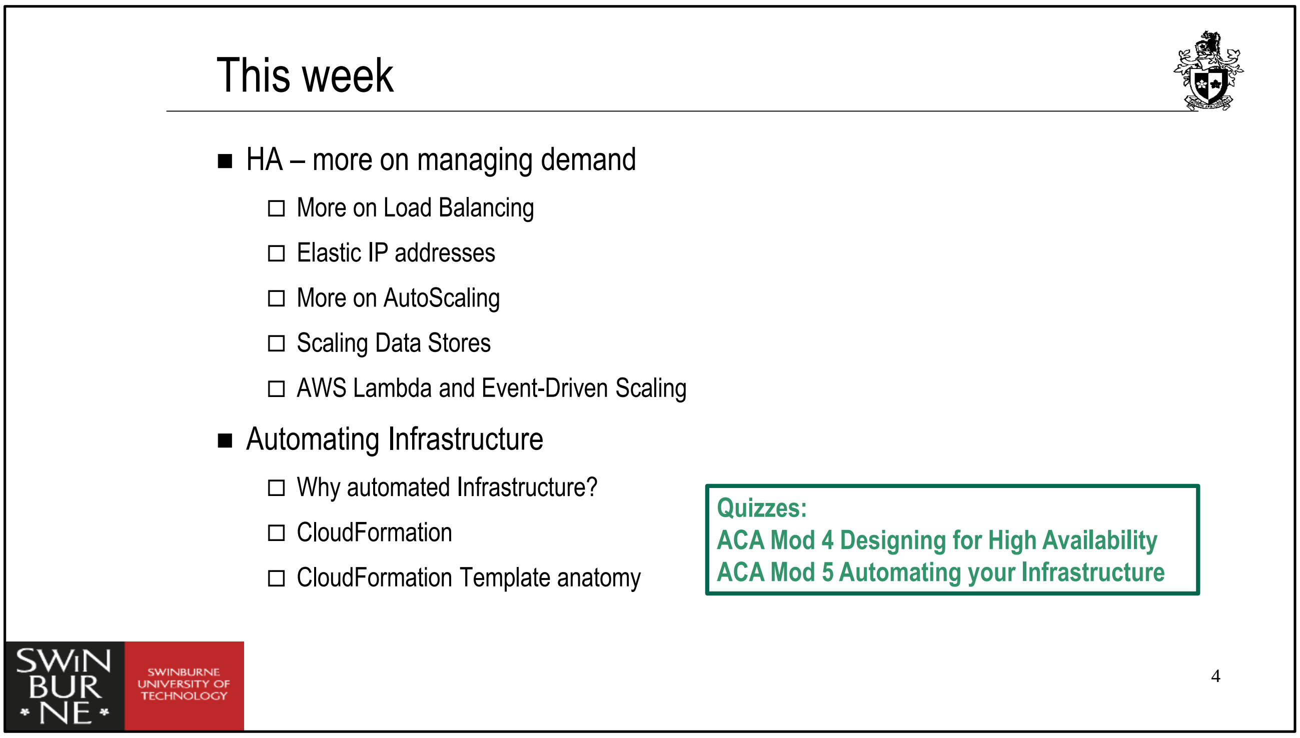
- **Assignment 2 - Due date: Thursday 8th October 9 am**
  - Implement on specified Vocareum classroom (SDCC - Assignments)
  - Due to Canvas
    Late submission penalty: 10% of total available marks per day.
  - Discussion board provided
    - Feel free to ask and answer questions (don't post code)
- **Assignment 3 now released**

Last week – Scalable Architectures

- **High availability and Scaling**
- Automated Scaling and Monitoring on AWS
  - ☐Elastic Load Balancing
  - ☐Amazon CloudWatch
  - ☐Amazon EC2 Auto Scaling
- Labs
  - ☐ *ACF Lab 6*
  - ☐ACA Module 3

Quizzes:
ACF 2.0.5 Balancing, Scaling and Monitoring
ACA Mod 4 Designing for High Availability

SWINBURNE UNIVERSITY OF TECHNOLOGY



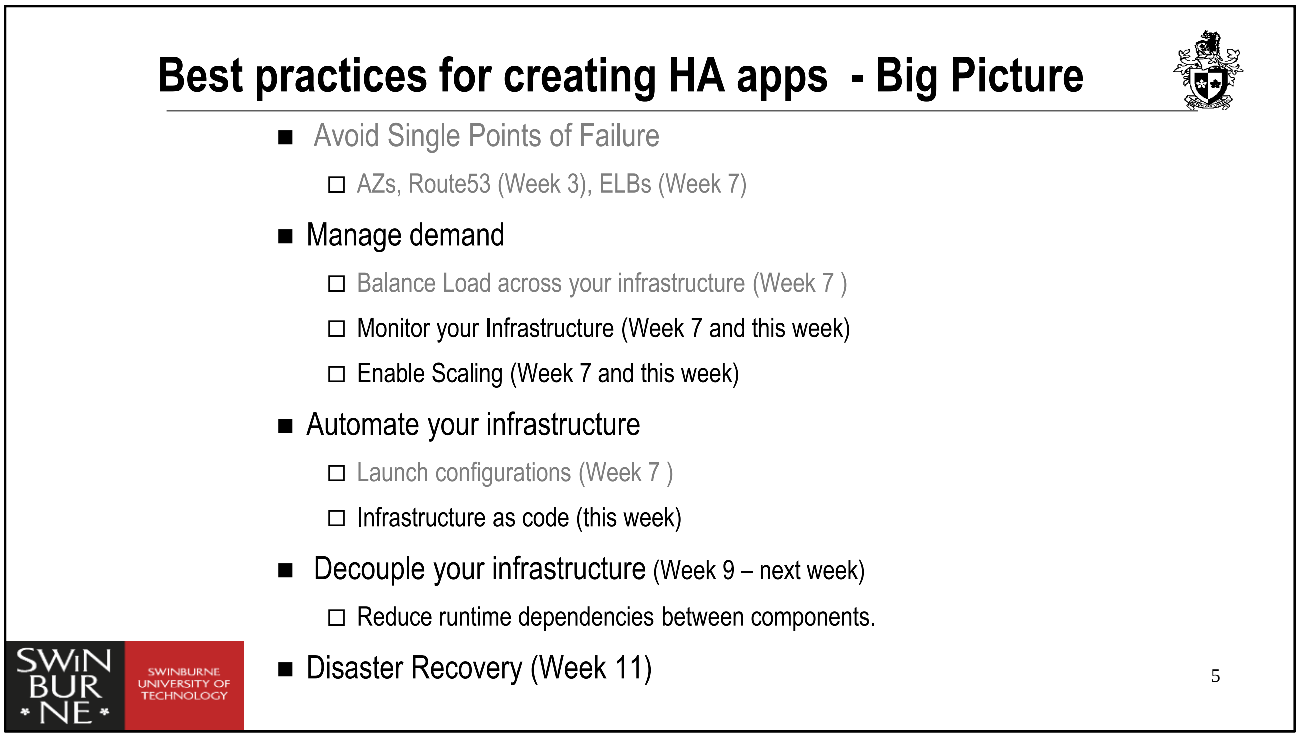
This week

- HA – more on managing demand
  - ☐ More on Load Balancing
  - ☐ Elastic IP addresses
  - ☐ More on AutoScaling
  - ☐ Scaling Data Stores
  - ☐ AWS Lambda and Event-Driven Scaling
- Automating Infrastructure
  - ☐ Why automated Infrastructure?
  - ☐ CloudFormation
  - ☐ CloudFormation Template anatomy

Quizzes:
ACA Mod 4 Designing for High Availability
ACA Mod 5 Automating your Infrastructure

SWINBURNE UNIVERSITY OF TECHNOLOGY

4



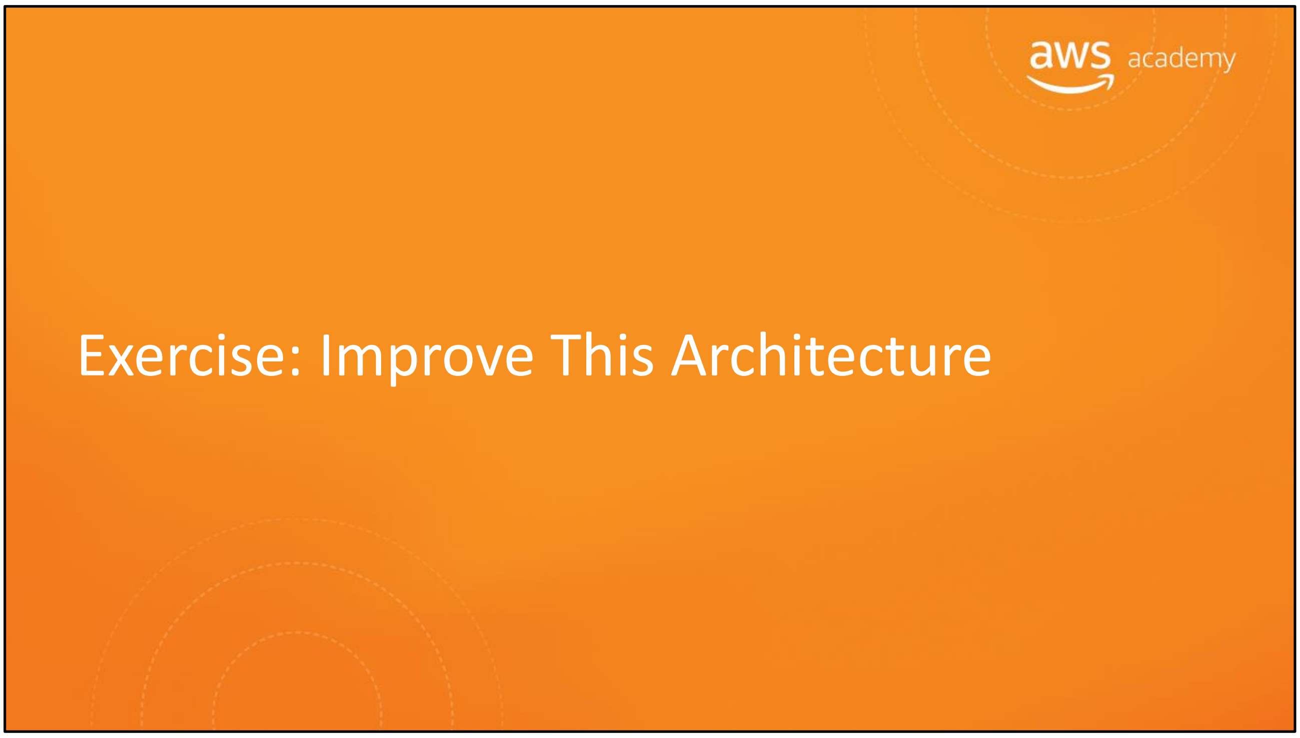
Best practices for creating HA apps - Big Picture

- Avoid Single Points of Failure
  - ☐ AZs, Route53 (Week 3), ELBs (Week 7)
- Manage demand
  - ☐ Balance Load across your infrastructure (Week 7)
  - ☐ Monitor your Infrastructure (Week 7 and this week)
  - ☐ Enable Scaling (Week 7 and this week)
- Automate your infrastructure
  - ☐ Launch configurations (Week 7)
  - ☐ Infrastructure as code (this week)
- Decouple your infrastructure (Week 9 – next week)
  - ☐ Reduce runtime dependencies between components.
- Disaster Recovery (Week 11)

5

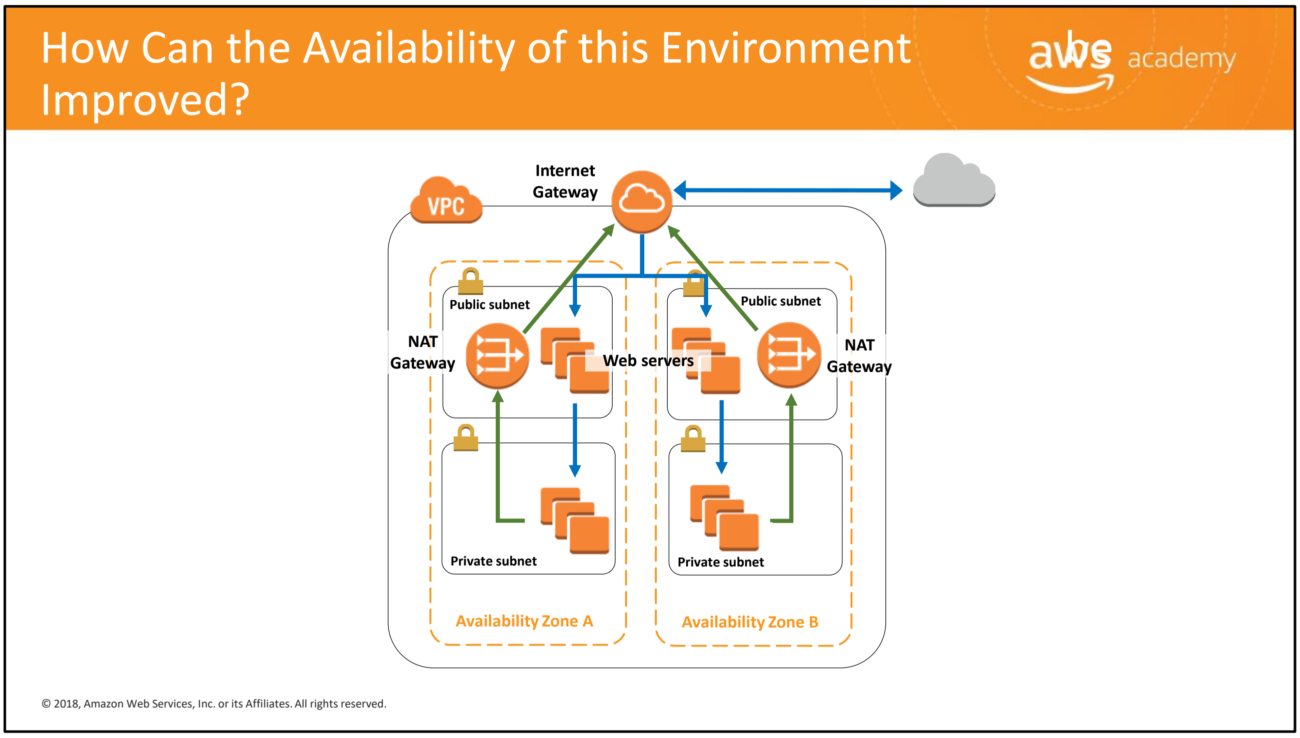
Extras for ACA certification (not covered in COS80001)

- ACA Mod 3.3: Connections Outside of Amazon VPC
- ACA Mod 4.3: EC2 Auto-recovery
- ACA Mod 4.5: Lambda scaling Case Study
- ACA Mod 5.4: Other AWS Infrastructure as code services
  - Elastic Beanstalk
  - AWS OpWorks (uses Puppet or CHEF)
  - AWS Run Command
  - Third Part Apps on EC2 (e.g. CHEF, Ansible, …)

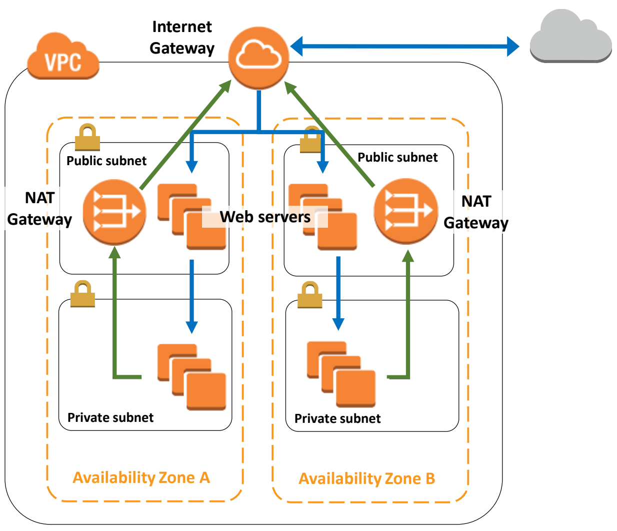


Exercise: Improve This Architecture

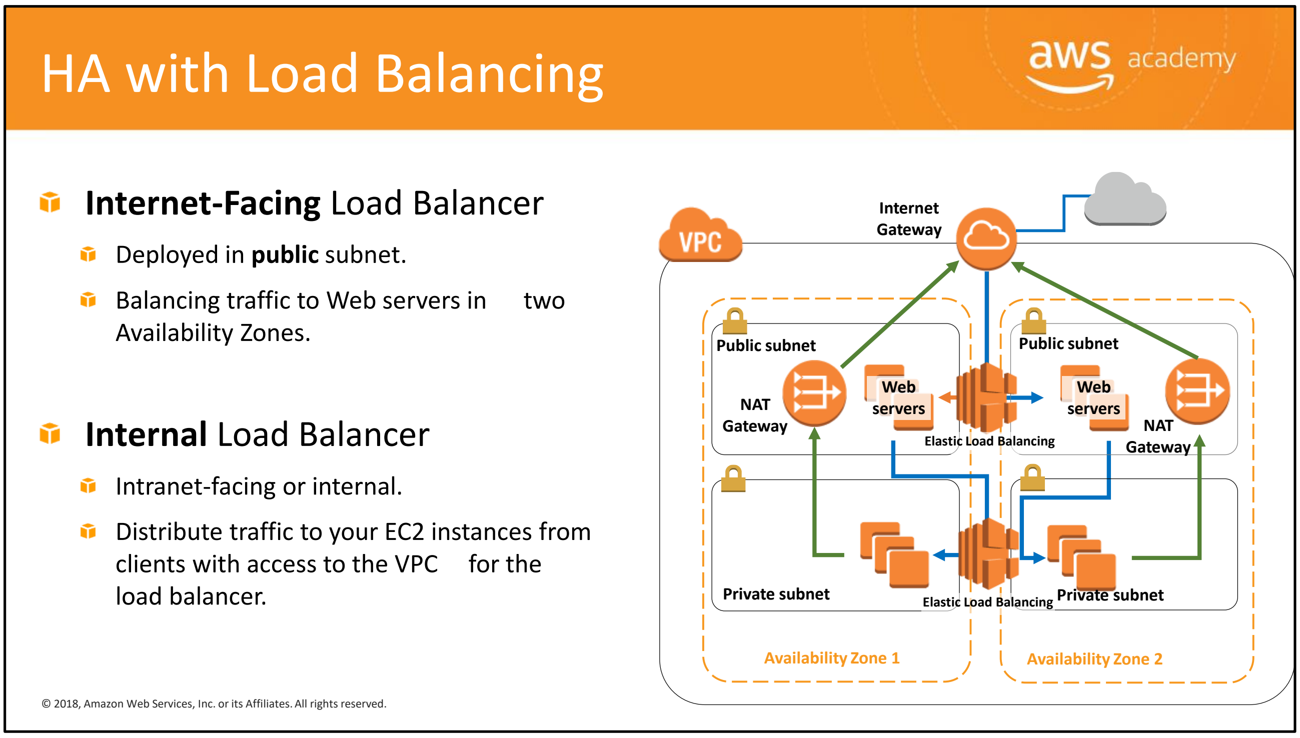
Now, it is your turn to apply your cloud architecting knowledge.



How Can the Availability of this Environment Improved?

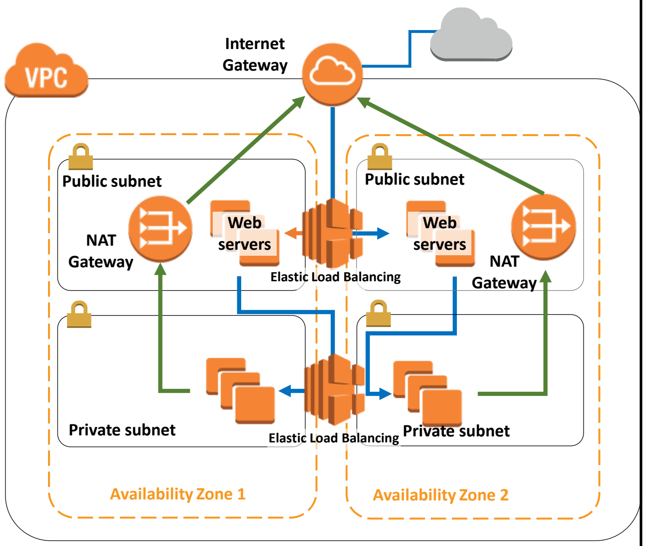


Review this architecture and identify changes that will improve the availability of the solution.



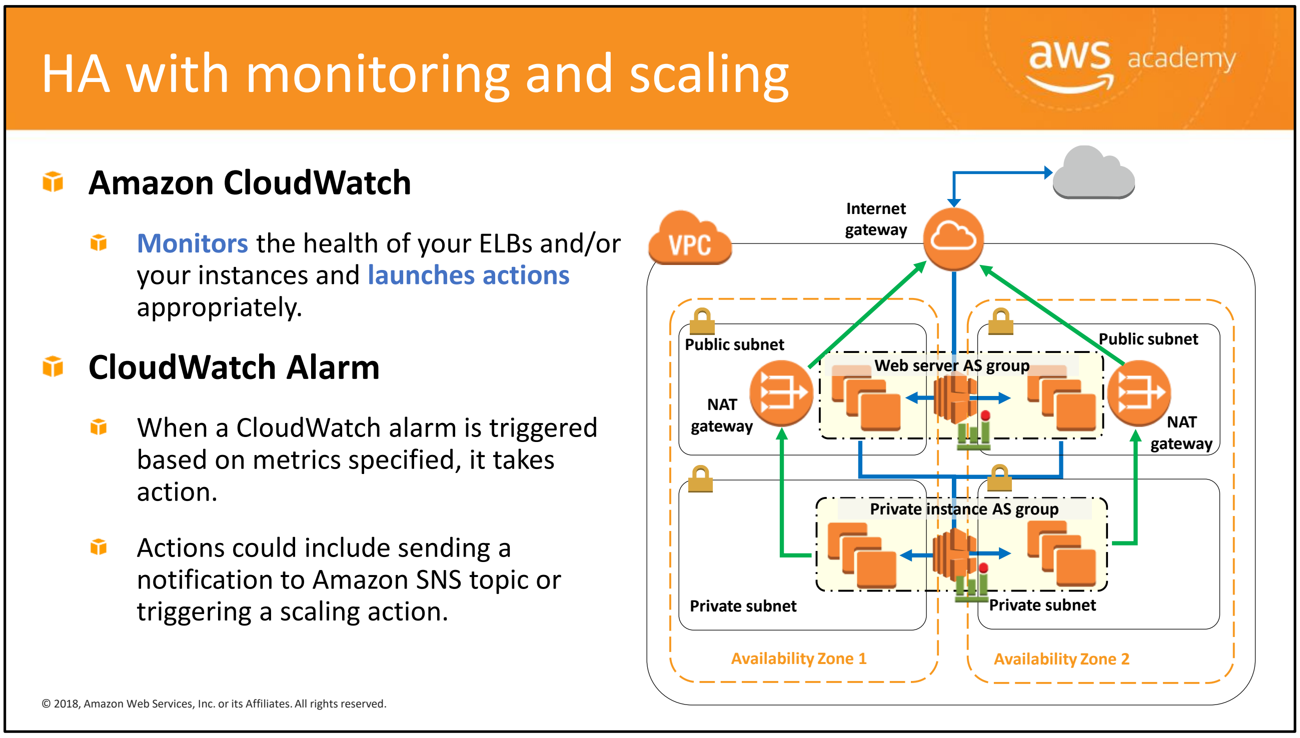
HA with Load Balancing

- **Internet-Facing** Load Balancer
  - Deployed in **public** subnet.
  - Balancing traffic to Web servers in two Availability Zones.
- **Internal** Load Balancer
  - Intranet-facing or internal.
  - Distribute traffic to your EC2 instances from clients with access to the VPC for the load balancer.



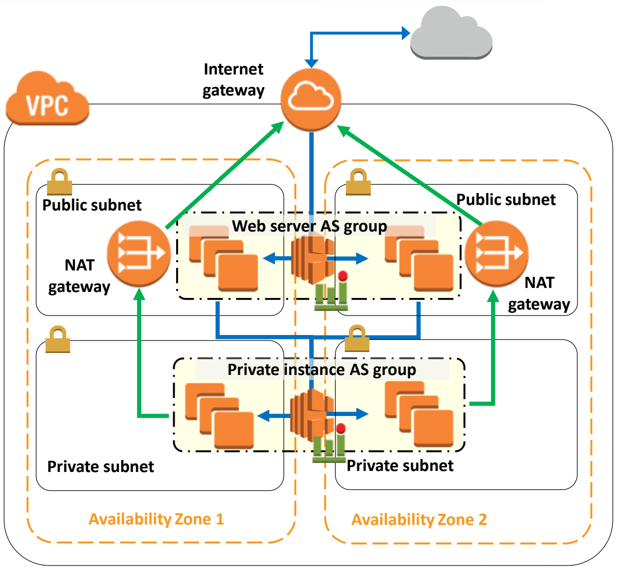
When you create your load balancer in a VPC, you can make your load balancer internal—or private—or internet-facing—or public. Internet-facing load balancers are deployed in the public subnet, and they can balance traffic between two separate Availability Zones, as shown in the diagram. When you make your load balancer internet-facing, a DNS name will be created with the public IP address. The DNS records are publicly resolvable in both cases.

An internal load balancer is not exposed to the internet. An internal load balancer distributes traffic to Amazon EC2 instances from clients with access to the VPC for the load balancer. In the diagram, the load balancer between the two private subnets distributes traffic between the private subnets. When you make your load balancer internal, a DNS name will be created, and it will contain the private IP address of your load balancer.



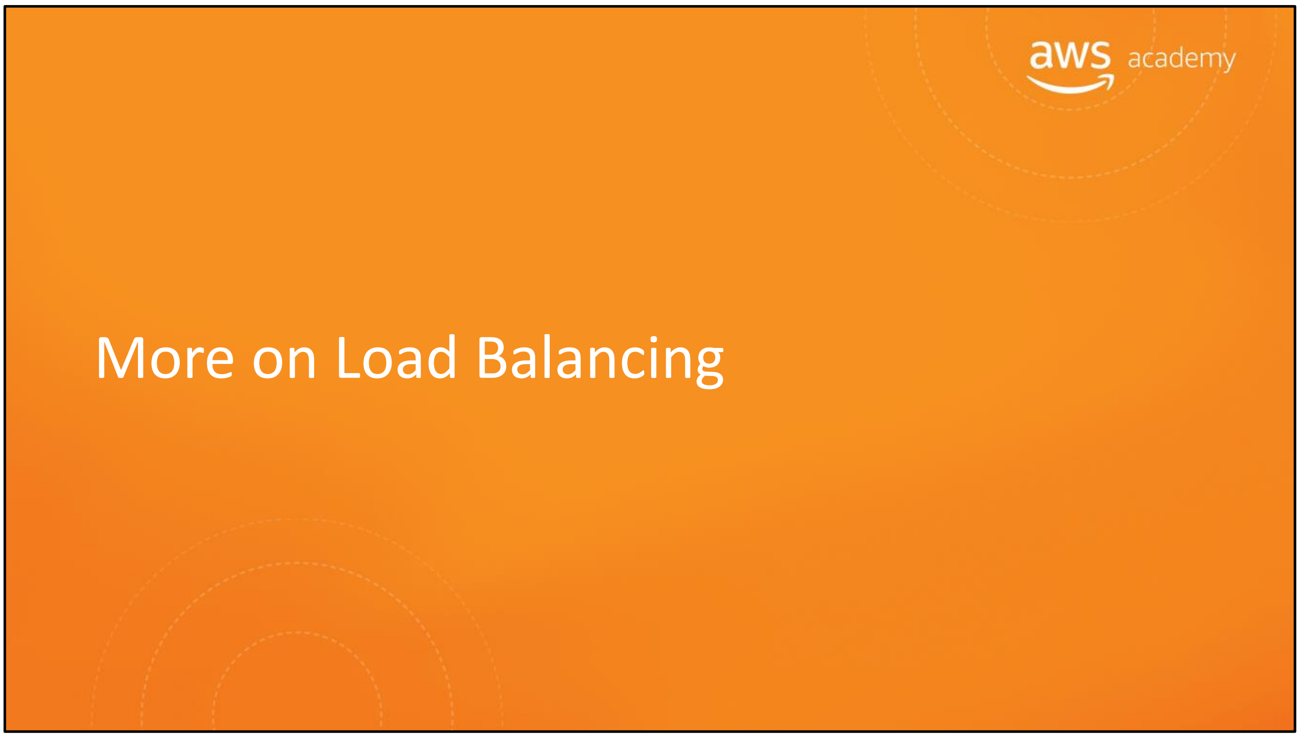
HA with monitoring and scaling

- **Amazon CloudWatch**
  - **Monitors** the health of your ELBs and/or your instances and **launches actions** appropriately.
- **CloudWatch Alarm**
  - When a CloudWatch alarm is triggered based on metrics specified, it takes action.
  - Actions could include sending a notification to Amazon SNS topic or triggering a scaling action.



To make this application environment even more highly available, you can monitor the Elastic Load Balancing load balancers, the instances, or both to determine when new instances need to be launched. When a CloudWatch alarm is triggered based on the metrics that you specified for it to monitor, it takes the action that you specified. The actions could include sending a notification to an Amazon SNS topic, triggering a scaling action, or both.

We will spend more time on event-driven scaling in a later module, but for now it's important to understand this basic pattern for making a web tier highly available.



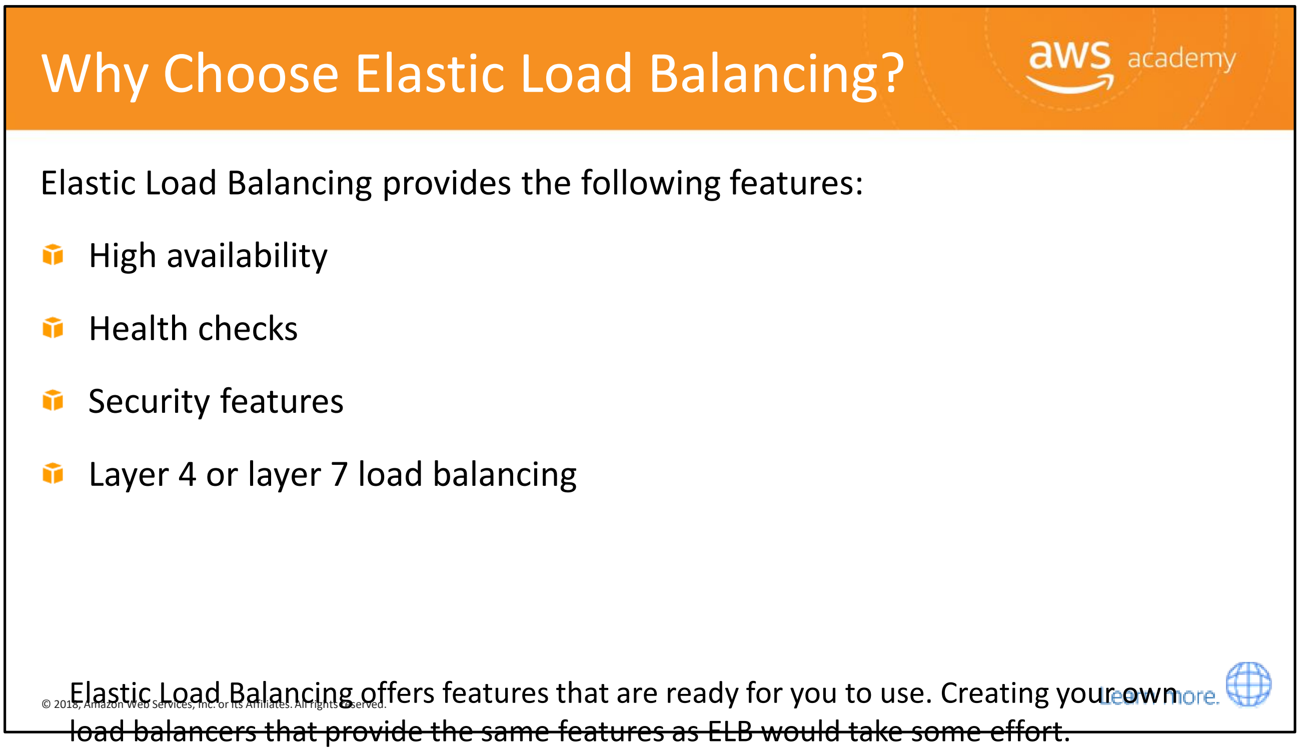
More on Load Balancing

Introducing Part 1: Scalability.

What is scalability? Scalability is an attribute that describes the ability of a process, network, software, or organization to increase or decrease resource size based on changes in demand.

Scalability is important because many applications must maintain steady application performance, even when demand is unpredictable or unexpected. Without enough resources, your applications can experience downtime or sluggish response times. With too many resources, you could end up paying for cloud services that you don't really need. Various scaling options are available that enable you to take full advantage of the elasticity of the cloud, while lowering cloud spend.

services you don't really need. Various scaling options are available that enable you to take full advantage of the elasticity of the cloud while lowering cloud spend.



Why Choose Elastic Load Balancing?

Elastic Load Balancing provides the following features:

- High availability
- Health checks
- Security features
- Layer 4 or layer 7 load balancing

Learn more.

Elastic Load Balancing offers features that are ready for you to use. Creating your own load balancers that provide the same features as ELB would take some effort.

For high availability, Elastic Load Balancing automatically distributes traffic across multiple targets—Amazon EC2 instances, containers and IP addresses—in a single Availability Zone or across multiple Availability Zones.

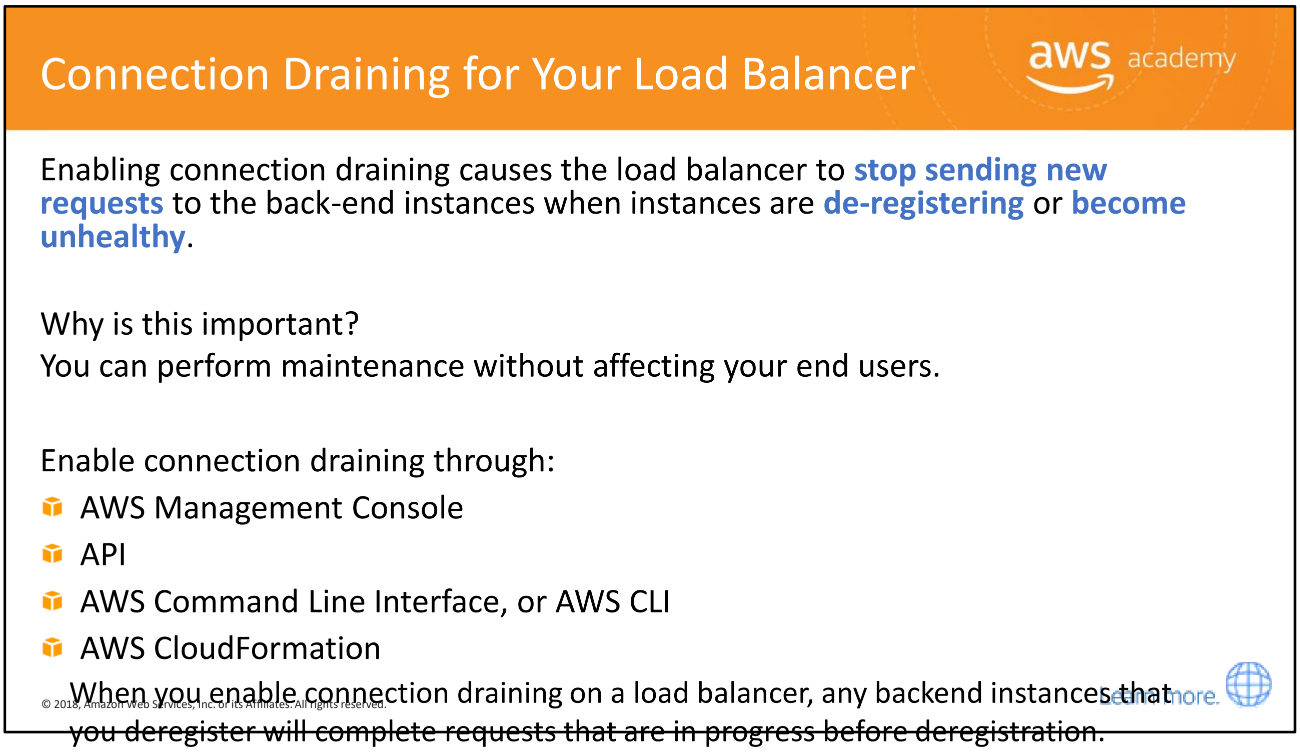
To discover the availability of your Amazon EC2 instances, the load balancer periodically sends pings, attempts connections, or sends requests to test the Amazon EC2 instances. These tests are called health checks. Each registered Amazon EC2 instance must respond to the target of the health check with an HTTP status code of 200 to be considered healthy by your load balancer.

You can use Amazon Virtual Private Cloud to create and manage security groups associated with load balancers to provide additional networking and security options. You can also create an internal—or non-internet-facing—load balancer.

For TLS termination, Elastic Load Balancing provides integrated certificate management and SSL decryption, which allows you the flexibility to centrally manage the SSL settings of the load balancer and offload CPU intensive work from your application.

You can load-balance HTTP or HTTPS applications for layer 7-specific features, or use strict layer 4 load balancing for applications that rely purely on the TCP protocol.

For a side-by-side feature comparison, see the Elastic Load Balancing features page.
https://aws.amazon.com/elasticloadbalancing/details/#compare



Connection Draining for Your Load Balancer

Enabling connection draining causes the load balancer to **stop sending new requests** to the back-end instances when instances are **de-registering** or **become unhealthy**.

Why is this important?
You can perform maintenance without affecting your end users.

Enable connection draining through:
- AWS Management Console
- API
- AWS Command Line Interface, or AWS CLI
- AWS CloudFormation

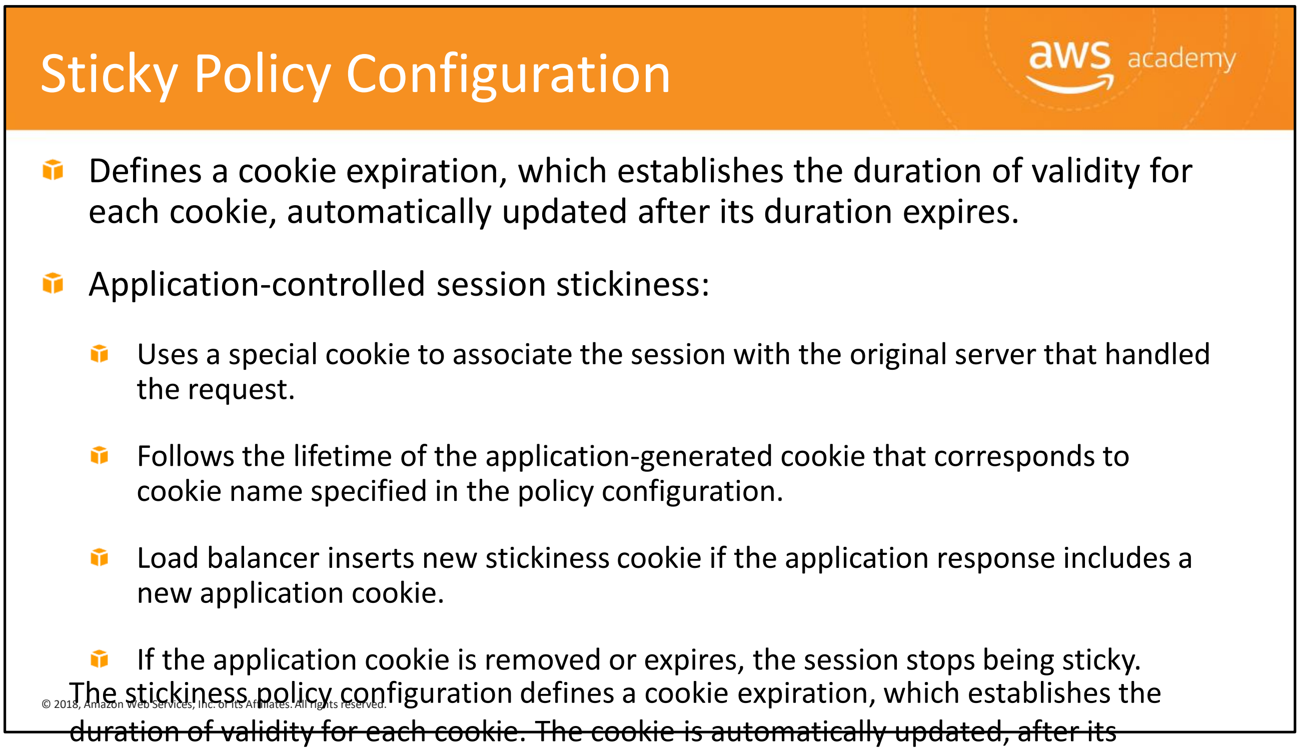
Learn more.

When you enable connection draining on a load balancer, any backend instances that you deregister will complete requests that are in progress before deregistration. Likewise, if a backend instance fails health checks, the load balancer will not send any new requests to the unhealthy instance, but it will allow existing requests to be completed while ensuring that in-flight requests continue to be served. This means that you can perform maintenance, such as deploying software upgrades or replacing backend instances, without affecting your customers' experience.

Connection draining is also integrated with Auto Scaling, which you can use to manage the capacity behind your load balancer. When connection draining is enabled, Auto Scaling will wait for outstanding requests to be completed before it terminates instances.

You can enable connection draining through:
• AWS Management Console
• API
• AWS Command Line Interface, or AWS CLI
• And AWS CloudFormation

For more information, see the ELB documentation page for how Elastic Load Balancing works.
http://docs.aws.amazon.com/ElasticLoadBalancing/latest/DeveloperGuide/how-elb-works.html

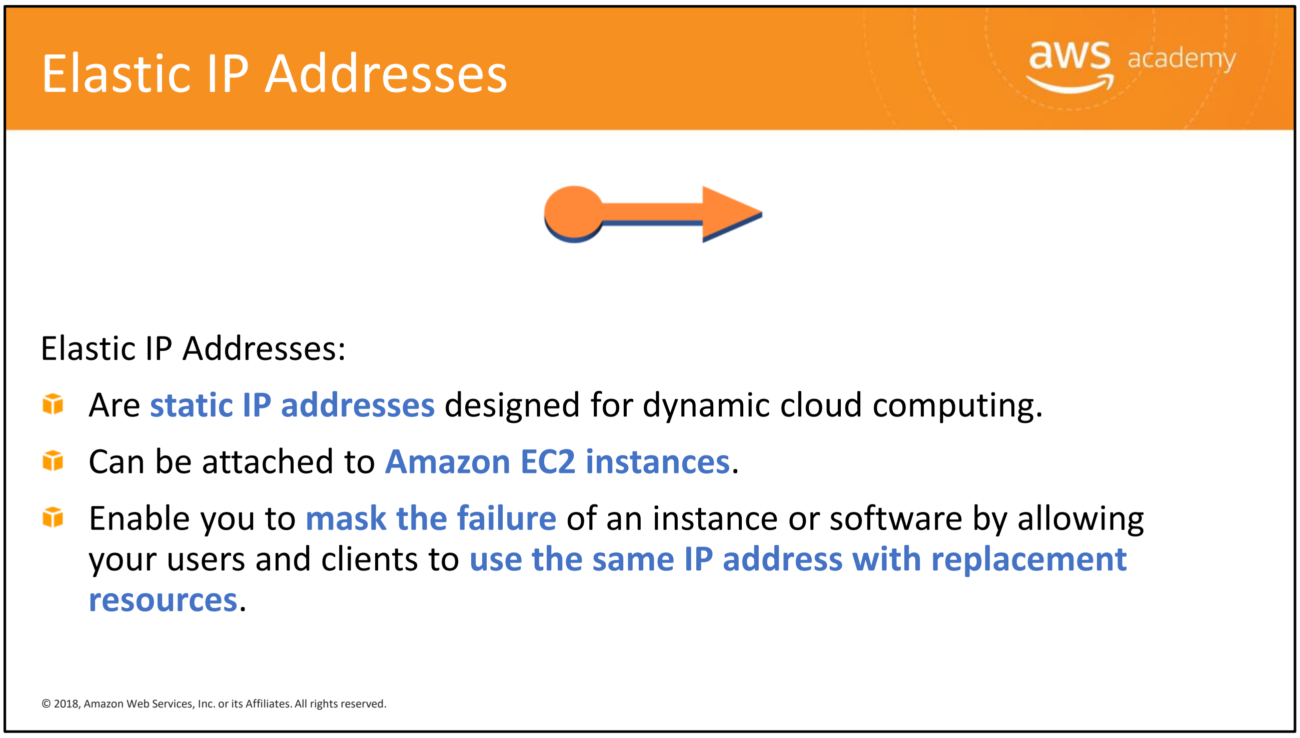


Sticky Policy Configuration

- Defines a cookie expiration, which establishes the duration of validity for each cookie, automatically updated after its duration expires.
- Application-controlled session stickiness:
  - Uses a special cookie to associate the session with the original server that handled the request.
  - Follows the lifetime of the application-generated cookie that corresponds to cookie name specified in the policy configuration.
  - Load balancer inserts new stickiness cookie if the application response includes a new application cookie.
  - If the application cookie is removed or expires, the session stops being sticky.

The stickiness policy configuration defines a cookie expiration, which establishes the duration of validity for each cookie. The cookie is automatically updated, after its duration expires.

For application-controlled session stickiness, the load balancer uses a special cookie to associate the session with the original server that handled the request, but follows the lifetime of the application-generated cookie that corresponds to the cookie name specified in the policy configuration. The load balancer only inserts a new stickiness cookie if the application response includes a new application cookie. The load balancer stickiness cookie does not update with each request. If the application cookie is explicitly removed or expires, the session stops being sticky until a new application cookie is used.



Elastic IP Addresses

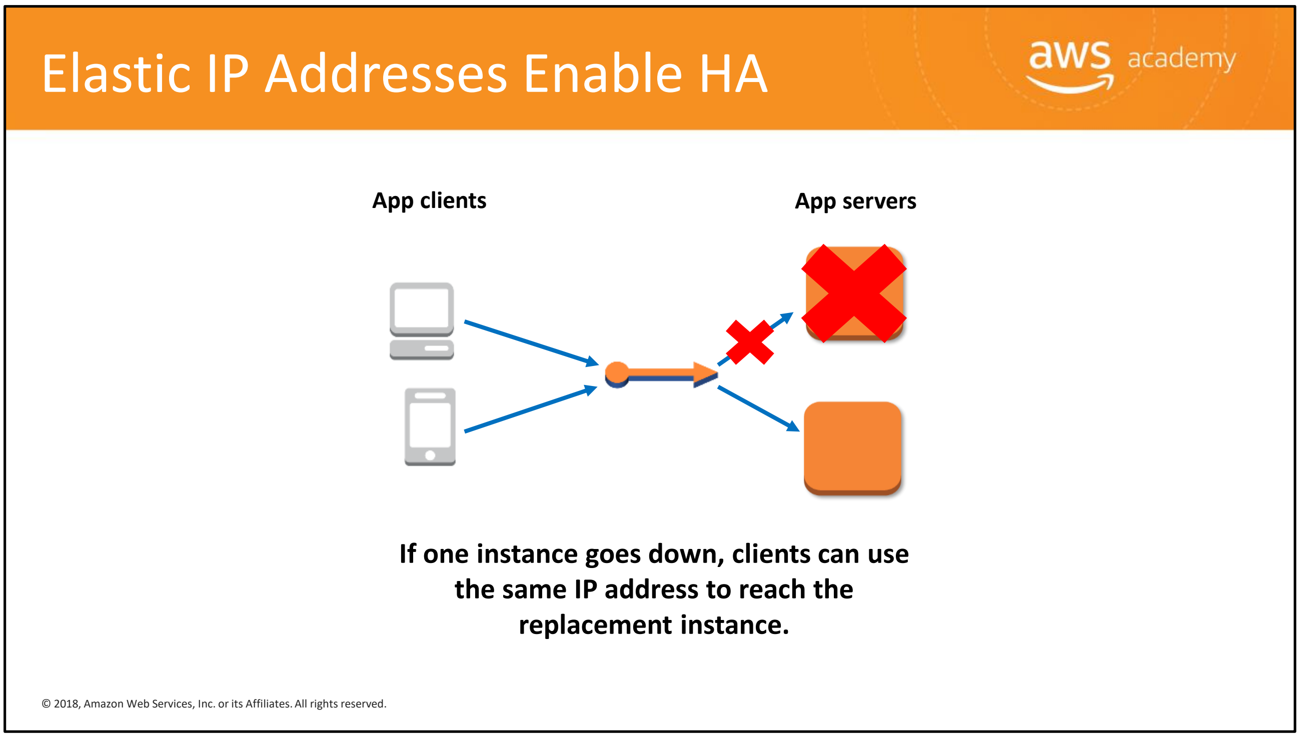
Elastic IP Addresses:

- Are **static IP addresses** designed for dynamic cloud computing.
- Can be attached to **Amazon EC2 instances**.
- Enable you to **mask the failure** of an instance or software by allowing your users and clients to **use the same IP address with replacement resources**.

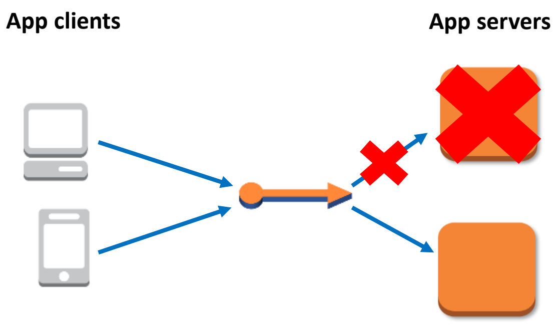
An Elastic IP address is a static IP address designed for dynamic cloud computing, and it can be attached to an Amazon EC2 instance. Elastic IP addresses are very important because they allow us to mask the failure of an instance or software by allowing users and clients to use the same IP address with replacement resources.

An Elastic IP address is an IP address that can be assigned to any Amazon EC2 instance. If that Amazon EC2 instance failed, a new Amazon EC2 instance could be launched with that IP address and the application does not require any changes as it will have the same IP address.

The foundation of the web tier includes the use of Elastic load balancers in the architecture. These load balancers send traffic to Amazon EC2 instances, and they can also send metrics to Amazon CloudWatch, which is a managed monitoring service. The metrics from Amazon EC2 and ELB can act as triggers, so that if you notice a particularly high latency or that your servers are becoming overused, you can take advantage of Auto Scaling to add more capacity to your web server fleet.



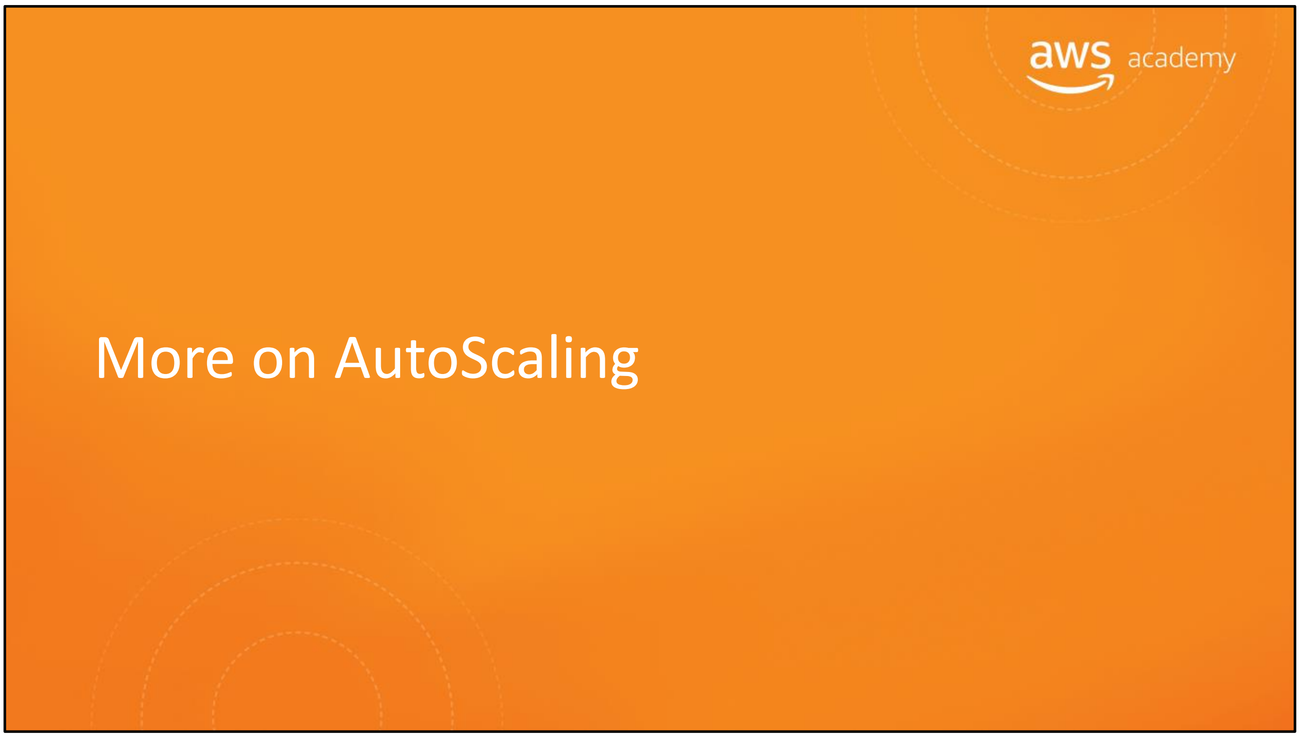
Elastic IP Addresses Enable HA



If one instance goes down, clients can use the same IP address to reach the replacement instance.

Let’s look at how using an Elastic IP address can enable high availability.

The diagram shows two separate clients: desktop and mobile. The clients are connecting to an application server. If that application server goes down, you can launch a new Amazon EC2 instance and assign the same Elastic IP address to it. This enables you to survive a server failure and not have to rewrite any code.



More on AutoScaling

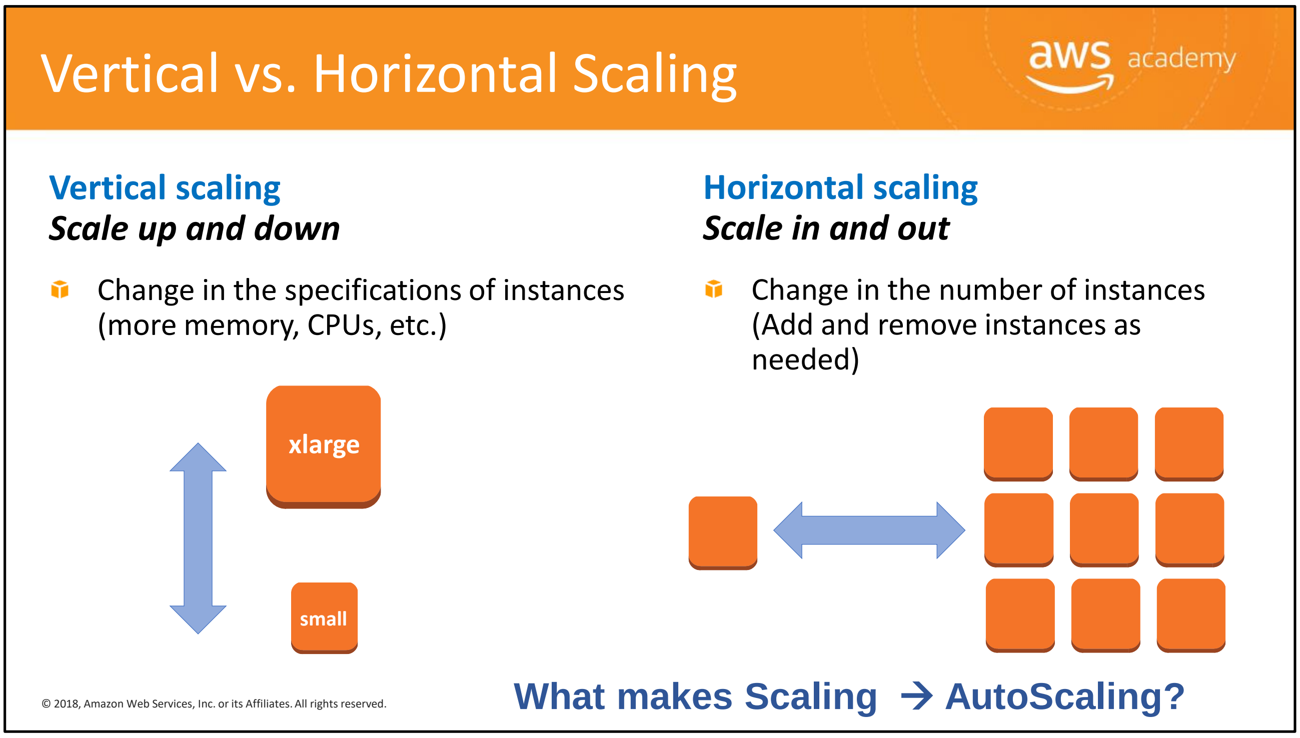
aws academy

Introducing Part 1: Scalability.

What is scalability? Scalability is an attribute that describes the ability of a process, network, software, or organization to increase or decrease resource size based on changes in demand.

Scalability is important because many applications must maintain steady application performance, even when demand is unpredictable or unexpected. Without enough resources, your applications can experience downtime or sluggish response times. With too many resources, you could end up paying for cloud services that you don't really need. Various scaling options are available that enable you to take full advantage of the elasticity of the cloud, while lowering cloud spend.

services you don't really need. Various scaling options are available that enable you to take full advantage of the elasticity of the cloud while lowering cloud spend.



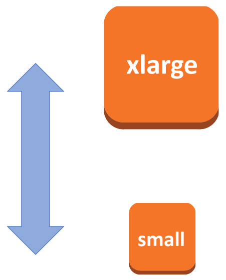
Vertical vs. Horizontal Scaling

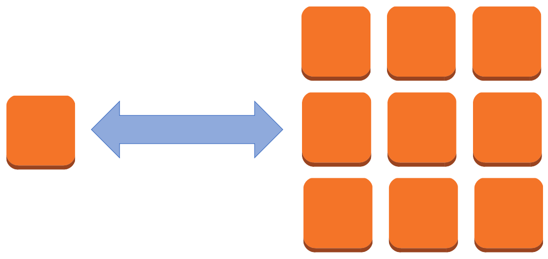
Vertical scaling
Scale up and down

- Change in the specifications of instances (more memory, CPUs, etc.)

Horizontal scaling
Scale in and out

- Change in the number of instances (Add and remove instances as needed)



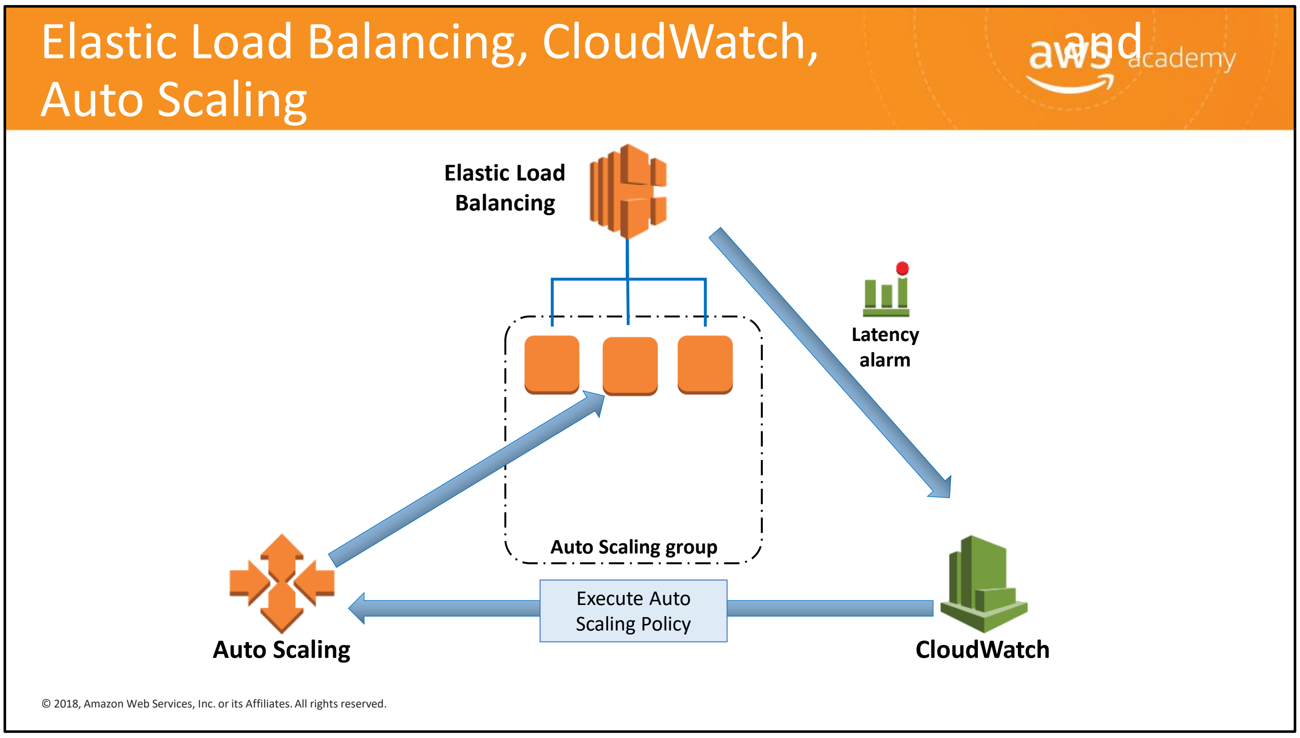


What makes Scaling → AutoScaling?

Now, let’s discuss vertical scaling versus horizontal scaling.

Scaling up—or vertical scaling–refers to using a more powerful piece of hardware. Vertical scaling changes the specifications of instances, such as adding memory and central processing units, or CPUs, as the workload increases. Eventually, this process will reach the limit . Also, there are some performance concerns. For example, if you run Java applications, more memory—or more memory for the Java heap—means that it takes longer for garbage collection to run. It can introduce a longer pause time. Another point to consider is that vertical scaling might require the server to be rebooted.

Horizontal scaling is virtually limitless. You can quickly scale in and out by changing the number of instances. Horizontal scaling is a good solution to handle a growing workload. However, if you're going to rely on horizontal scaling, it’s important to design your application so that it can fully take advantage of it.



Elastic Load Balancing, CloudWatch, and Auto Scaling

The slide shows an example of using Elastic Load Balancing, CloudWatch, and Auto Scaling. All of these services work well individually, but together, they become more powerful and increase the control and flexibility our customers demand.

• In this scenario, a CloudWatch alarm is assigned to the load balancer that keeps track of latency. When it is triggered, the alarm notifies Amazon CloudWatch.
• This triggers CloudWatch to execute an Auto Scaling policy.
• Auto Scaling scales the assigned Auto Scaling group out and adds another instance.

Amazon CloudWatch (revisited)

Amazon CloudWatch:

- **Monitors your instances**, and collects and processes raw data into readable, near real-time metrics.
- **Sends notifications and triggers Auto Scaling** actions based on metrics you specify.
- **Turns metrics into statistics**, to be used by CloudWatch alarms.

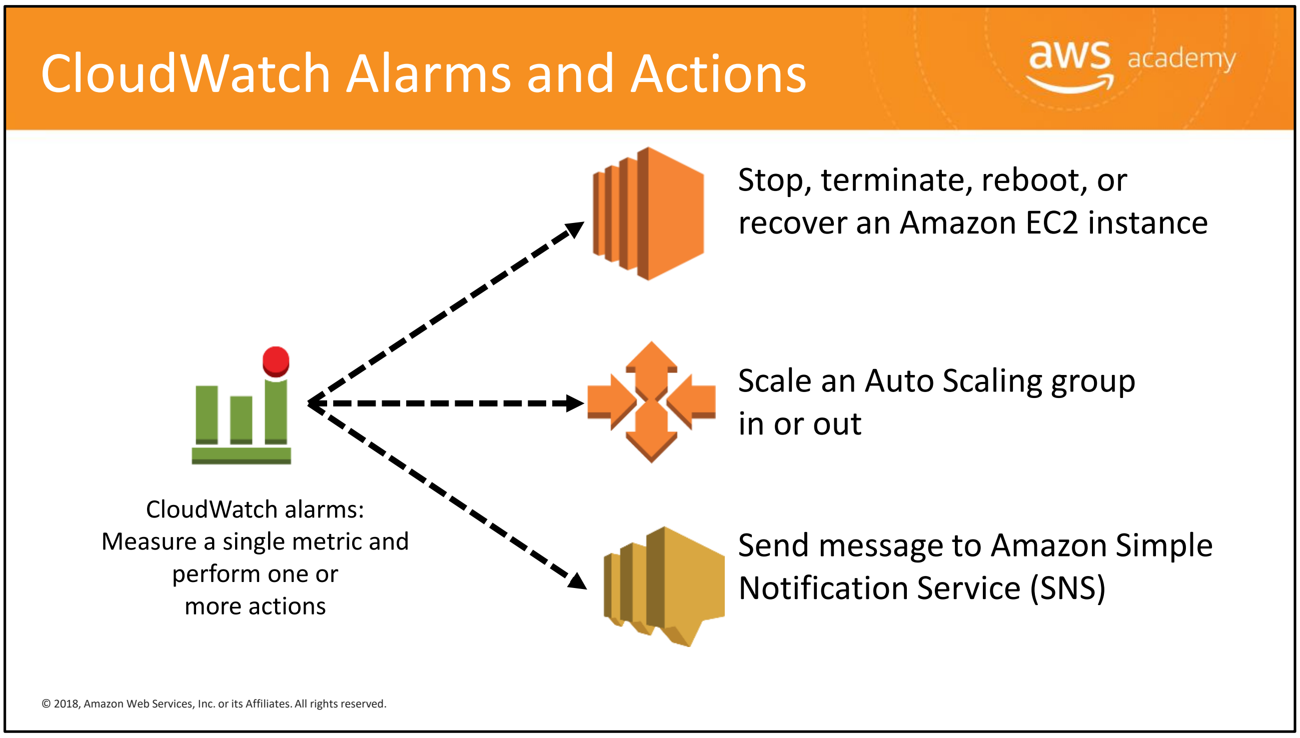
You can capture this information and monitor your infrastructure by using Amazon CloudWatch.

Amazon CloudWatch monitors your instances, and it collects and processes raw data into readable, near real-time metrics. The foundation of the web tier includes the use of Elastic Load Balancing load balancers in the architecture. These load balancers send traffic to Amazon EC2 instances, and they can also send metrics to Amazon CloudWatch. Set alarms on any of your metrics to receive notifications, or to take other automated actions when your metric crosses your specified threshold.

Amazon CloudWatch sends notifications and triggers Auto Scaling actions based on metrics that you specify, such as CPU usage. The metrics from Amazon EC2 and the Elastic Load Balancing can act as triggers, so if you notice a particularly high latency or that servers are becoming overused, you can take advantage of Auto Scaling to add more capacity to your web server fleet. You can use alarms to detect and shut down Amazon EC2 instances that are unused or underused.

Amazon CloudWatch collects metrics, turns the metrics into statistics that can be used by CloudWatch alarms, and displays them all in one place. CloudWatch alarms are based on statistics. Statistics are metric data that is aggregated over specified periods of time. CloudWatch provides statistics based on the metric data points from your custom data, or from data that other AWS services send to CloudWatch .

Aggregations are made using the namespace, metric name, dimensions, and the data point unit of measure, within the time period that you specify.

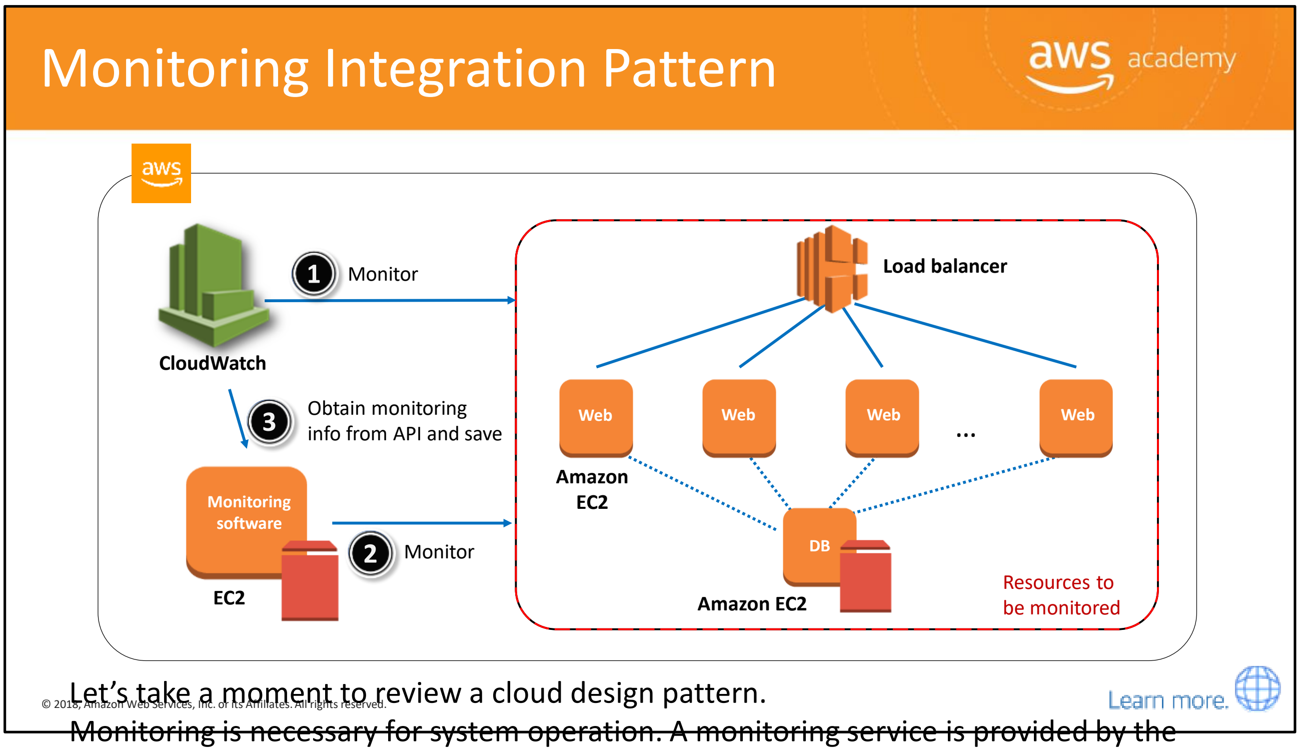


CloudWatch Alarms and Actions

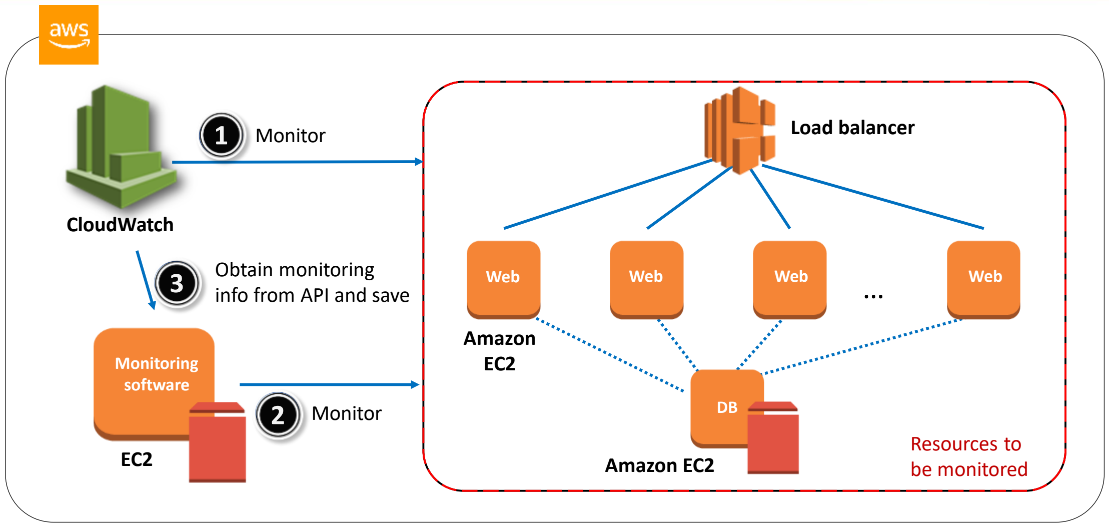
aws academy
Stop, terminate, reboot, or recover an Amazon EC2 instance
Scale an Auto Scaling group in or out
Send message to Amazon Simple Notification Service (SNS)
CloudWatch alarms: Measure a single metric and perform one or more actions

Amazon CloudWatch alarms and actions that are based on a single metric can do all of the following actions.

It can stop, terminate, reboot, or recover an Amazon EC2 instance. It can scale an Auto Scaling group in or out, and it can send a message to Amazon Simple Notification Service—or Amazon SNS—so that an administrator can know what's happening.



Monitoring Integration Pattern



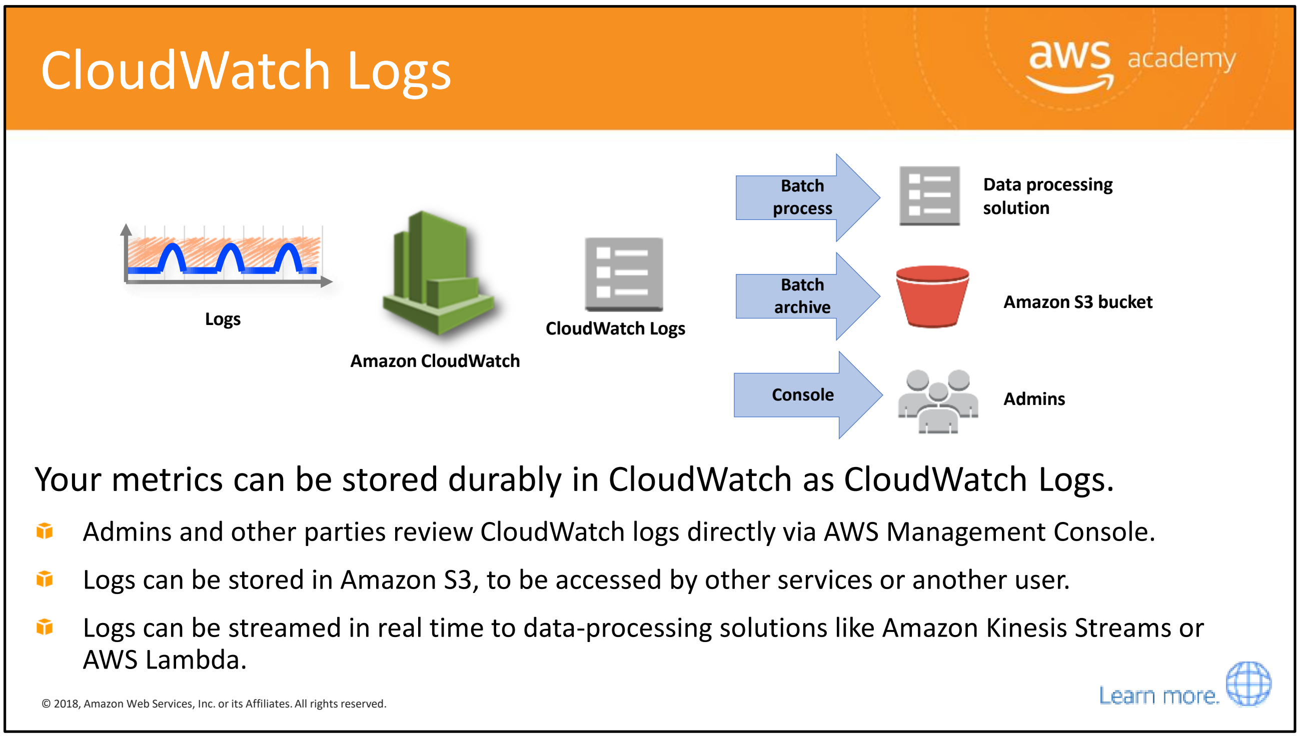
Learn more.

Let’s take a moment to review a cloud design pattern.
Monitoring is necessary for system operation. A monitoring service is provided by the AWS Cloud. However, because the monitoring service in the AWS Cloud cannot monitor the internal workings of a virtual server—such as the operating system, middleware, applications, etc.—you need to have an independent monitoring system.

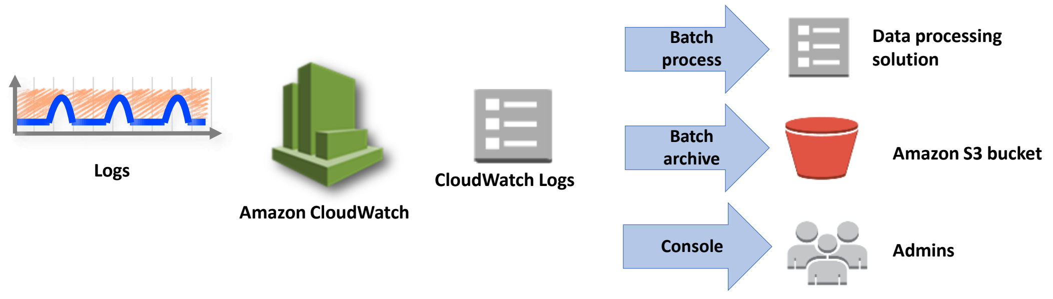
For example, the virtual server is monitored by the AWS Cloud monitoring service, but you need to use your own system to monitor the operating system, middleware, applications, etc. The cloud monitoring service provides an API, which enables you to use your monitoring system to perform centralized control, including of the cloud side, through this API, to obtain information from the cloud monitoring system.

To implement the monitoring service, install monitoring software on the Amazon EC2 instance so that you can obtain monitoring information from the CloudWatch monitoring service.
• Install monitoring software, such as Nagios, Zabbix, Munin , or the like.
• Use a plug-in to obtain monitoring information by using the CloudWatch API, and to write that information to the monitoring software.
• And use the plug-in to perform monitoring, including the information from AWS.

To learn more about the Cloud Design Pattern, go to the following Monitoring Integration Pattern page.
http://en.clouddesignpattern.org/index.php/CDP:Monitoring_Integration_Pattern



CloudWatch Logs



Your metrics can be stored durably in CloudWatch as CloudWatch Logs.

- Admins and other parties review CloudWatch logs directly via AWS Management Console.
- Logs can be stored in Amazon S3, to be accessed by other services or another user.
- Logs can be streamed in real time to data-processing solutions like Amazon Kinesis Streams or AWS Lambda.

Learn more.

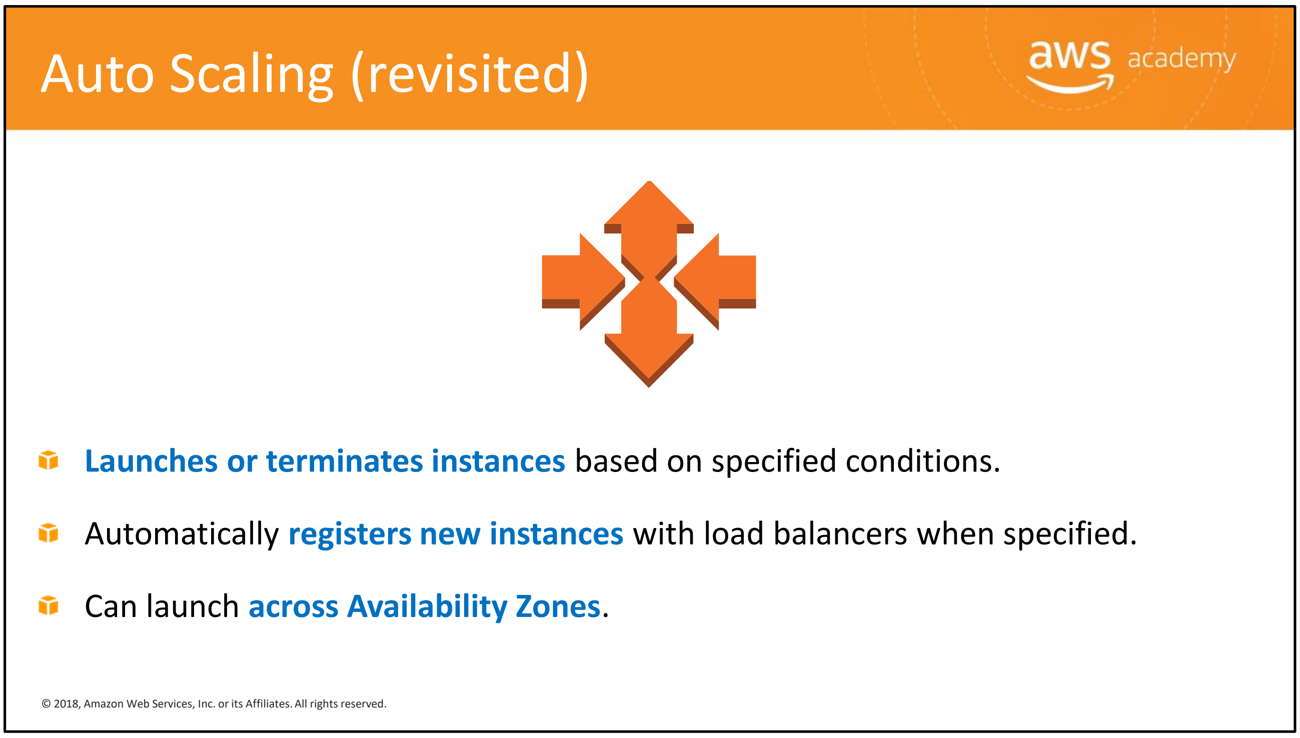
You can use CloudWatch Logs to store and monitor your log data in highly durable storage.

With Amazon CloudWatch Logs, you can take those logs and send them to streams for data processing, send them to an Amazon Simple Storage Service—or Amazon S3—bucket for durability, or have administrators access them directly from the AWS Management Console.

Logs can be streamed in real time to data-processing solutions, such as Amazon Kinesis Streams or AWS Lambda.

You can change the log retention setting so that any log events that are older than this setting are automatically deleted. With the CloudWatch Logs agent, you can quickly send both rotated and non-rotated log data off a host and into the log service. You can then access the raw log data when you need it.

To learn more about CloudWatch Logs, go to the following CloudWatch documentation page. http://docs.aws.amazon.com/AmazonCloudWatch/latest/DeveloperGuide/WhatIsCloudWatchLogs.html

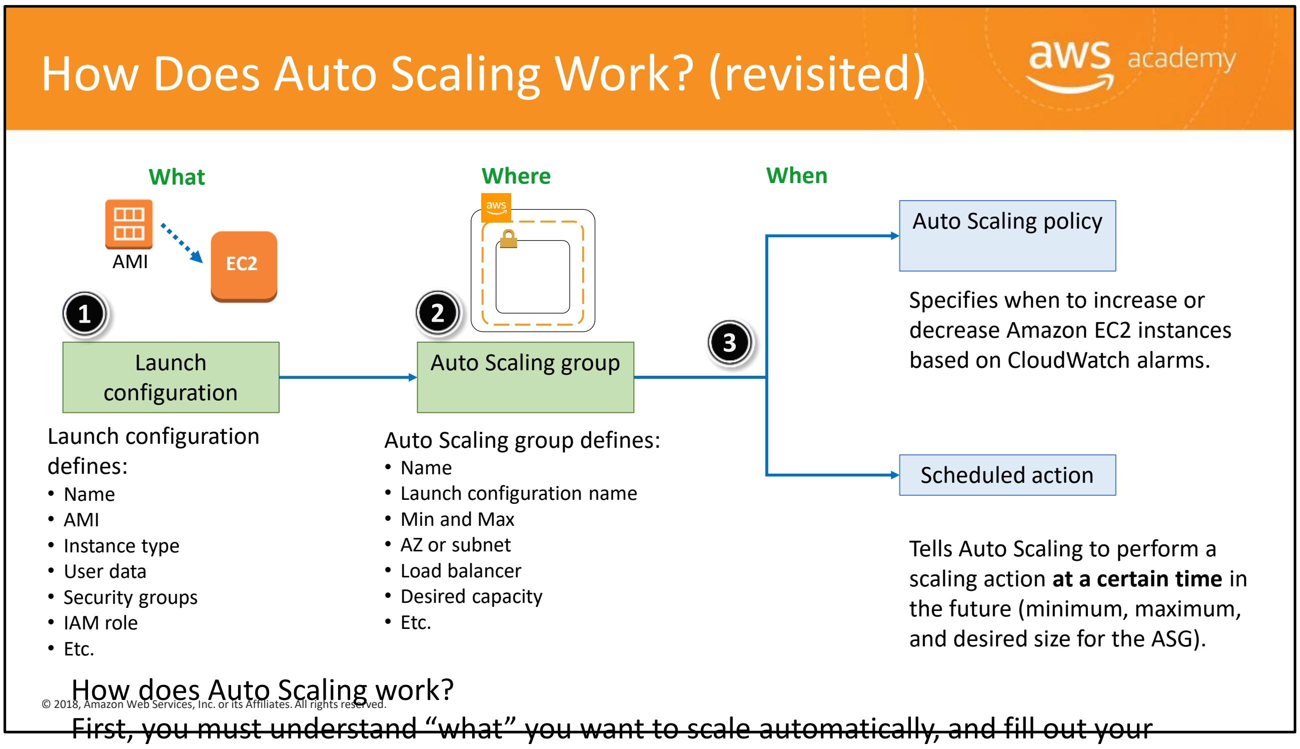


Auto Scaling (revisited)

- **Launches or terminates instances** based on specified conditions.
- Automatically **registers new instances** with load balancers when specified.
- Can launch **across Availability Zones**.

Using automatic scaling, you can launch or terminate instances as demand on your application increases or decreases, based on specific conditions. It automatically registers new instances with your Elastic Load Balancing load balancers, when this setting is specified. This service is available across all Availability Zones.

Auto Scaling integrates with Elastic Load Balancing so that you can attach one or more load balancers to an existing Auto Scaling group. After you attach the load balancer, it automatically registers the instances in the group, and it distributes incoming traffic across the instances. When one Availability Zone becomes unhealthy or is unavailable, Auto Scaling launches new instances in an unaffected Availability Zone. When the unhealthy Availability Zone returns to a healthy state, Auto Scaling automatically redistributes the application instances evenly across all of the Availability Zones for your Auto Scaling group. Auto Scaling does this by attempting to launch new instances in the Availability Zone with the fewest instances. If the attempt fails, however, Auto Scaling attempts to launch instances in other Availability Zones until it succeeds.



How Does Auto Scaling Work? (revisited)

Launch configuration defines:

- Name
- AMI
- Instance type
- User data
- Security groups
- IAM role
- Etc.

Auto Scaling group defines:

- Name
- Launch configuration name
- Min and Max
- AZ or subnet
- Load balancer
- Desired capacity
- Etc.

Specifies when to increase or decrease Amazon EC2 instances based on CloudWatch alarms.

Tells Auto Scaling to perform a scaling action **at a certain time** in the future (minimum, maximum, and desired size for the ASG).

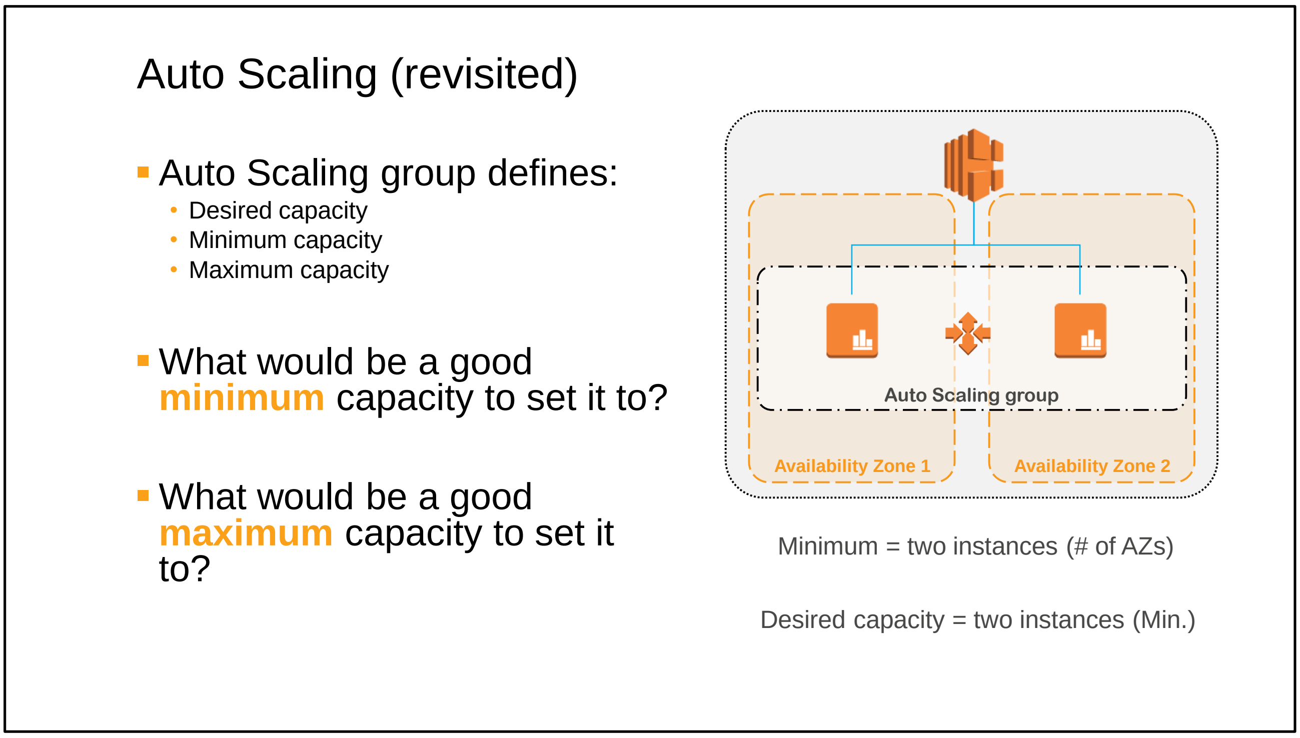
How does Auto Scaling work?
First, you must understand “what” you want to scale automatically, and fill out your launch configuration.
A launch configuration defines how Auto Scaling should launch your Amazon EC2 instances, which is similar to the ec2-run-instances API operation. Auto Scaling provides you with an option to create a new launch configuration by using the attributes from an existing Amazon EC2 instance. When you use this option, Auto Scaling copies the attributes from the specified instance into a template, which you can use to launch one or more Auto Scaling groups.

Next, you’ll determine “where” you want to launch the server. You must determine whether you need to launch the Auto Scaling group in a particular Availability Zone, what subnets it needs access to, what the name of the launch configuration is, what the desired capacity is, etc.

Finally, you’ll determine “when,” which is set by your Auto Scaling policy. A policy is a set of instructions for Auto Scaling that tells the service how to respond to alarm messages. An Auto Scaling group uses a combination of policies and alarms to determine when the specified conditions for launching and terminating instances are met. An alarm is an object that watches over a single metric—for example, the average CPU use of your Amazon EC2 instances in an Auto Scaling group—over a time period that you specify. The alarm performs one or more actions—such as sending messages to Auto Scaling—when the value of the metric breaches the thresholds that you define, and over the number of time periods that you specify.

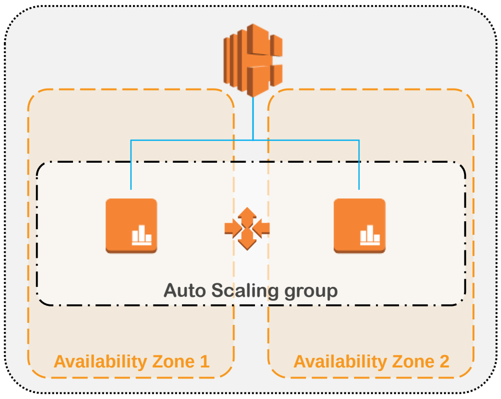
The policy type determines how the scaling action is performed. Auto Scaling supports the following policy types:
• Simple scaling to increase or decrease the current capacity of the group based on a single scaling adjustment.
• And step scaling to increase or decrease the current capacity of the group based on a set of scaling adjustments--also known as step adjustments--that vary based on the size of the alarm breach.

If you base scaling on a scheduled action, you can scale your application in response to predictable load changes.
To configure your Auto Scaling group to scale based on a schedule, you need to create scheduled actions. A scheduled action tells Auto Scaling to perform a scaling action at a certain time in the future. To create a scheduled scaling action, you specify the start time when you want the scaling action to take effect, and you specify the new minimum, maximum, and desired sizes that you want for that group at that time. At the specified time, Auto Scaling updates the group to set the new values for minimum, maximum, and desired sizes, as specified by your scaling action. For example, if you know you have a large batch job every Thursday night and you need extra CPU cycles, you might have Auto Scaling groups automatically handle the batch job. The Auto Scaling groups can then scale back down after the batch job is over. This way, no one needs to wake up at 2AM to make sure the servers are up, and you don’t have to pay for the extra capacity on the other six days of the week when you don’t need it.



Auto Scaling (revisited)

- Auto Scaling group defines:
  - Desired capacity
  - Minimum capacity
  - Maximum capacity
- What would be a good **minimum** capacity to set it to?
- What would be a good **maximum** capacity to set it to?

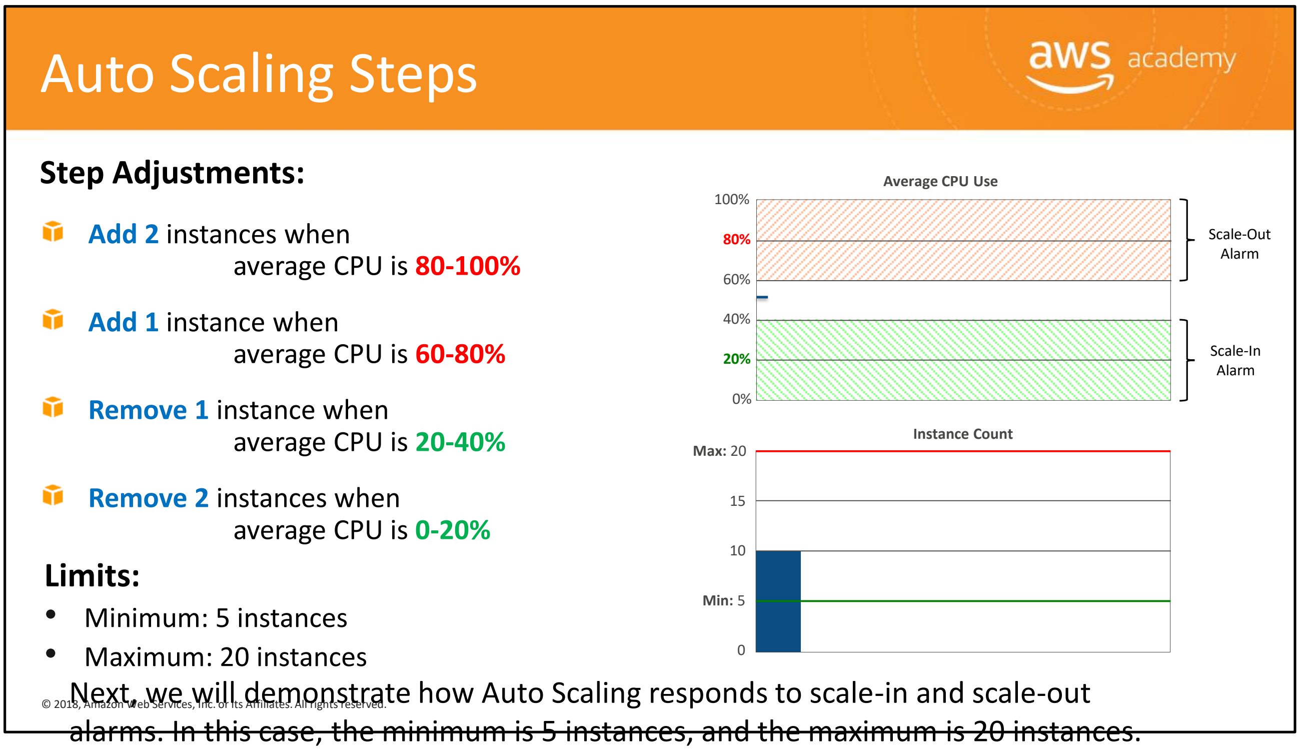


Minimum = two instances (# of AZs)

Desired capacity = two instances (Min.)

You don’t have to specify the desired capacity because minimum capacity is the initial desired capacity. You pay for what you use; therefore, you don’t need to launch more instances than you need and auto scale based on demand.
Deciding on the minimum capacity size depends on the type of applications that you are running. Here are some things to consider:

- If you are processing batches that run once a week, you might want to set the minimum to zero.
- Remember that it takes minutes to launch a new instance (depending on the complexity of your launch configuration). You may not be able to afford zero minimum capacity to start with.



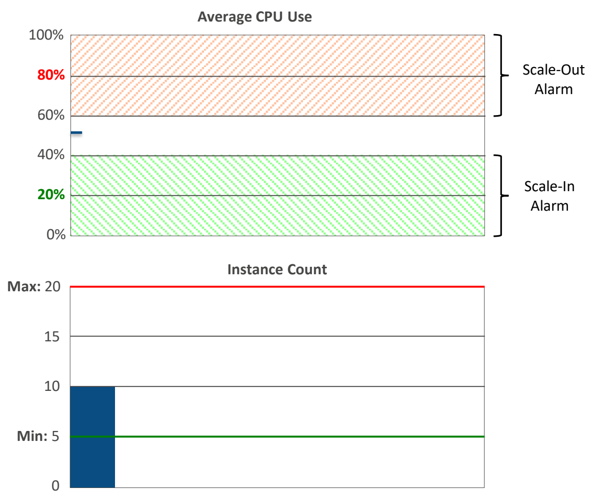
Auto Scaling Steps

Step Adjustments:

- **Add 2** instances when average CPU is **80-100%**
- **Add 1** instance when average CPU is **60-80%**
- **Remove 1** instance when average CPU is **20-40%**
- **Remove 2** instances when average CPU is **0-20%**

Limits:

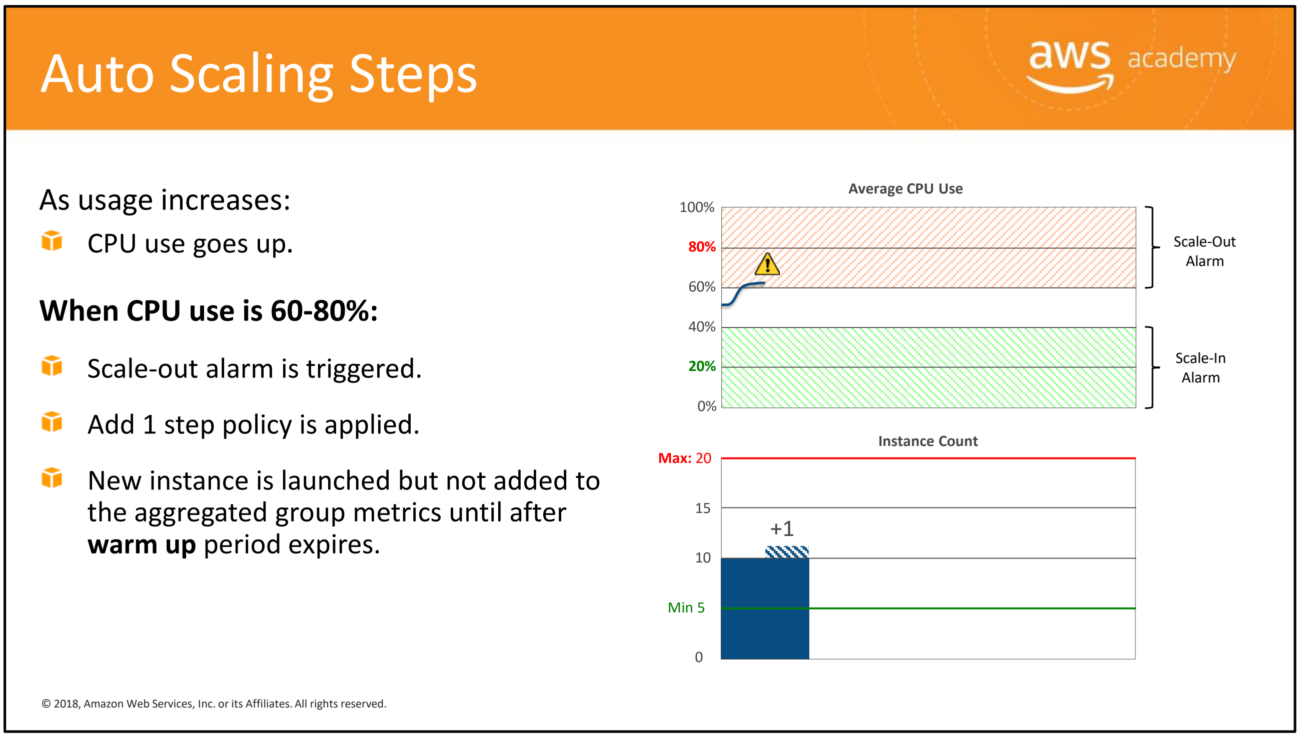
- Minimum: 5 instances
- Maximum: 20 instances



Next, we will demonstrate how Auto Scaling responds to scale-in and scale-out alarms. In this case, the minimum is 5 instances, and the maximum is 20 instances.

When you create a step scaling policy, you add one or more step adjustments that enable you to scale resources based on the size of the alarm breach. Each step adjustment specifies a lower bound for the metric value, an upper bound for the metric value, and the amount by which to scale, based on the scaling adjustment type.
There are a few rules for the step adjustments for your policy:

- The ranges of your step adjustments can't overlap or have a gap.
- Only one step adjustment can have a null lower bound, or negative infinity. If one step adjustment has a negative lower bound, then there must be a step adjustment with a null lower bound.
- Only one step adjustment can have a null upper bound, or positive infinity. If one step adjustment has a positive upper bound, then there must be a step adjustment with a null upper bound.
- The upper and lower bound can't be null in the same step adjustment.
- If the metric value is above the breach threshold, the lower bound is inclusive and the upper bound is exclusive. If the metric value is below the breach threshold, the lower bound is exclusive and the upper bound is inclusive.



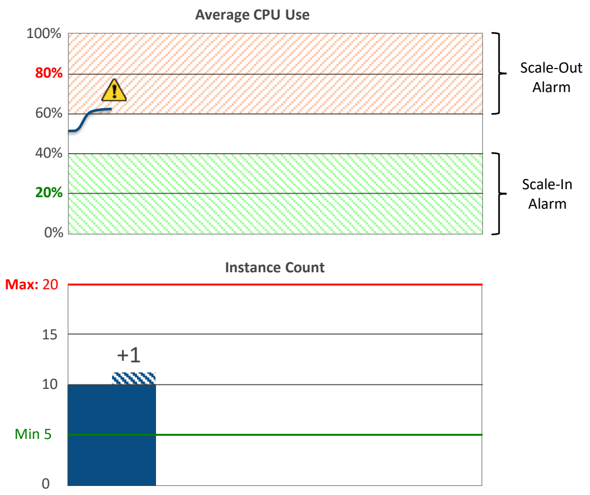
Auto Scaling Steps

As usage increases:

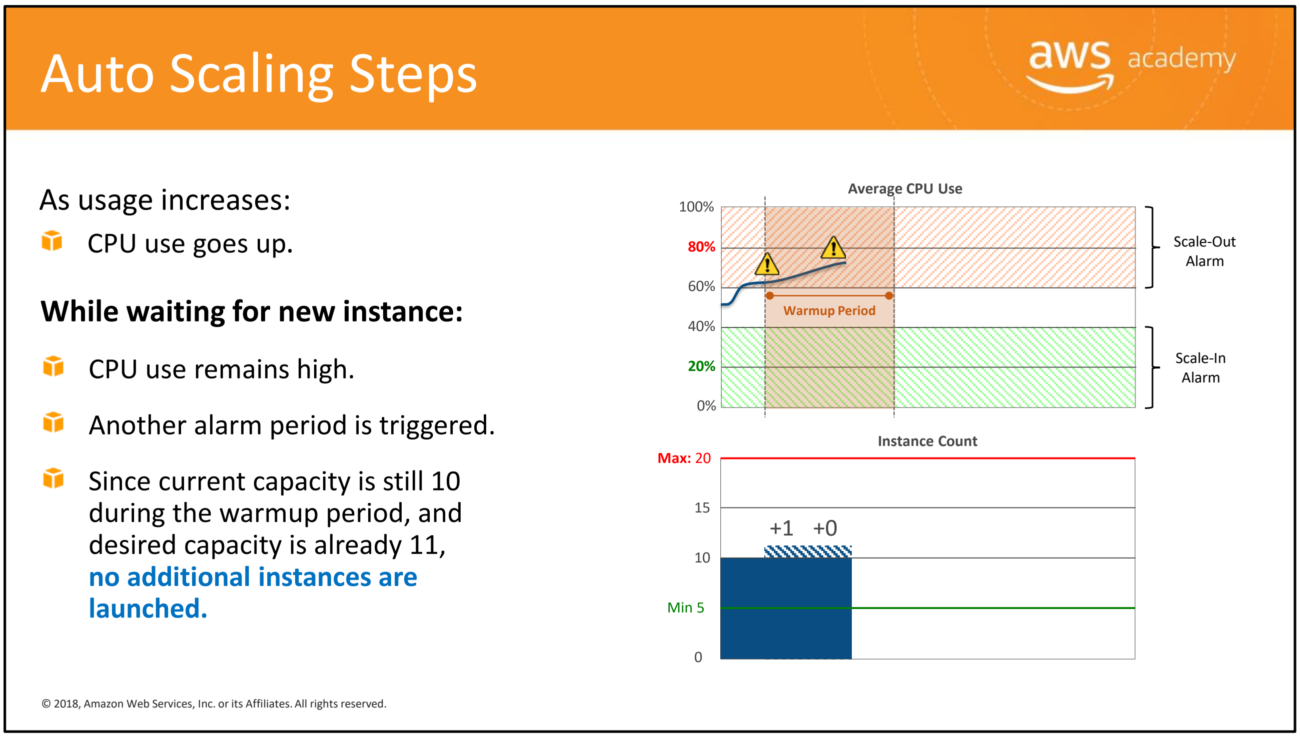
- CPU use goes up.

When CPU use is 60-80%:

- Scale-out alarm is triggered.
- Add 1 step policy is applied.
- New instance is launched but not added to the aggregated group metrics until after **warm up** period expires.



As usage increases, CPU use goes up. If the CPU use is between 60%-80%, the desired capacity of the group increases by 1 instance to 11 instances. This change is based on the second step adjustment of the scale-out policy, which will add 1 instance when the average CPU use is between 60%-80%. The new instance is launched, but not added, to the aggregated group metrics until after a warm-up period expires.



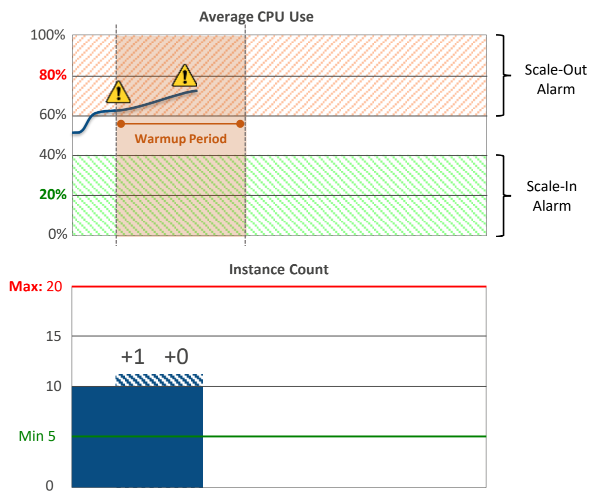
Auto Scaling Steps

As usage increases:

- CPU use goes up.

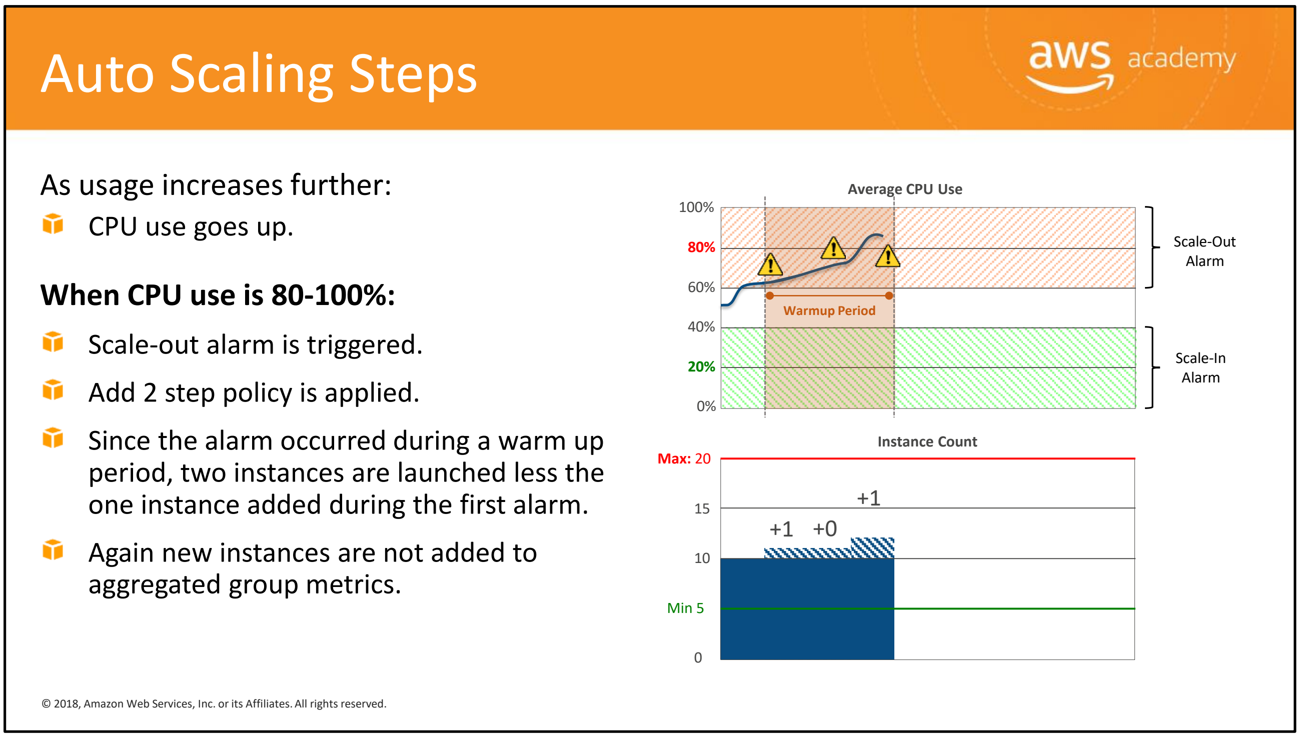
While waiting for new instance:

- CPU use remains high.
- Another alarm period is triggered.
- Since current capacity is still 10 during the warmup period, and desired capacity is already 11, **no additional instances are launched.**



With step scaling policies, you can specify the number of seconds that it takes for a newly launched instance to warm up. An instance is not counted toward the aggregated metrics of the Auto Scaling group until after its specific warm-up time expires. In this case, because the current capacity is still 10 during the warmup period, and desired capacity is already 11, no additional instances are launched.

While scaling out, we do not consider instances that are warming up as part of the current capacity of the group. Therefore, multiple alarm breaches that fall in the range of the same step adjustment result in a single scaling activity. This ensures that we don't add more instances than you need.



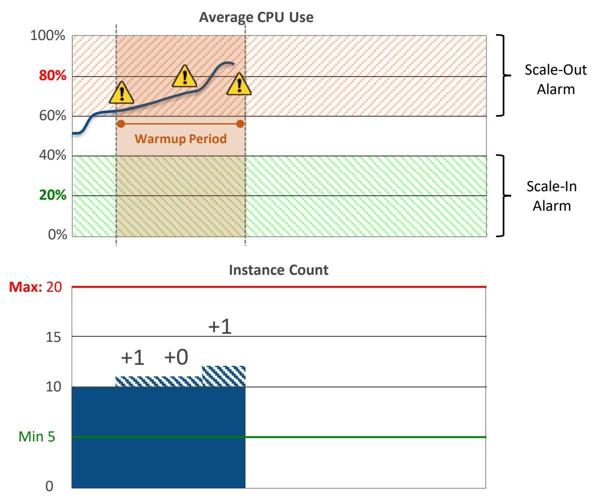
Auto Scaling Steps

As usage increases further:

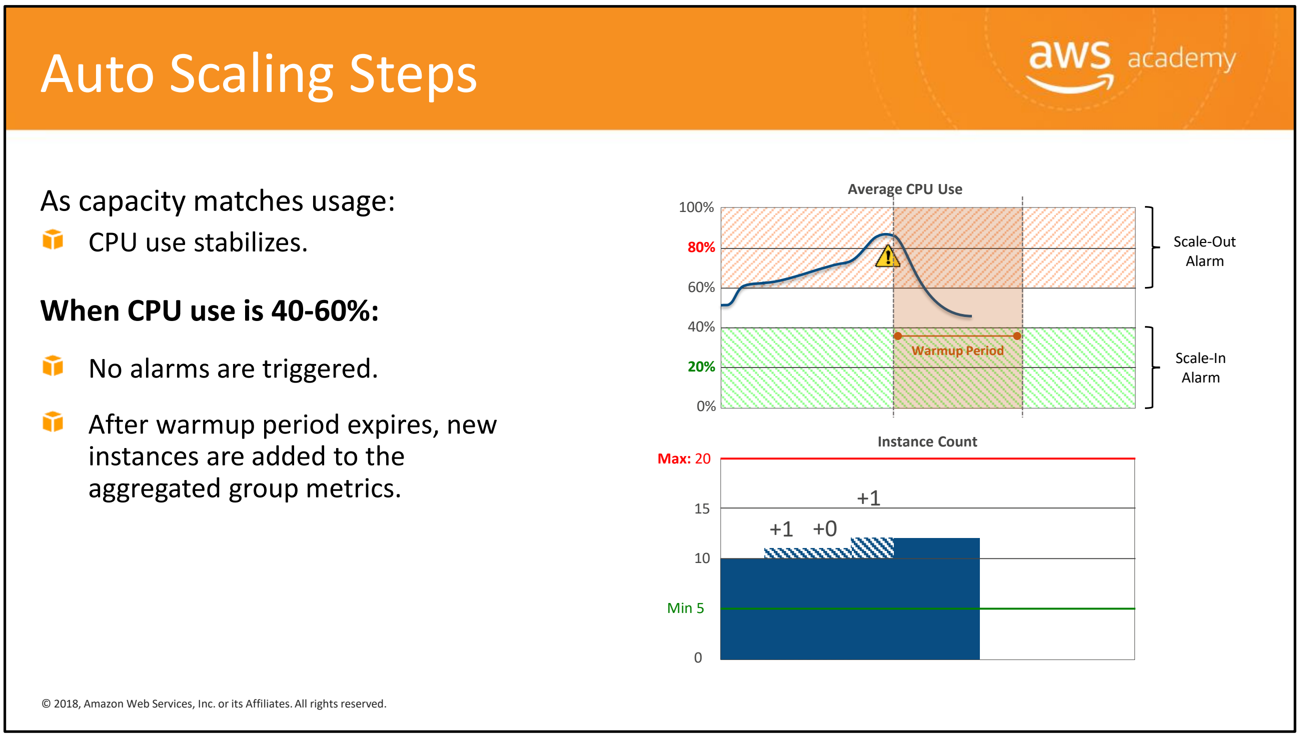
- CPU use goes up.

When CPU use is 80-100%:

- Scale-out alarm is triggered.
- Add 2 step policy is applied.
- Since the alarm occurred during a warm up period, two instances are launched less the one instance added during the first alarm.
- Again new instances not added to aggregated group metrics.



If the CPU use continues to rise and becomes to more than 80%, the second step adjustments would be triggered, and an additional server would be launched. The current capacity of this Auto Scaling group is 10, and the desired capacity was at 11, but the capacity would be increased to 12 when the second step adjustment is triggered. The warmup period is about to end for the first instance that was launched, but Auto Scaling will trigger a second warmup period to ensure that this instance can finish bootstrapping.



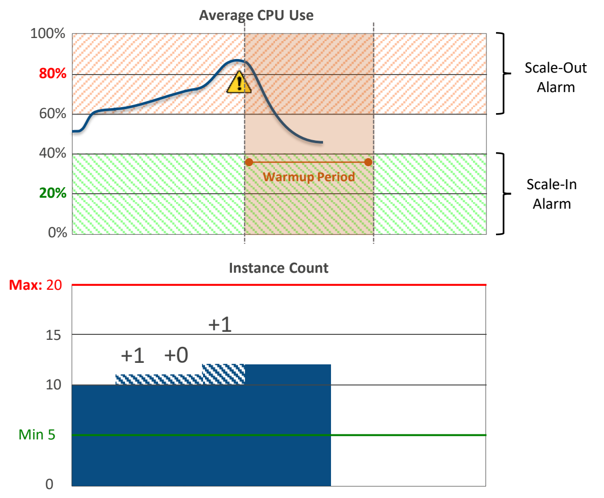
Auto Scaling Steps

As capacity matches usage:

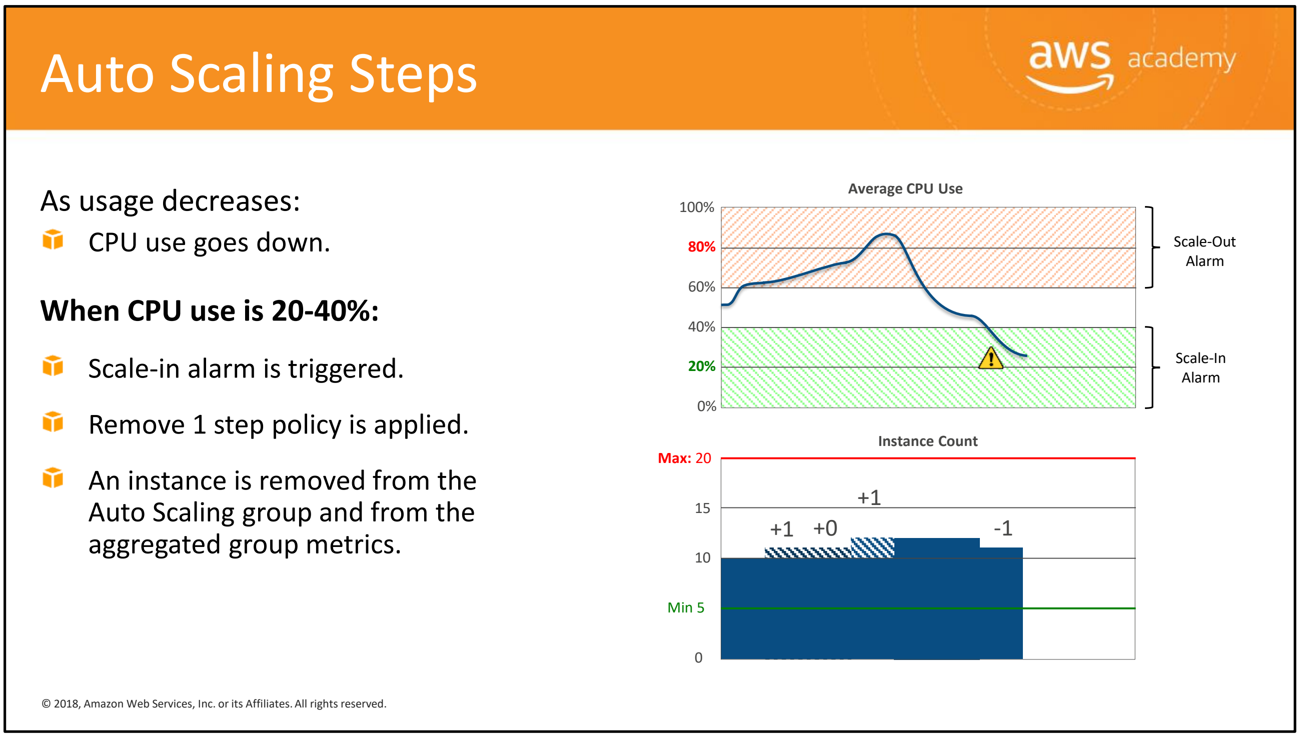
- CPU use stabilizes.

When CPU use is 40-60%:

- No alarms are triggered.
- After warmup period expires, new instances are added to the aggregated group metrics.



The instance from the first set of step adjustments was brought online and is responding to users' requests. This can help remove the spike in CPU use. After entering the warmup period from the second step adjustment, the second instance is added to the Auto Scaling group. Lowering the average CPU use to the target zone of greater than 40% and less than 60% then brings the current capacity and the desired capacity to 12 instances.



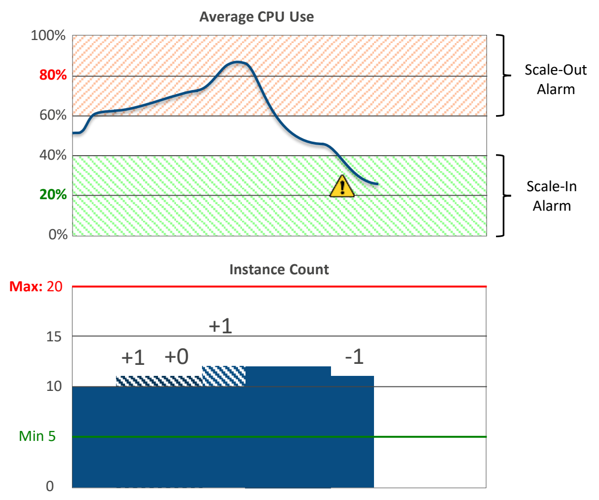
Auto Scaling Steps

As usage decreases:

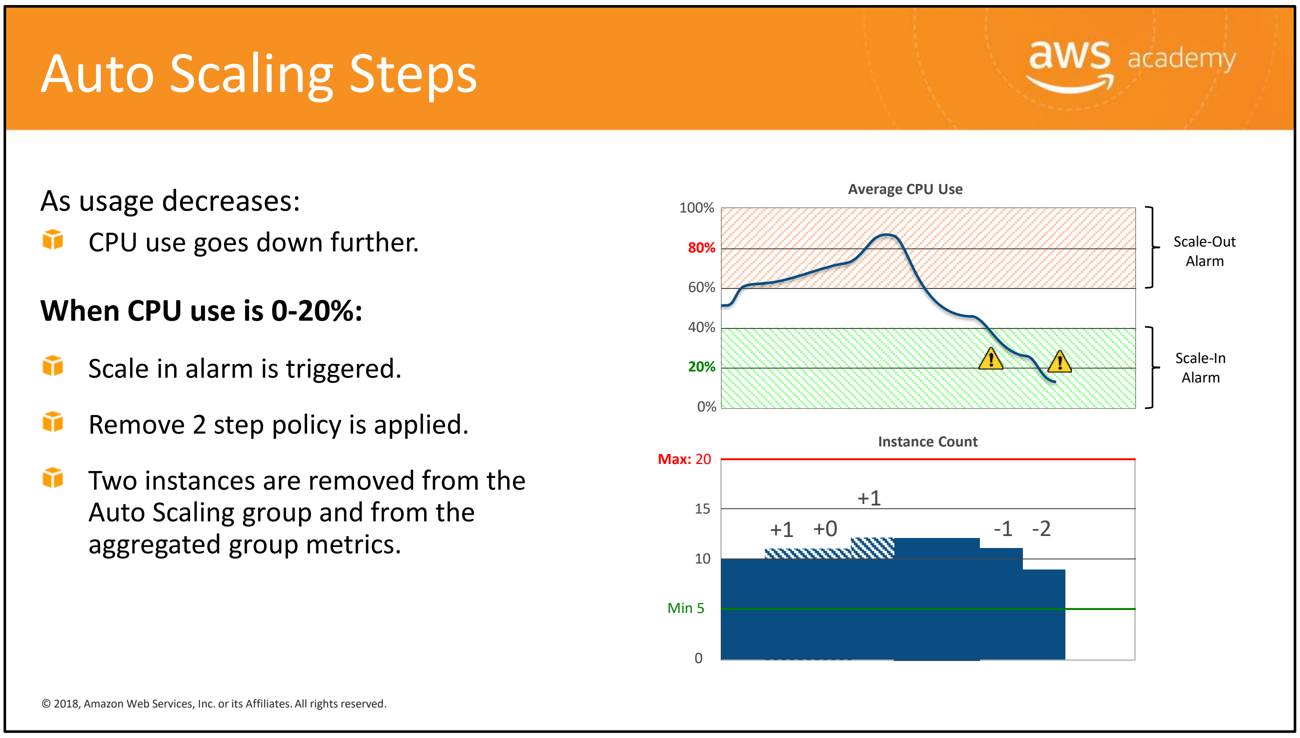
- CPU use goes down.

When CPU use is 20-40%:

- Scale-in alarm is triggered.
- Remove 1 step policy is applied.
- An instance is removed from the Auto Scaling group and from the aggregated group metrics.



When the usage spike subsides and average CPU use drops to 20%-40%, the desired capacity of the group decreases by 1 instance to 11 instances. This change is based on the third step adjustment of the policy, which removes 1 instance when the average CPU use is 20%-40%. Auto Scaling step adjustments do not support the use of “cooldown periods.” However, Elastic Load Balancing has a feature called connection draining. This feature is a period of time when Elastic Load Balancing stops sending requests to the instance that was identified for termination prior to deregistering it . After the time has elapsed, Elastic Load Balancing will forcefully close all open connections and terminate the targeted instance.



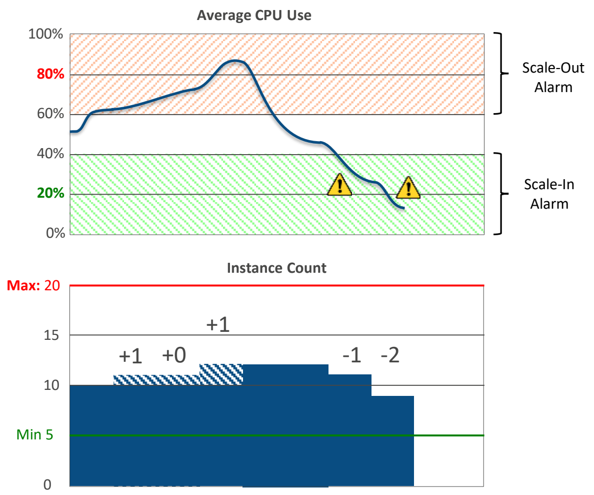
Auto Scaling Steps

As usage decreases:

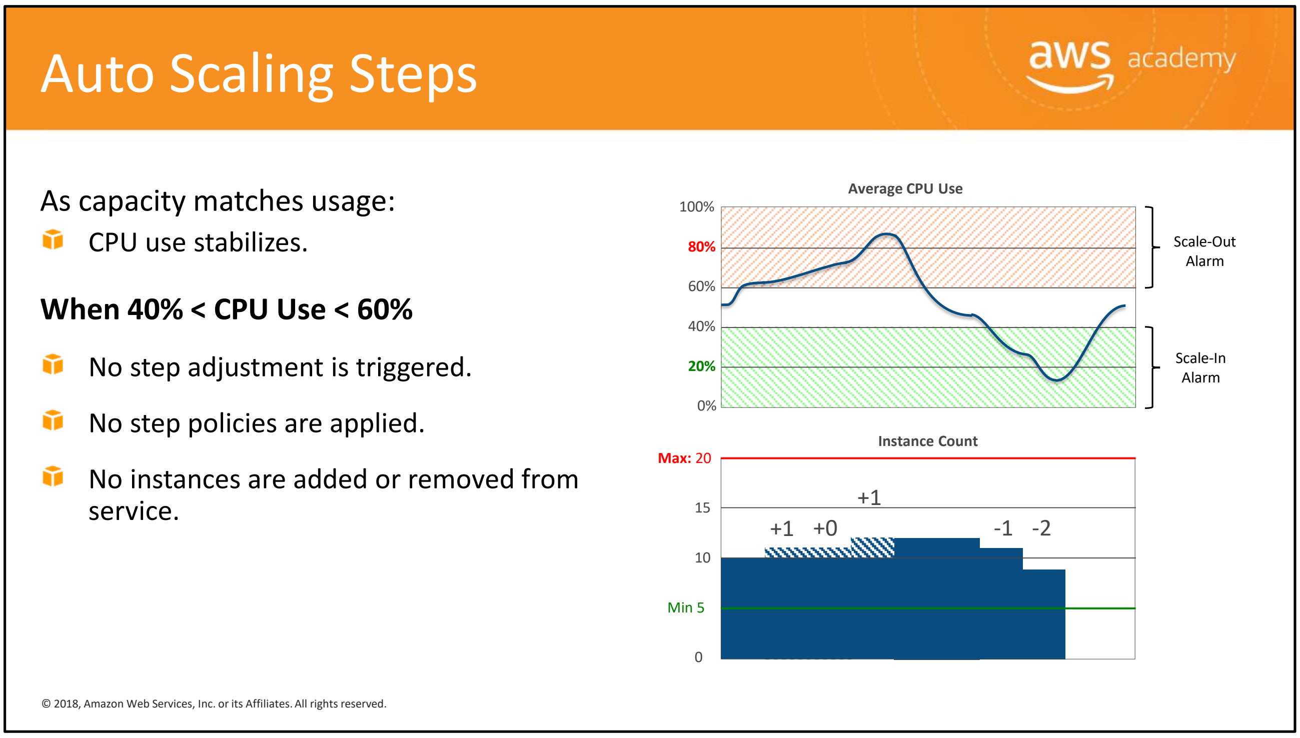
- CPU use goes down further.

When CPU use is 0-20%:

- Scale in alarm is triggered.
- Remove 2 step policy is applied.
- Two instances are removed from the Auto Scaling group and from the aggregated group metrics.



It would be possible to set up a step adjustment to scale in more aggressively if the load continues to drop. Just remember that it is better to scale up quickly, and scale down slowly. If you terminate an instance too quickly, another spike might take place and cause more servers to be created. If this situation occurs, it will take a short time until it's able to accept the spike in traffic. In this example, the scale-in step was triggered, and two more instances will be terminated while the average CPU usage continues to drop.



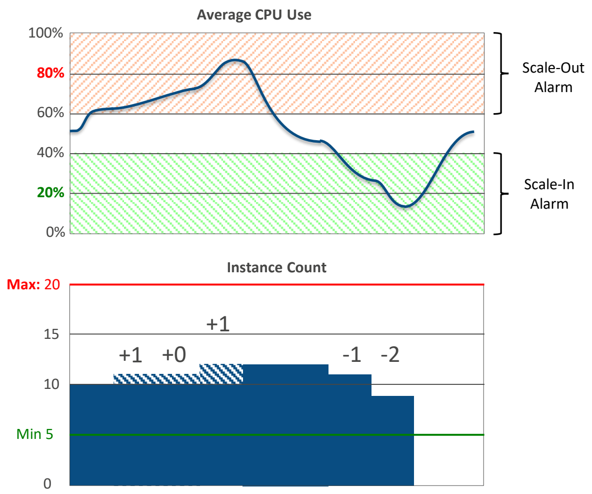
Auto Scaling Steps

As capacity matches usage:

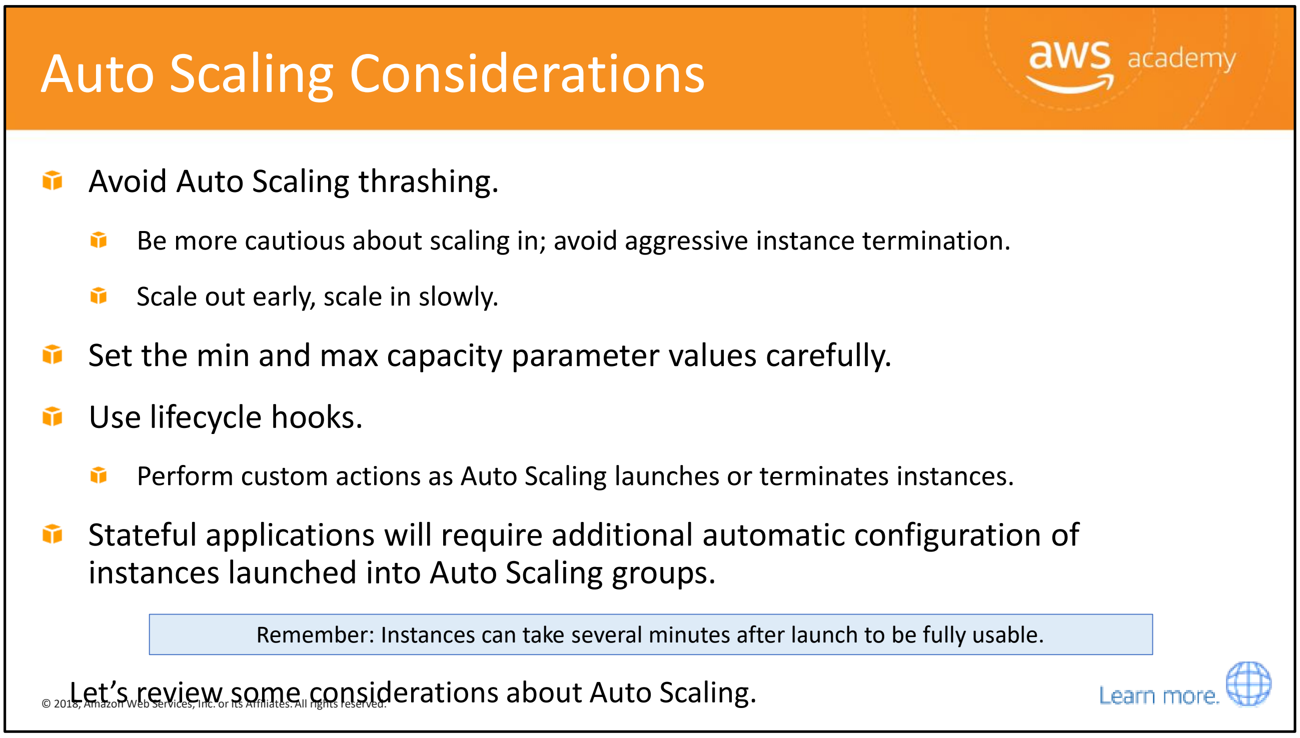
- CPU use stabilizes.

When 40% < CPU Use < 60%

- No step adjustment is triggered.
- No step policies are applied.
- No instances are added or removed from service.



So now CPU usage has stabilized between 40%-60%. With this state, no step adjustment is triggered, and no policy is applied. No instances are added or removed. Be sure to scale back very slowly, because you do not want to create something that causes CPU thrashing.



Auto Scaling Considerations

- Avoid Auto Scaling thrashing.
  - Be more cautious about scaling in; avoid aggressive instance termination.
  - Scale out early, scale in slowly.
- Set the min and max capacity parameter values carefully.
- Use lifecycle hooks.
  - Perform custom actions as Auto Scaling launches or terminates instances.
- Stateful applications will require additional automatic configuration of instances launched into Auto Scaling groups.

Remember: Instances can take several minutes after launch to be fully usable.

Learn more.

Let's review some considerations about Auto Scaling.

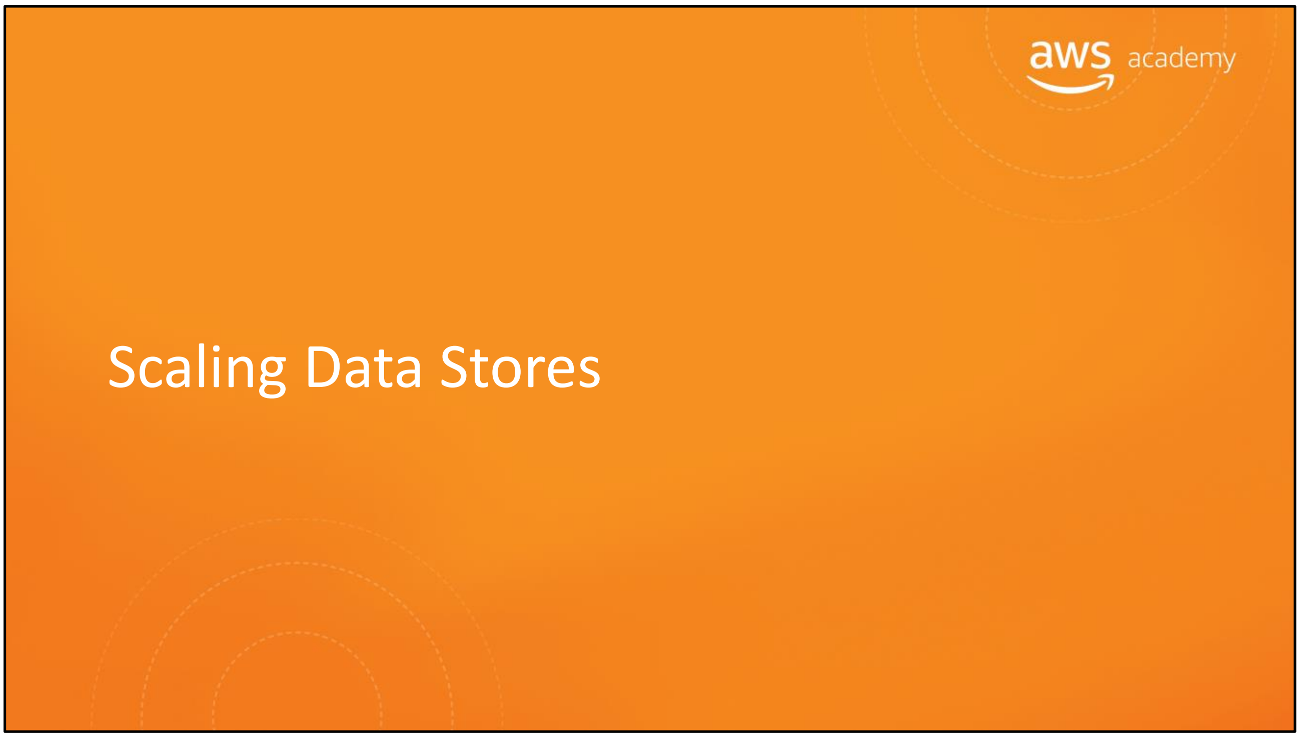
You want to avoid thrashing when you use Auto Scaling. To avoid thrashing, be more cautious about scaling in, and avoid aggressive instance termination. "Scaling in" means that you decrease the computing capacity due to low use. If you have an unpredictable workload, it is possible that the workload might spike right after you scale in. Depending on your launch configuration, it might take a few minutes for Auto Scaling to launch additional instances. Therefore, you should scale out early, and scale in slowly, rather than aggressively. Being cost-conscious is good, but your applications should not suffer.

Set the minimum and maximum capacity parameter values carefully. The desired capacity is different from the minimum capacity. The desired capacity of an Auto Scaling group is the default number of instances that should be running. The minimum capacity of a group is the fewest number of instances the group can have running. For instance, a group with a desired capacity of four will run four instances if no scaling policies are in effect. If the group has a minimum capacity of two, then any scale-in policy that goes into effect cannot reduce the total number of instances in the group below two.

Use lifecycle hooks to perform custom actions when Auto Scaling launches or terminates instances. An example of a lifecycle hook would be when someone uploads something to your Amazon S3 bucket.

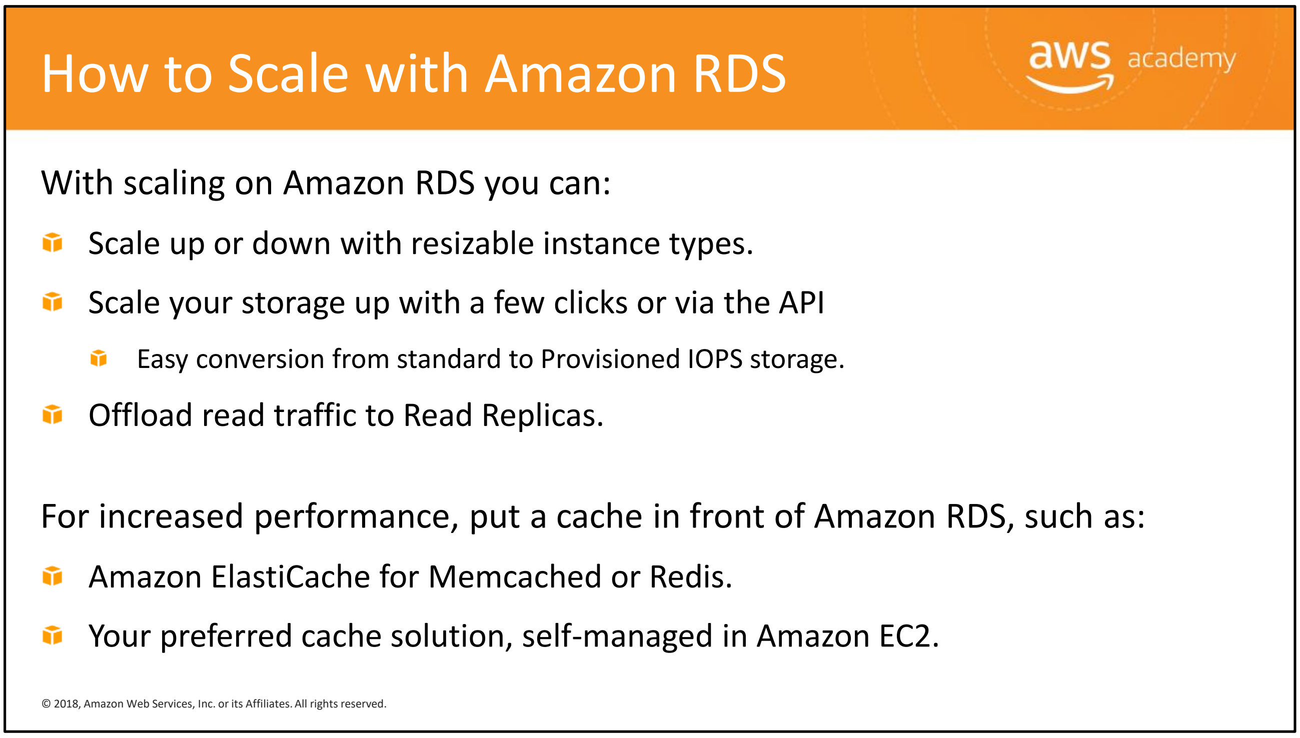
Stateful applications require additional automatic configuration of instances that are launched into Auto Scaling groups. Remember that instances can take several minutes after launch before they are fully usable.

Select the link to learn more about Auto Scaling lifecycle hooks.
http://docs.aws.amazon.com/AutoScaling/latest/DeveloperGuide/lifecycle-hooks.html



Scaling Data Stores

Now, in Part 4, let's take a look at scaling data stores.



How to Scale with Amazon RDS

aws academy

With scaling on Amazon RDS you can:

- Scale up or down with resizable instance types.
- Scale your storage up with a few clicks or via the API
  - Easy conversion from standard to Provisioned IOPS storage.
- Offload read traffic to Read Replicas.

For increased performance, put a cache in front of Amazon RDS, such as:

- Amazon ElastiCache for Memcached or Redis.
- Your preferred cache solution, self-managed in Amazon EC2.

With scaling on Amazon Relational Database Service , you can:

- Scale up or down with resizable instance types.
- Scale your storage up with through the console or via the API. This is an easy conversion from standard to provisioned IOPS storage.

- And offload read traffic to read replicas.

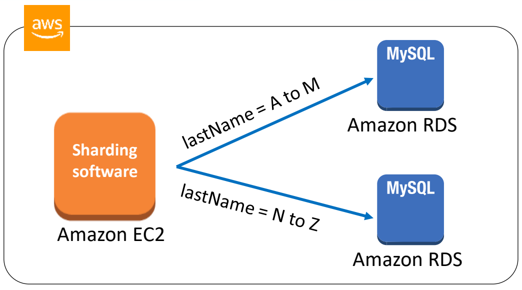
For increased performance, you can put a cache in front of Amazon Relational Database Service, such as:

- Amazon ElastiCache for Memcached or Redis.
- Or your preferred cache solution, which you self-manage in Amazon EC2.

Database Sharding

aws academy

- Without shards, all data resides in one partition.
  - Example: Users by last name, A to Z, in one database.
- With **sharding**, split your data into **large chunks** (shards).
  - Example: Users by last name, A through M, in one database; N through Z in another database.
- In many circumstances, sharding gives you **higher performance** and **better operating efficiency**.

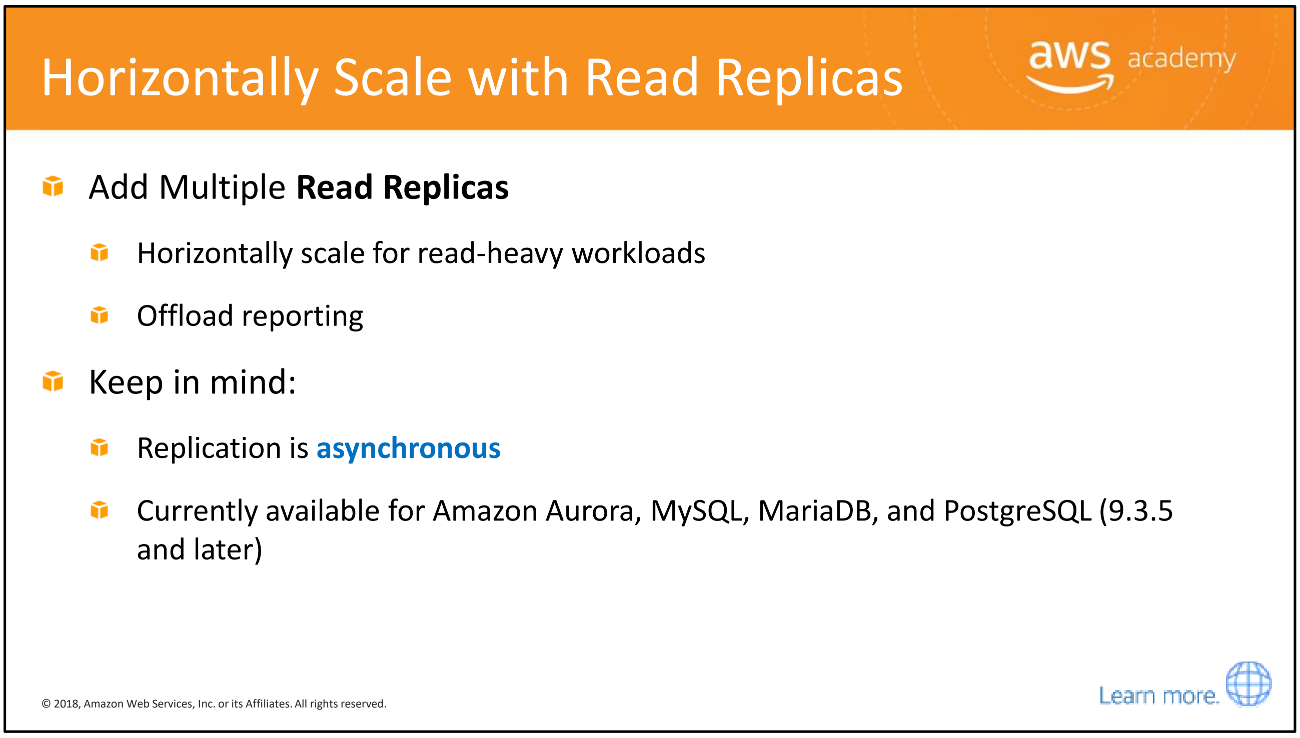


Learn more.

You can also use database sharding, depending on the type of data you have. Without shards, all data resides in one partition.
Sharding is a technique for improving the performance of writing with multiple database servers. Fundamentally, sharding is when you prepare databases with identical structures, and divide them—using appropriate table columns as keys—to distribute writing processes. By using the RDBMS service provided in the AWS Cloud, you can perform sharding to achieve an increase in availability and performance, and also gain better operating efficiency.

You can use Amazon RDS to shard backend databases. To do so, install sharding software, such as MySQL server combined with a Spider Storage Engine on an Amazon EC2 instance. Then, prepare multiple Amazon RDS instances and use them as the backend databases for sharding. You can distribute the Amazon RDS instances to multiple Regions.

For more information, refer to the "Sharding Write Pattern" page at Cloud Design Pattern.
http://en.clouddesignpattern.org/index.php/CDP:Sharding_Write_Pattern



Horizontally Scale with Read Replicas

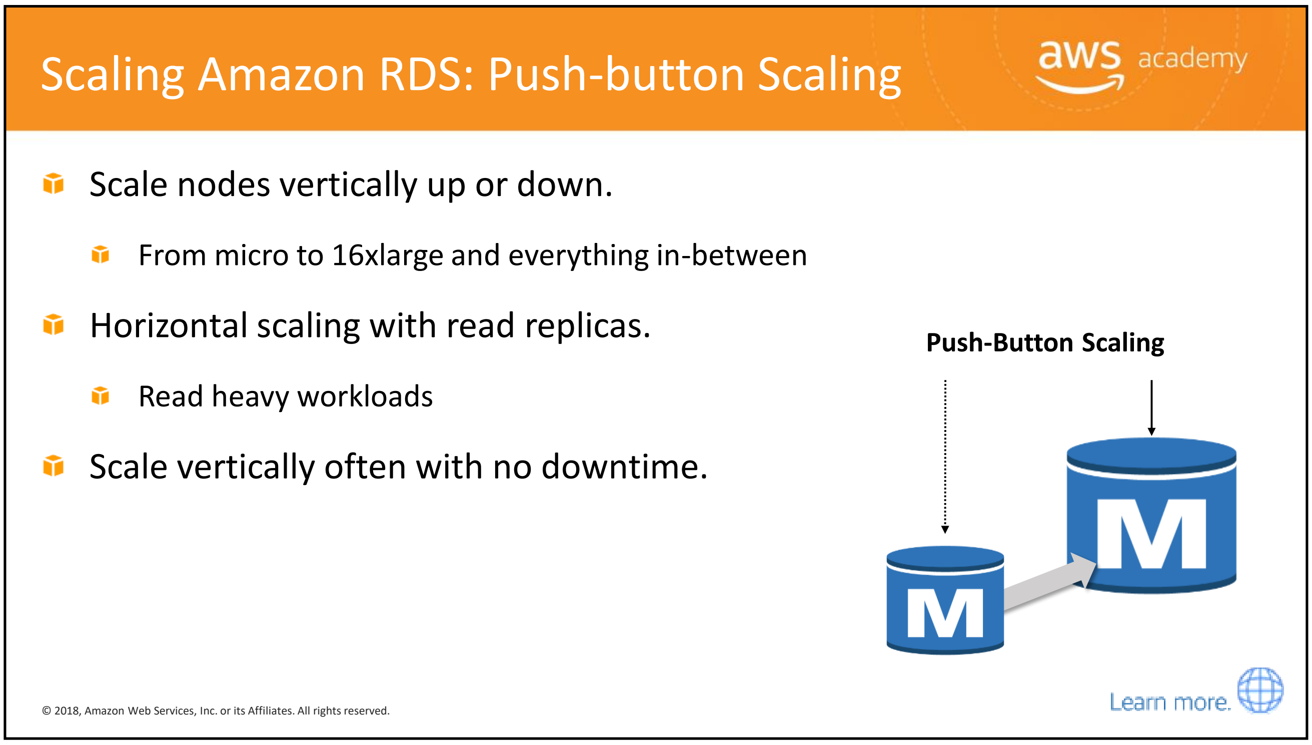
- Add Multiple **Read Replicas**
  - Horizontally scale for read-heavy workloads
  - Offload reporting
- Keep in mind:
  - Replication is **asynchronous**
  - Currently available for Amazon Aurora, MySQL, MariaDB, and PostgreSQL (9.3.5 and later)

Learn more.

How can you horizontally scale with read replicas? If most of your traffic only involves read operations, you can add multiple read replicas that will be asynchronously replicated with your main database. If you perform reports that could put an extra load on your production database, you can offload these operations to a read replica.

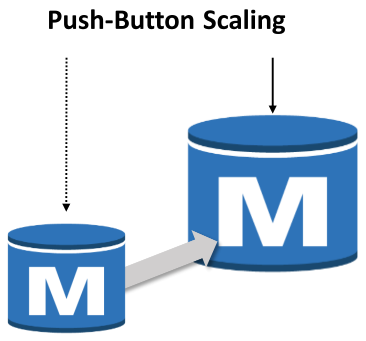
Keep in mind that replication is asynchronous. It’s currently available for Amazon Aurora, MySQL, MariaDB, and PostgreSQL version 9.3.5 and later.

Read replicas with PostgreSQL have specific requirements. To learn more, go to the AWS RDS documentation page about working with read replicas. http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER_ReadRepl.html#USER_ReadRepl.PostgreSQL



Scaling Amazon RDS: Push-button Scaling

- Scale nodes vertically up or down.
  - From micro to 16xlarge and everything in-between
- Horizontal scaling with read replicas.
  - Read heavy workloads
- Scale vertically often with no downtime.



Learn more.

With the Amazon RDS APIs or through the AWS Management Console, you can scale the compute and memory resources that power your deployment up or down from micro to 16xlarge and everything in between. Scaling operations typically finish within a few minutes.

You can scale horizontally with read replicas.
You can often scale vertically with no downtime.

As your storage requirements grow, you can provision additional storage on-the-fly with no downtime. If you are using Amazon RDS PIOPS—with the exception of Amazon RDS with SQL Server—you can also scale the throughput of your DB instance by specifying the IOPS rate between 1,000 IOPS-30,000 IOPS in 1,000 IOPS increments, and storage from 100 GB and 6 TB .

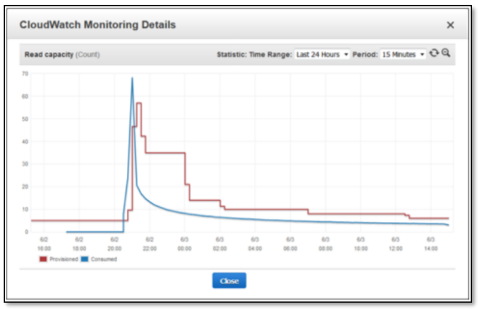
Amazon RDS for SQL Server does not currently support increasing storage or IOPS of an existing SQL Server DB instance.
There is minimal downtime when you scale up on a Multi-AZ environment because the standby database gets upgraded first; then, a failover will occur to the newly sized database. A Single-AZ instance will be unavailable during the scale operation.

To learn more, go to the Amazon RDS documentation on modifying an Amazon RDS instance.

http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Overview.DBInstance.Modifying.html#USER_ModifyInstance.Settings

Auto Scaling for DynamoDB

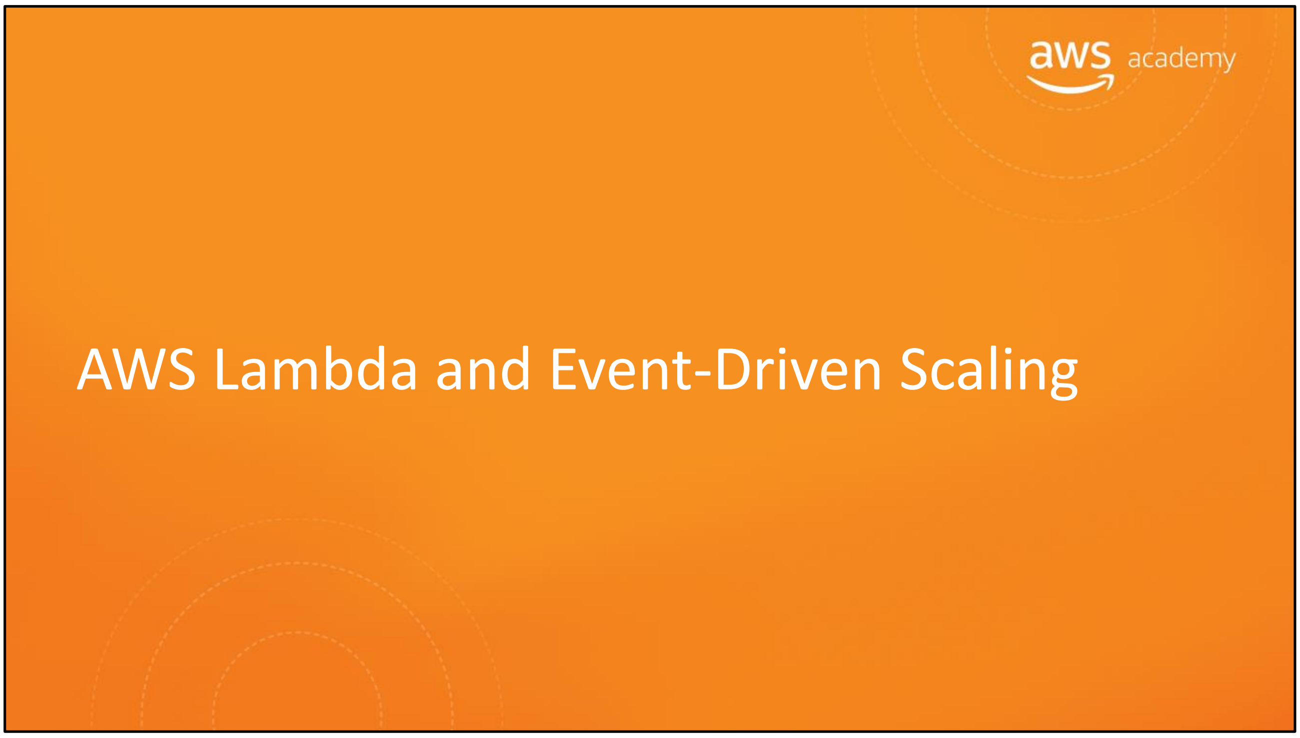
- Automate capacity management for tables and global secondary indexes.
- Specify the desired target use and provide upper and lower bounds for read and write capacity.
- DynamoDB monitors throughput consumption using Amazon CloudWatch alarms
  - Then will adjust provisioned capacity up or down as needed.
- Default for all new tables and indexes.



When you create a new DynamoDB table using the AWS Management Console, the table will have Auto Scaling enabled by default. DynamoDB Auto Scaling automatically adjusts read and write throughput capacity, in response to dynamically changing request volumes, with zero downtime. With DynamoDB Auto Scaling, you simply set your throughput utilization target, and minimum and maximum limits. Auto Scaling takes care of the rest.

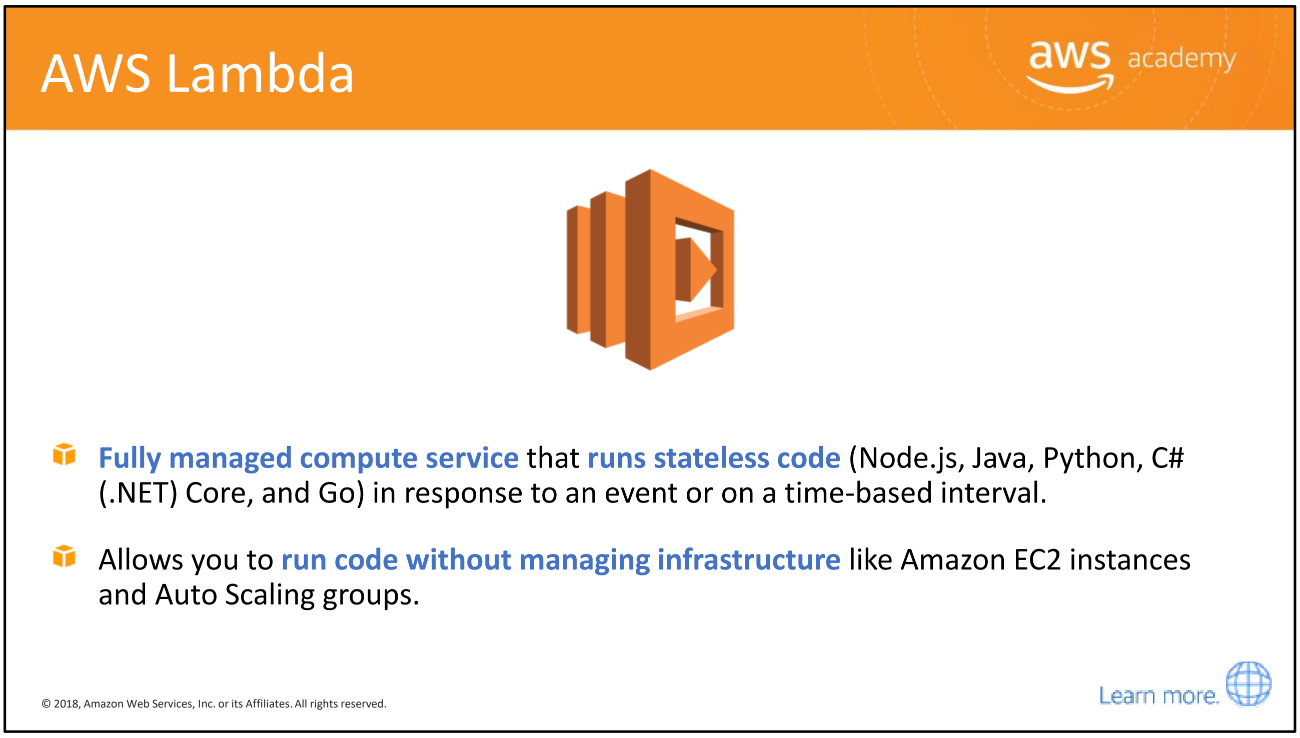
DynamoDB Auto Scaling works with Amazon CloudWatch to continuously monitor actual throughput consumption, and it scales capacity up or down automatically when actual usage deviates from your target. Auto Scaling can be enabled for new and existing tables, and global secondary indexes. You can enable Auto Scaling in the AWS Management Console, where you also have full visibility into scaling activities. You can also manage DynamoDB Auto Scaling programmatically, using the AWS Command Line Interface and the AWS Software Development Kits.

There is no additional cost to use DynamoDB Auto Scaling beyond what you already pay for DynamoDB and CloudWatch alarms. DynamoDB Auto Scaling is available in all AWS Regions.



AWS Lambda and Event-Driven Scaling

Introducing Part 5: AWS Lambda and Event-Driven Scaling.



AWS Lambda

- **Fully managed compute service** that **runs stateless code** (Node.js, Java, Python, C# (.NET) Core, and Go) in response to an event or on a time-based interval.
- Allows you to **run code without managing infrastructure** like Amazon EC2 instances and Auto Scaling groups.

Learn more.

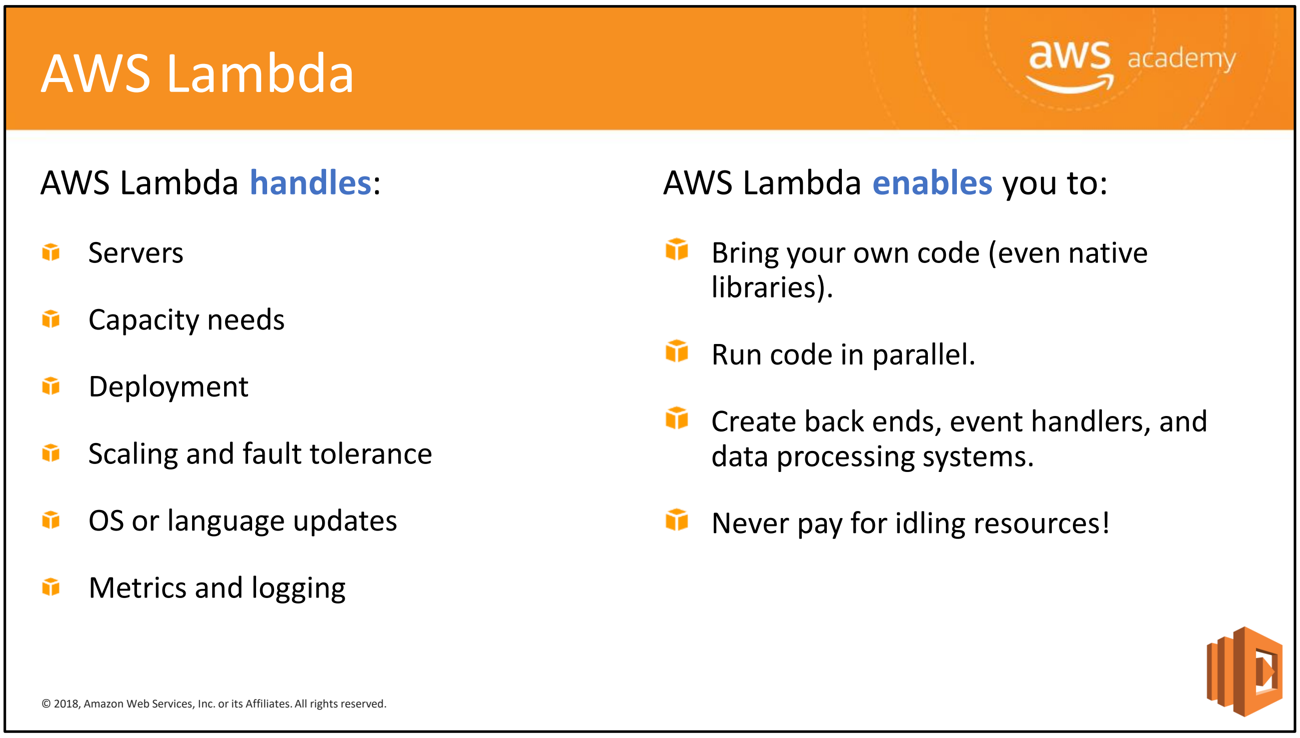
AWS Lambda is a fully managed compute service that runs stateless code in response to an event or on a time-based interval.

It allows you to run code without managing infrastructure, like Amazon EC2 instances and Auto Scaling groups.

AWS Lambda lets you run code without provisioning or managing servers. AWS Lambda runs your code on a high-availability compute infrastructure. It performs all of the administration of the compute resources, including server and operating system maintenance, capacity provisioning, automatic scaling, code monitoring, and logging. You only need to supply your code in one of the languages that AWS Lambda supports, which includes Node.js, Java, Python, C# or .NET Core, and Go.

For a list of the current supported runtime versions for AWS Lambda, go to the product documentation page for the Lambda execution environment.

https://docs.aws.amazon.com/lambda/latest/dg/current-supported-versions.html



AWS Lambda

AWS Lambda **handles**:

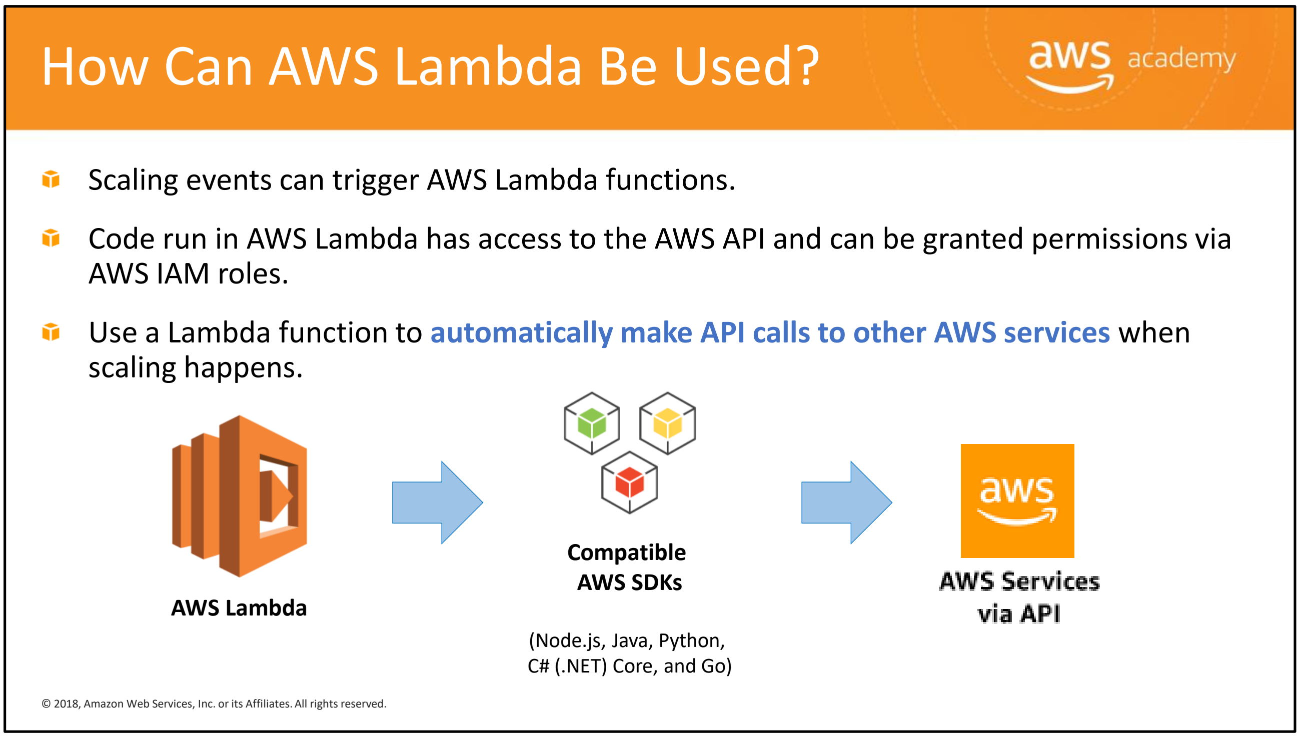
- Servers
- Capacity needs
- Deployment
- Scaling and fault tolerance
- OS or language updates
- Metrics and logging

AWS Lambda **enables** you to:

- Bring your own code (even native libraries).
- Run code in parallel.
- Create back ends, event handlers, and data processing systems.
- Never pay for idling resources!

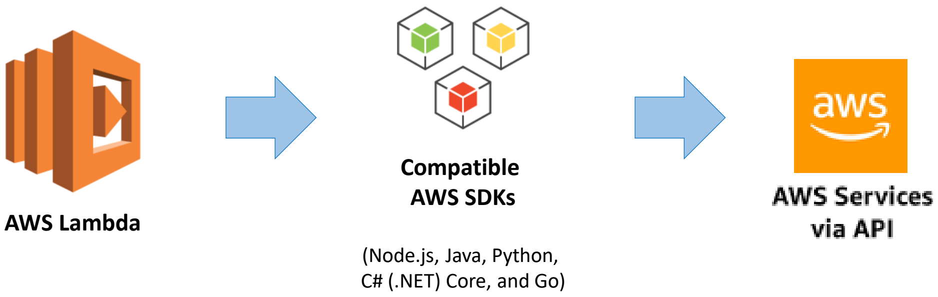
AWS Lambda handles:
- Servers
- Capacity needs
- Deployment
- Scaling and fault tolerance
- OS or language updates
- Metrics and logging

AWS Lambda enables you to:
- Bring your own code, even native libraries.
- Run code in parallel.
- Create backends, event handlers, and data processing systems.
- And never pay for idling resources!



How Can AWS Lambda Be Used?

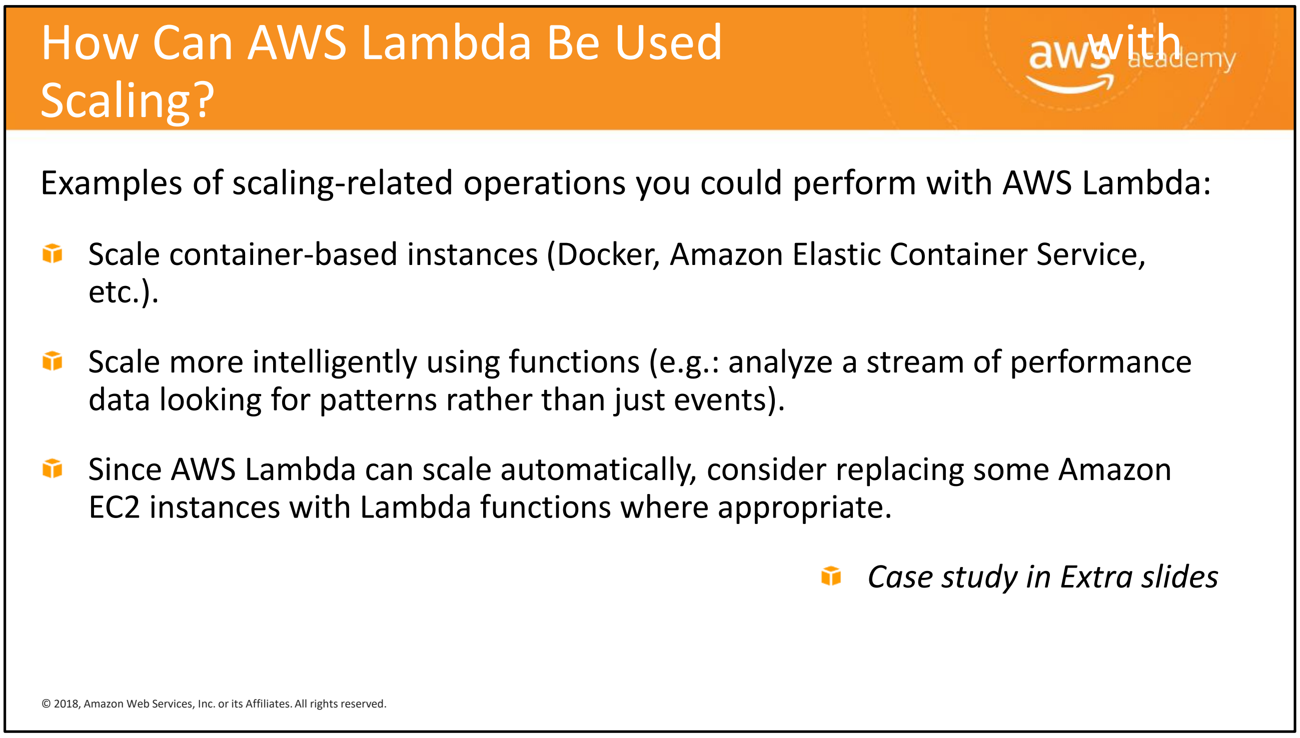
- Scaling events can trigger AWS Lambda functions.
- Code run in AWS Lambda has access to the AWS API and can be granted permissions via AWS IAM roles.
- Use a Lambda function to **automatically make API calls to other AWS services** when scaling happens.



Scaling events can trigger AWS Lambda functions. In addition, code that you run in AWS Lambda has access to the AWS API, and can be granted permissions via AWS IAM roles.

This means you can use a Lambda function to automatically make API calls to other AWS services when scaling happens.

Compatible AWS SDKs include Node.js, Java, Python, C# or .NET Core, and Go.



How Can AWS Lambda Be Used Scaling?

Examples of scaling-related operations you could perform with AWS Lambda:

- Scale container-based instances (Docker, Amazon Elastic Container Service, etc.).
- Scale more intelligently using functions (e.g.: analyze a stream of performance data looking for patterns rather than just events).
- Since AWS Lambda can scale automatically, consider replacing some Amazon EC2 instances with Lambda functions where appropriate.

- *Case study in Extra slides*

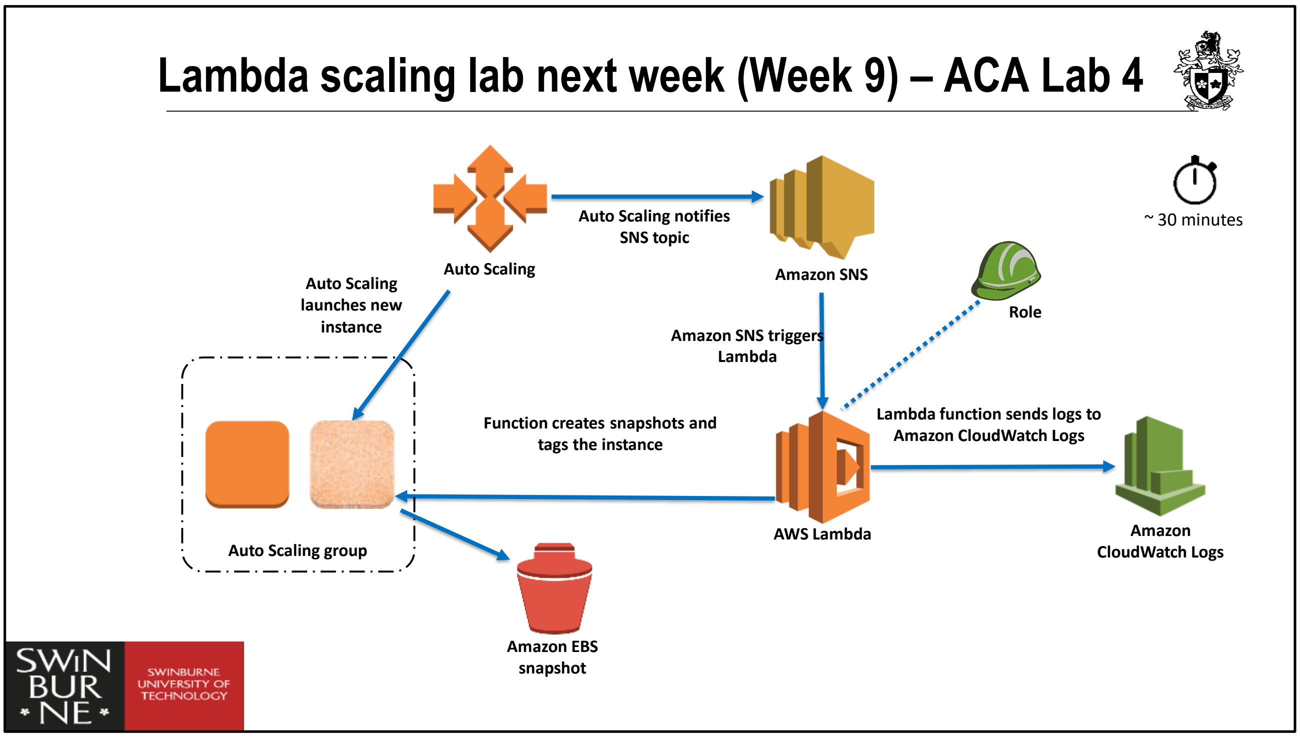
How can AWS Lambda be used with scaling?

Examples of scaling-related operations you could perform with AWS Lambda include:

• Scaling container-based instances, such as Docker, Amazon Elastic Container Service, etc..

• Scaling more intelligently by using functions, such as analyzing a stream of performance data that looks for patterns instead of events.

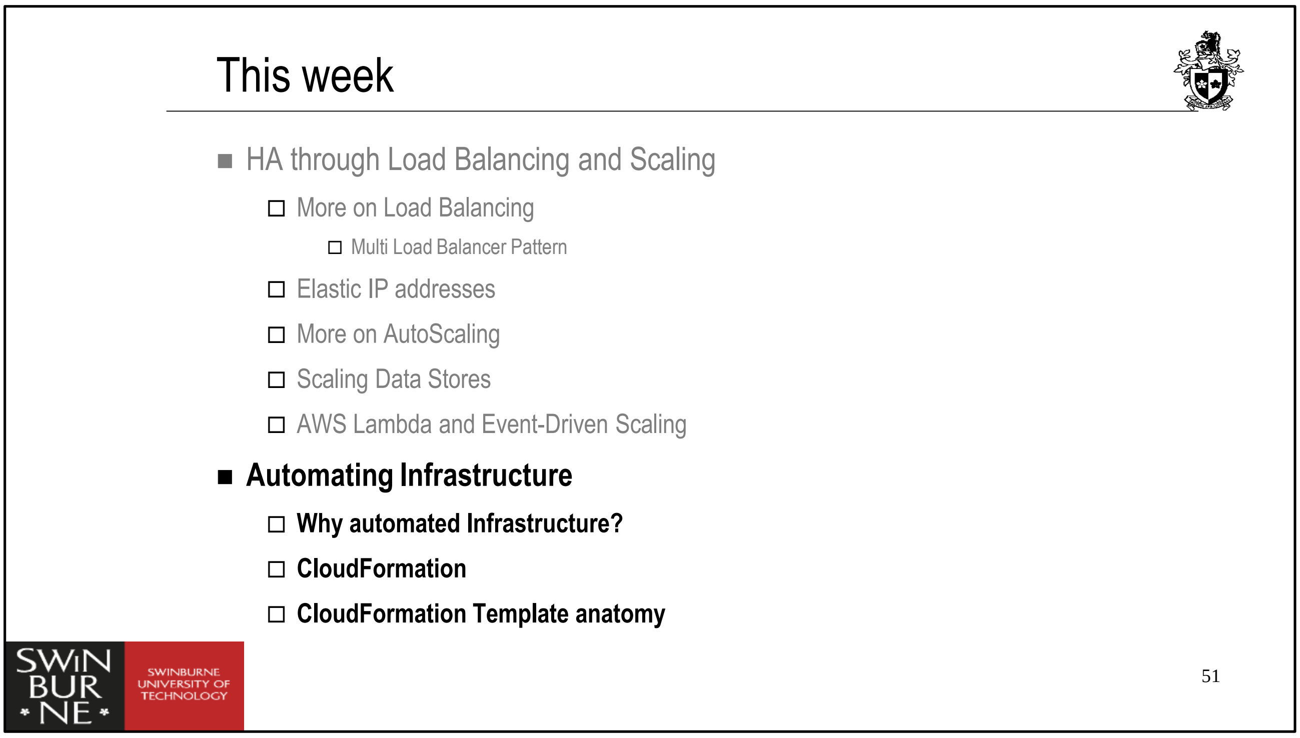
• Replacing some Amazon EC2 instances with Lambda functions where it’s appropriate because Lambda can scale automatically.



Lambda scaling lab next week (Week 9) – ACA Lab 4

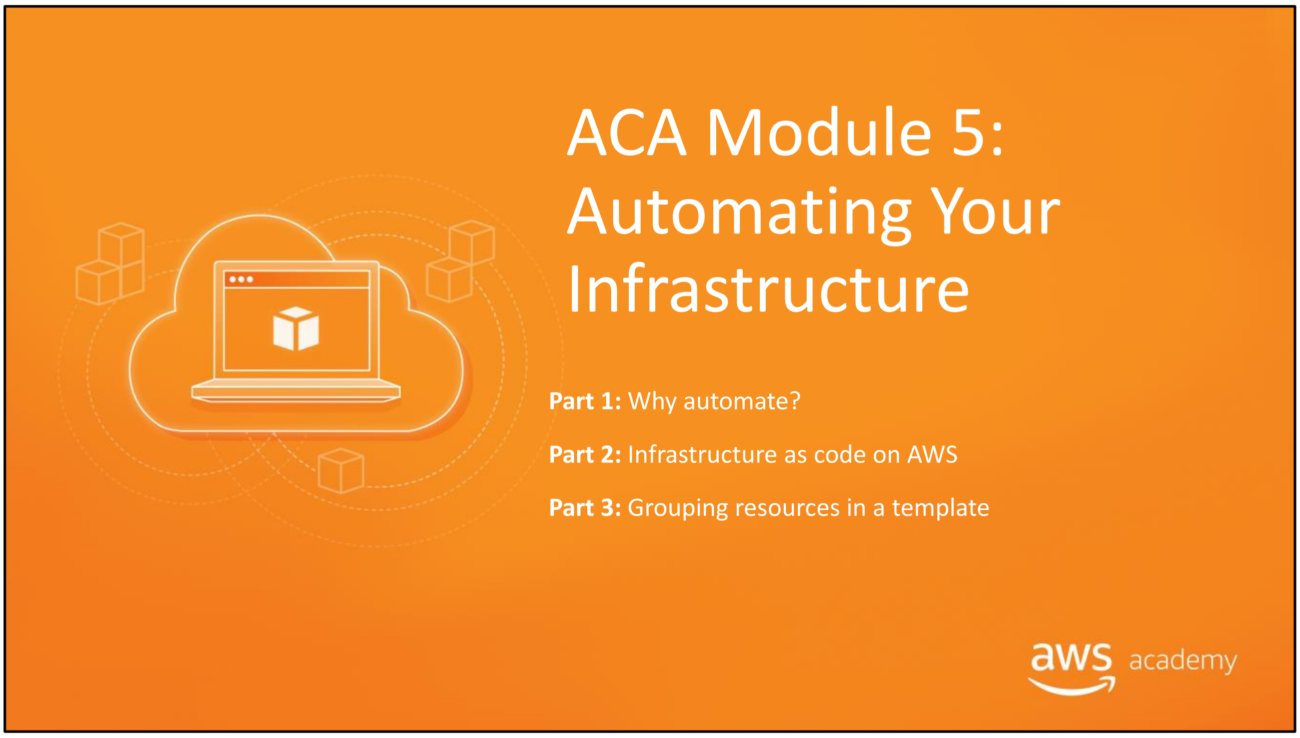
~ 30 minutes
Auto Scaling
Auto Scaling notifies SNS topic
Amazon SNS
Role
Auto Scaling launches new instance
Amazon SNS triggers Lambda
Function creates snapshots and tags the instance
Lambda function sends logs to Amazon CloudWatch Logs
Auto Scaling group
AWS Lambda
Amazon CloudWatch Logs
Amazon EBS snapshot
SWINBURNE UNIVERSITY OF TECHNOLOGY

This slide shows the final product for this lab.



This week

- HA through Load Balancing and Scaling
  - More on Load Balancing
    - Multi Load Balancer Pattern
  - Elastic IP addresses
  - More on AutoScaling
  - Scaling Data Stores
  - AWS Lambda and Event-Driven Scaling
- **Automating Infrastructure**
  - **Why automated Infrastructure?**
  - **CloudFormation**
  - **CloudFormation Template anatomy**



ACA Module 5: Automating Your Infrastructure

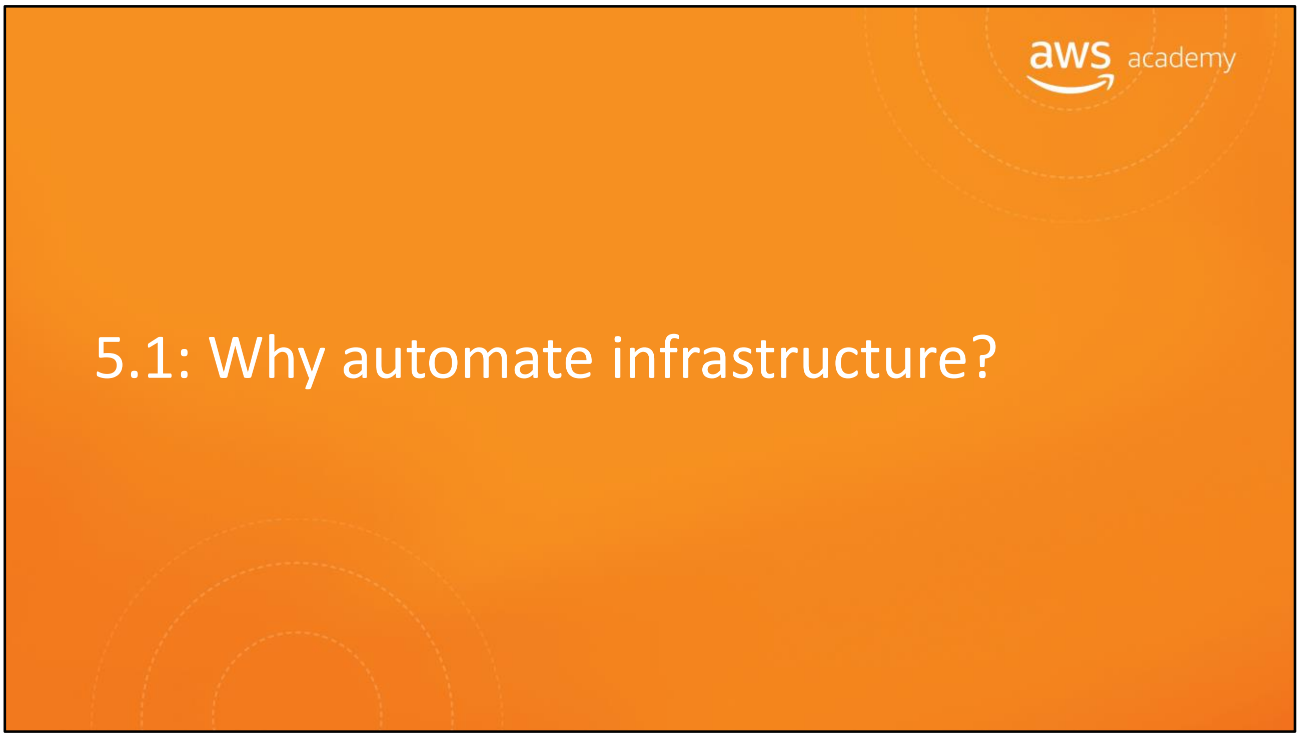
Part 1: Why automate?

Part 2: Infrastructure as code on AWS

Part 3: Grouping resources in a template

aws academy

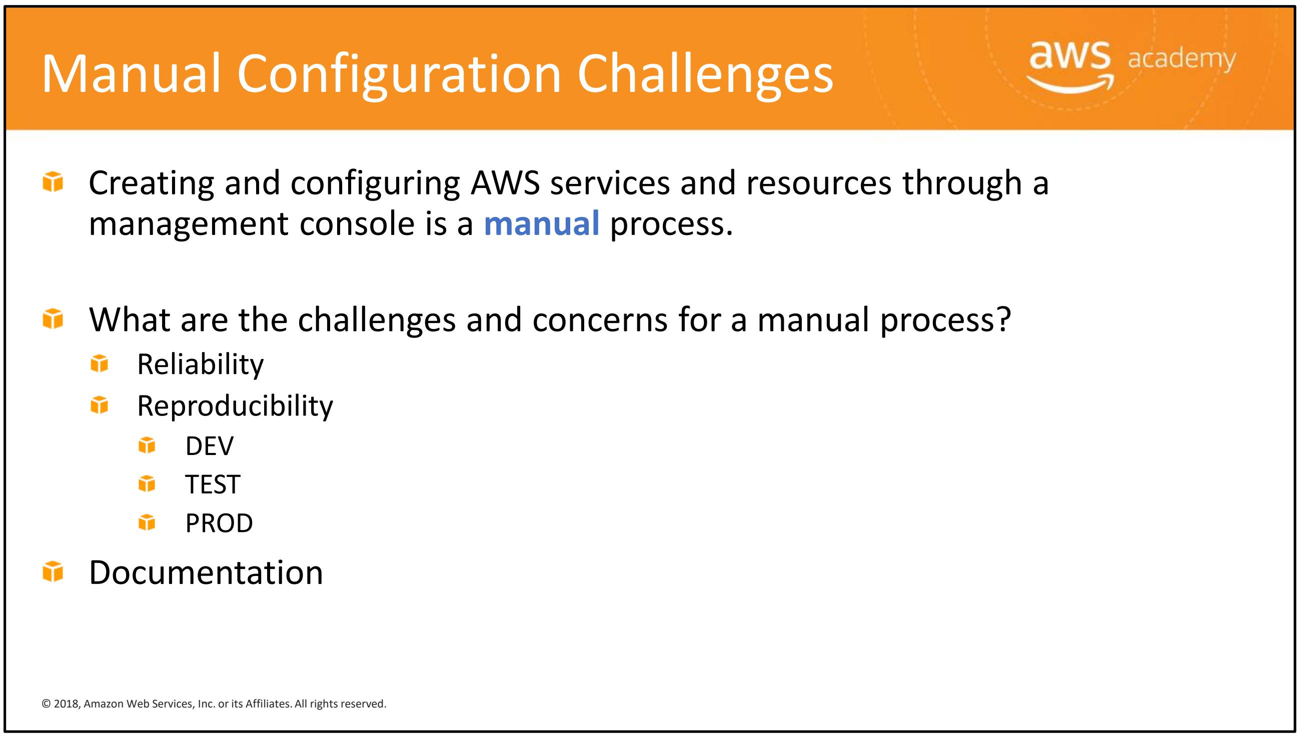
Welcome to Module 5: Automating Your Infrastructure. This module provides an in-depth analysis of microservices and serverless architectures to explain how they can make the infrastructure more resilient and cost effective. The goal of this module is to teach the fundamental concepts of these non-traditional approaches to deploying applications.



5.1: Why automate infrastructure?

aws academy

Introducing *Part 1: Manual Configuration.*



Manual Configuration Challenges

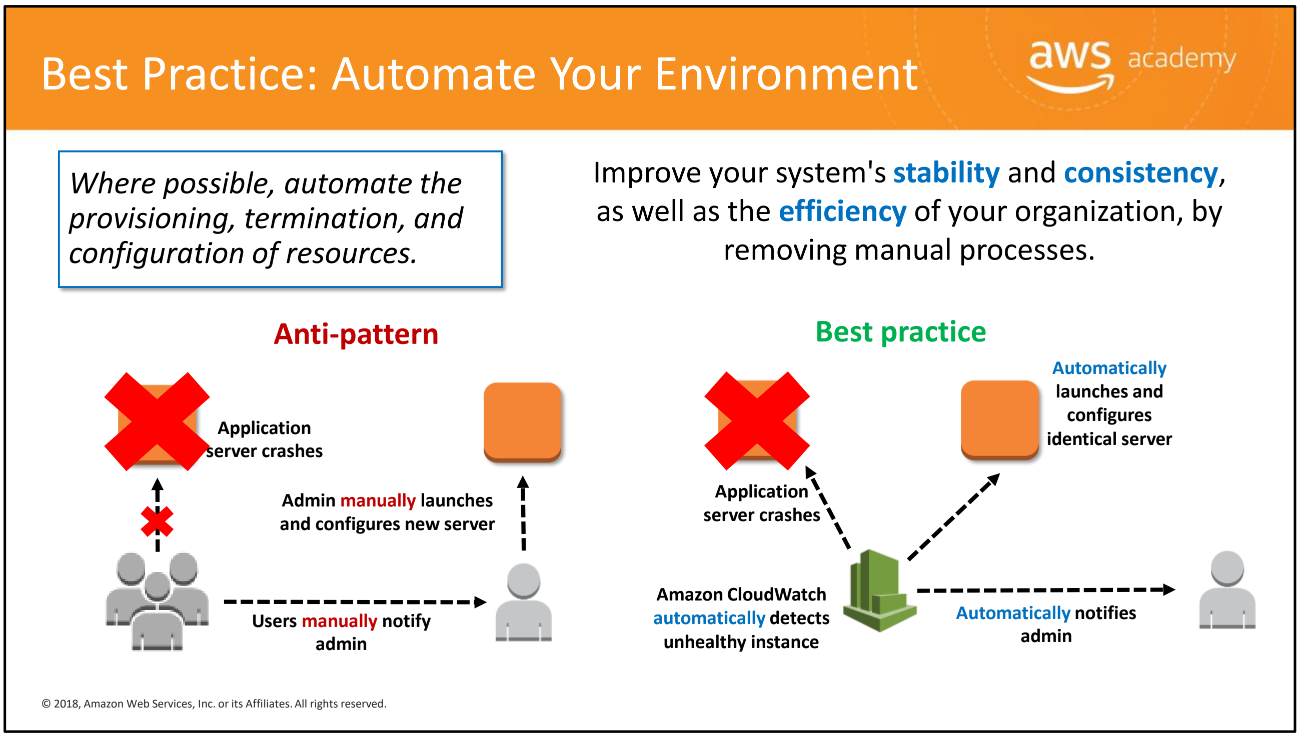
- Creating and configuring AWS services and resources through a management console is a **manual** process.
- What are the challenges and concerns for a manual process?
  - Reliability
  - Reproducibility
    - DEV
    - TEST
    - PROD
- Documentation

Creating and configuring AWS services and resources through a management console is a manual process, and repeating the same steps to configure each environment manually can be prone to error. Many mistakes can be introduced when there is human intervention.

There are issues with reliability and reproducibility as you move from your development environment—or DEV—to the test and production environments, which are known as TEST and PROD. There is always a chance of human error, such as typing a database name incorrectly a database name. Frequently, you need to apply the same configuration that you have on your DEV environment to quality assurance, or QA, environments. There might be multiple QA environments for each stage of testing, such as separate environments for functional testing, user acceptance testing, and stress testing. A QA tester can often discover a defect that’s caused by incorrectly configured resources, which could introduce further delay in the test schedule. Most importantly, you cannot afford to have a configuration error in production servers.

In order to exactly reproduce the same configuration, you might need to document a step-by-step set of instructions for configuration. You must also keep the documentation up-to-date.

The solution to these challenges is to automate these steps by creating a script. The automation script itself can be the documentation. As long as the script is written correctly, it is more reliable than manual configuration. It is also reproducible.



Best Practice: Automate Your Environment

Where possible, automate the provisioning, termination, and configuration of resources.

Improve your system's **stability** and **consistency**, as well as the **efficiency** of your organization, by removing manual processes.

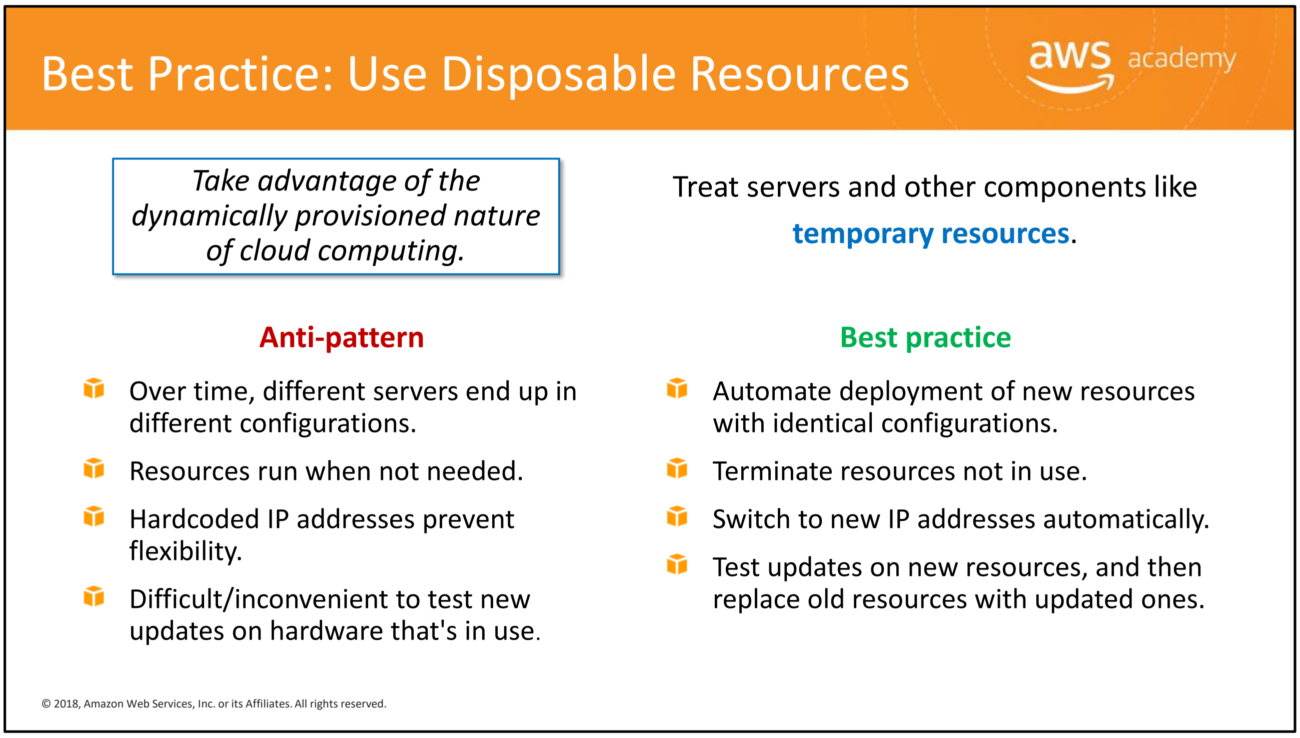
Anti-pattern
Application server crashes
Admin manually launches and configures new server
Users manually notify admin
Best practice
Automatically launches and configures identical server
Application server crashes
Amazon CloudWatch automatically detects unhealthy instance
Automatically notifies admin

Today, the best practice is to automate the environment—including the provisioning, termination, and configuration of resources to ensure that your system is stable, consistent, and efficient.

In the old pattern, the systems administrator is manually notified when the application server crashes, and then he or she manually launches a new server.

Instead of following the anti-pattern, the best practice would be to have Amazon CloudWatch automatically detect a crash. When that crash is detected, the administrator is notified, and new server with the same configuration is launched in parallel. All of these steps happen at the same time, without human intervention.

AWS offers built-in monitoring and automation tools at virtually every layer of your infrastructure. Take advantage of these resources to ensure that your infrastructure can respond quickly to changes in needs. Detecting unhealthy resources and launching replacement resources can be automated, and you can even be notified when resources are changed. You can improve your system’s stability and consistency, as well as the efficiency of the organization, by removing manual processes.



Best Practice: Use Disposable Resources

Take advantage of the dynamically provisioned nature of cloud computing.

Treat servers and other components like **temporary resources**.

Anti-pattern

- Over time, different servers end up in different configurations.
- Resources run when not needed.
- Hardcoded IP addresses prevent flexibility.
- Difficult/inconvenient to test new updates on hardware that's in use.

Best practice

- Automate deployment of new resources with identical configurations.
- Terminate resources not in use.
- Switch to new IP addresses automatically.
- Test updates on new resources, and then replace old resources with updated ones.

Over time, when things are not automated, servers end up with different configurations, different patch levels, or different application patch levels. Resources may be running when they are not needed. Hardcoded IP addresses prevent flexibility. It becomes difficult and inconvenient to test new updates on hardware that is in use. It also makes it difficult to troubleshooting issues.

Treat servers and other components like temporary resources. This best practice returns to the idea of thinking of your infrastructure as software instead of hardware. With hardware, it's easy to "buy in" too much on specific components, which makes it harder to upgrade when necessary because you have too much sunk cost.

On AWS, we recommend that you think of your resources the opposite way: migrating between instances or other discrete resources is fairly simple, which means that you can (and should) treat your resources as easily replaceable. This enables you to move more quickly to respond to changes in capacity needs and upgrade applications and underlying software.

Along with treating the infrastructure as software, the deployment of new resources with identical configurations should be automated, and resources that are not being used should be terminated. Also, switching to new IP address should be done automatically, and updates should be tested on new resources that will replace old resources.

What Does Infrastructure as Code Mean?

Automating your infrastructure:

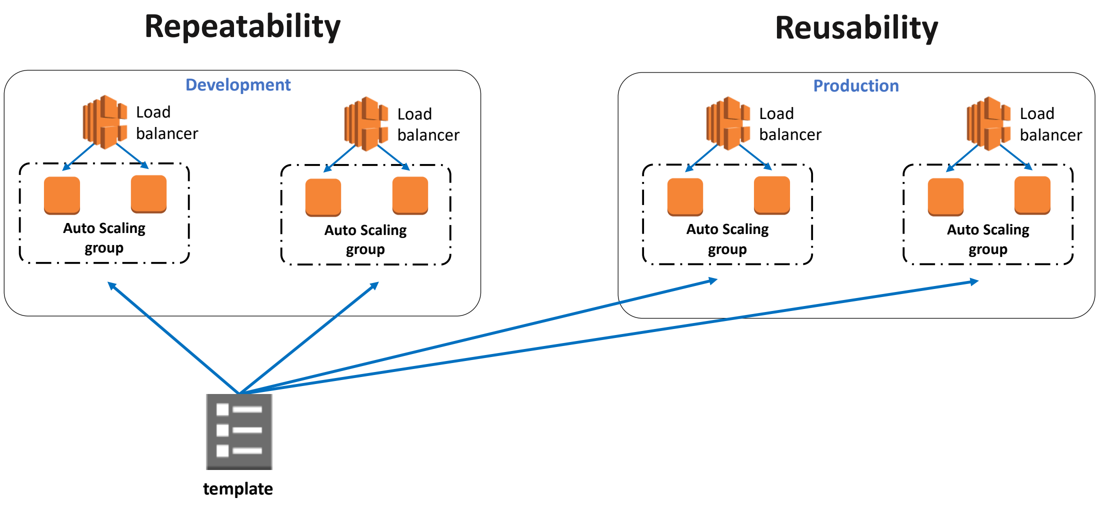
Define your infrastructure as **code**, not as bundles of hardware components.

Process of applying techniques, practices, and tools from **software development** to create **reusable**, **maintainable**, **extensible,** and **testable** infrastructure.

When we talk about infrastructure as code, what do we mean?

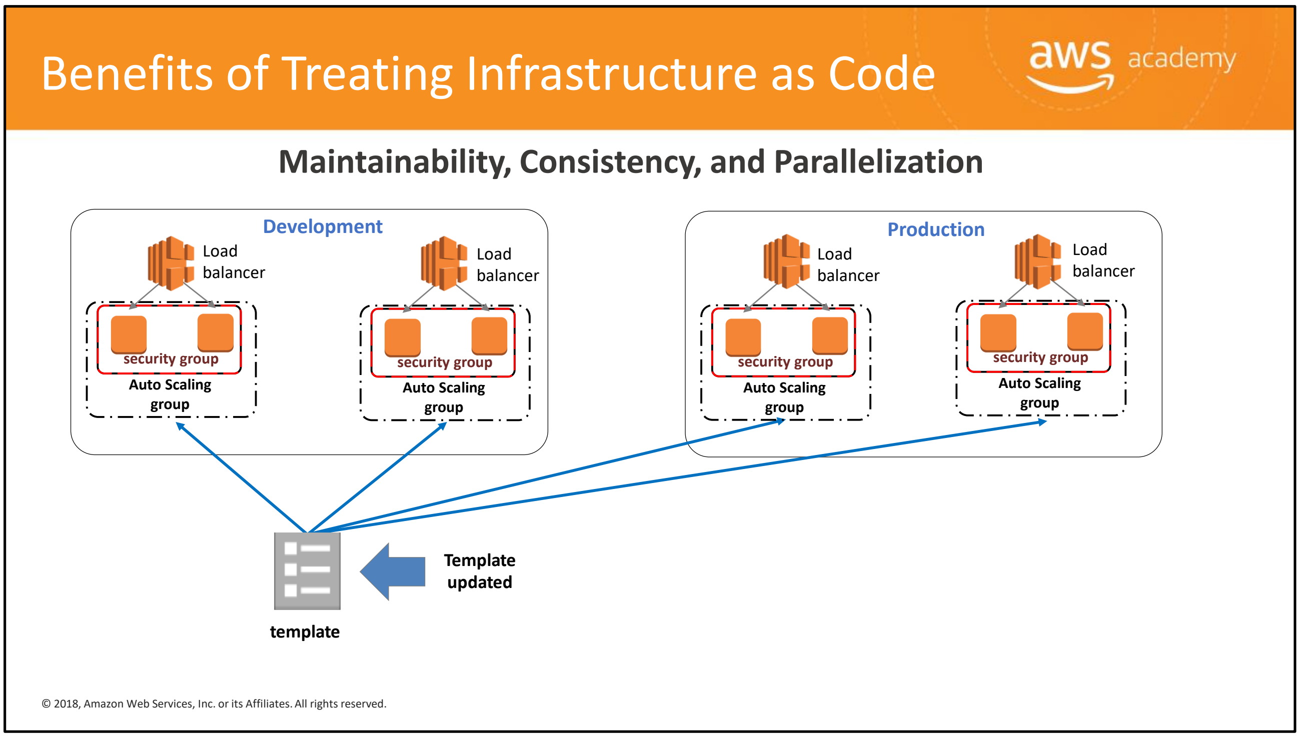
There are many components to automating your infrastructure, but the most critical one is defining your infrastructure as code, not as bundles of hardware components. Thinking of infrastructure as code is the process of applying techniques, practices, and tools from software development to create reusable, maintainable, extensible, and testable infrastructure.

Benefits of Treating Infrastructure as Code



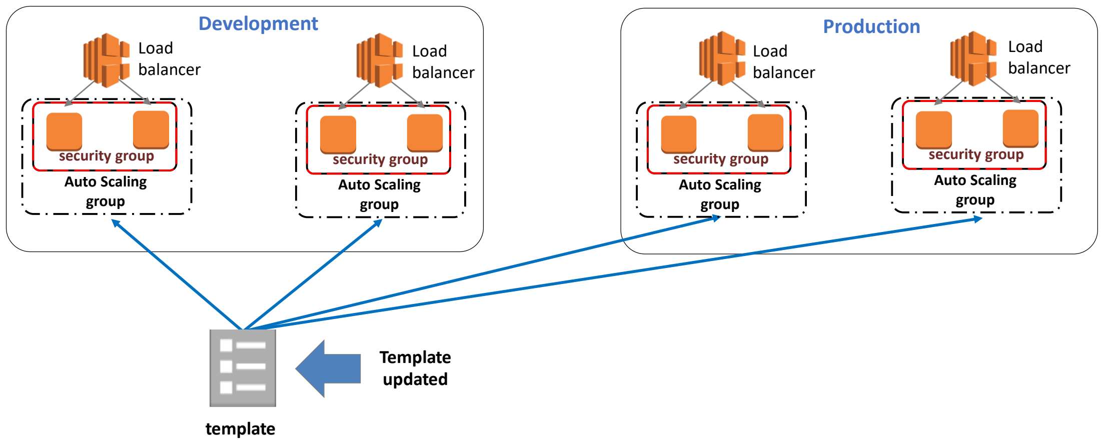
Repeatability is a big advantage when you treat infrastructure as code. For example, you can take an AWS CloudFormation template and roll out the development environment. You can test automatic scaling and the applications. When the test is successful, that exact template can be reused to launch resources that have been thoroughly tested in the development environment into production.

If you build infrastructure with code, you gain the benefits of repeatability and reusability while you build your environments. With one template—or a combination of templates—you can build the same complex environments over and over again. When you do this with AWS, you can even create environments that depend on conditions, so that what you build is specific to the context where you created it. For instance, a template can be designed so that different Amazon Machine Images (AMIs) are used, based on whether this template was launched into the development environment or the production environment.



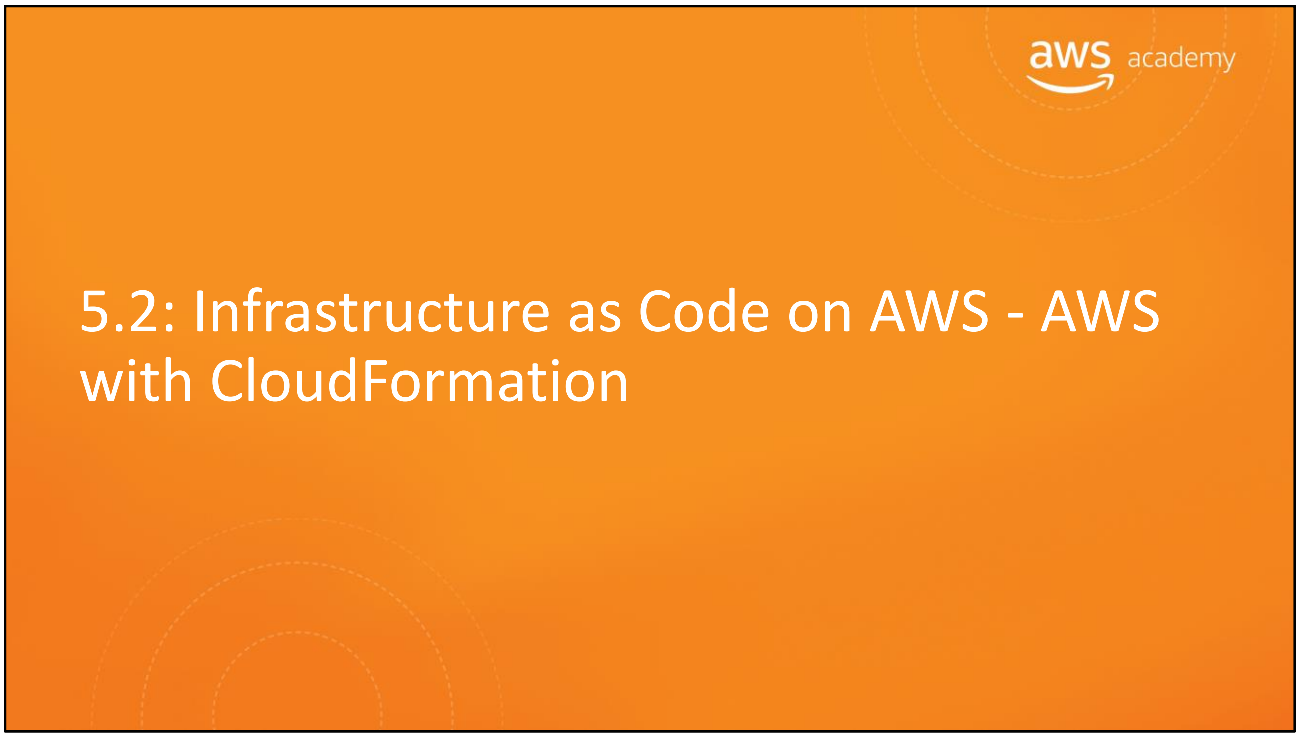
Benefits of Treating Infrastructure as Code

Maintainability, Consistency, and Parallelization



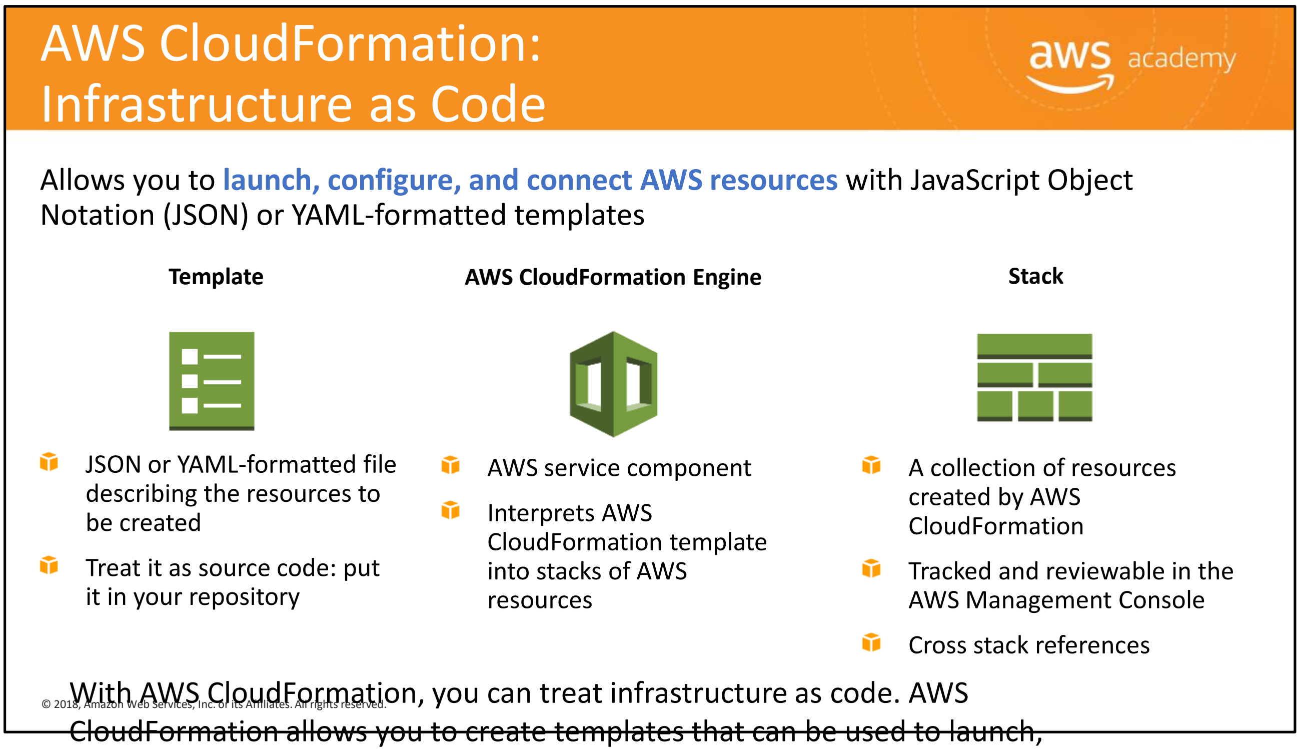
In this scenario, the template was updated to add new security groups to the instance stacks. With one change to the template, all four environments can have the new security group resource added to them.

This feature provides the benefit of more easily maintaining resources, as well as greater consistency, and a reduction in effort through parallelization.



5.2: Infrastructure as Code on AWS - AWS with CloudFormation

Introducing Part 2: Infrastructure as Code on AWS with AWS CloudFormation.



AWS CloudFormation: Infrastructure as Code

Allows you to **launch, configure, and connect AWS resources** with JavaScript Object Notation (JSON) or YAML-formatted templates

Template

- JSON or YAML-formatted file describing the resources to be created
- Treat it as source code: put it in your repository

AWS CloudFormation Engine

- AWS service component
- Interprets AWS CloudFormation template into stacks of AWS resources

Stack

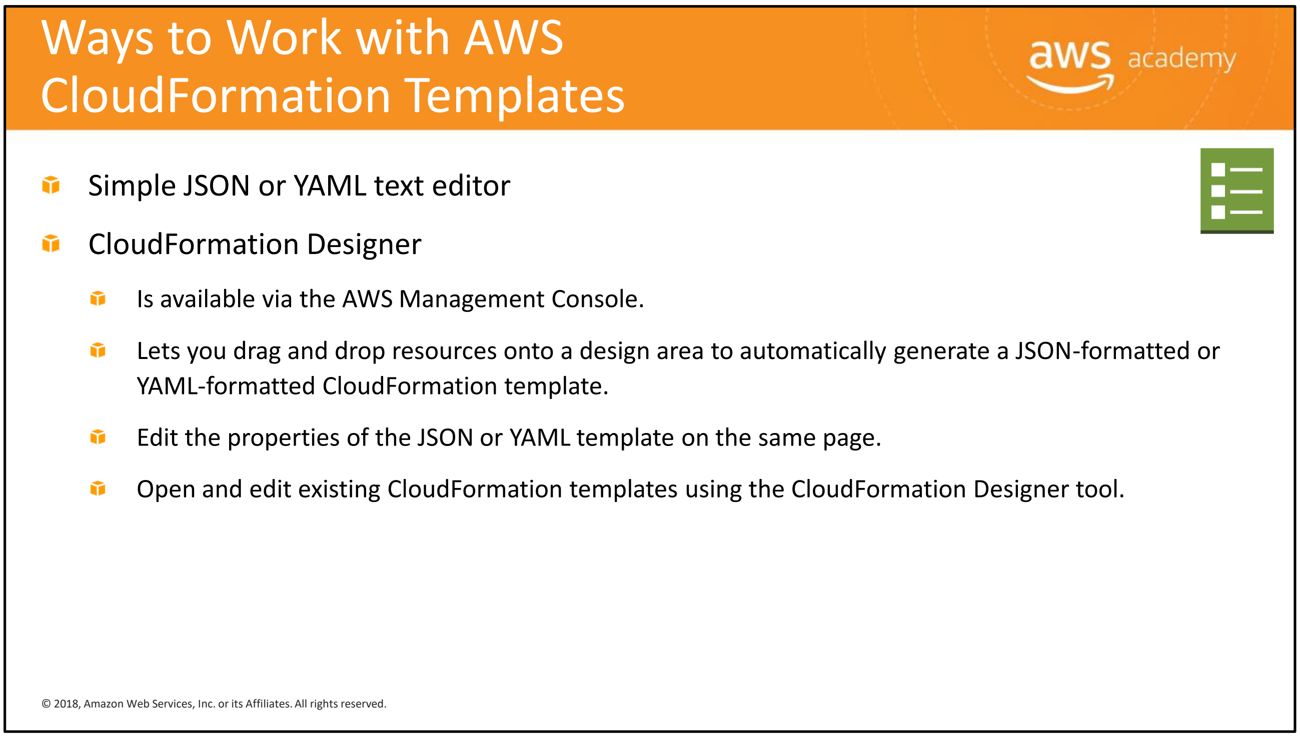
- A collection of resources created by AWS CloudFormation
- Tracked and reviewable in the AWS Management Console
- Cross stack references

With AWS CloudFormation, you can treat infrastructure as code. AWS CloudFormation allows you to create templates that can be used to launch, configure, and connect AWS resources with JavaScript Object Notation (JSON) or YAML-formatted templates.

An AWS CloudFormation template allows you to:
• Treat it as code, and manage it by using your choice of version control, such as Git or Subversion.
• Define an entire application stack—which is all the resources required for your application—in a JSON template file.
• Define runtime parameters for a template, such as Amazon Elastic Compute Cloud—or Amazon EC2— Instance Size, Amazon EC2 Key Pair, etc.

You can now create YAML-formatted templates to describe your AWS resources and properties in AWS CloudFormation. You can use either YAML-formatted templates or JSON-formatted templates to model and describe the resources and properties in your AWS infrastructure. Both YAML-formatted and JSON-formatted AWS CloudFormation templates have the same structure, and they both support all the same features.
You can now also create cross stack references that let you share outputs from one stack with another stack. This feature lets you share things like AWS Identity and Access Management—or IAM—roles, virtual private cloud—or VPC—information, and security groups. Previously, you needed to use AWS CloudFormation custom resources to accomplish these tasks. Now, you can export values from one stack and import them to another stack by using the new ImportValue intrinsic function.

Cross-stack references are useful for customers who separate their AWS infrastructure into logical components that grouped by stack—such as a network stack, an application stack, etc.—and who need a way to loosely couple stacks together as an alternative to nested stacks.



Ways to Work with AWS CloudFormation Templates

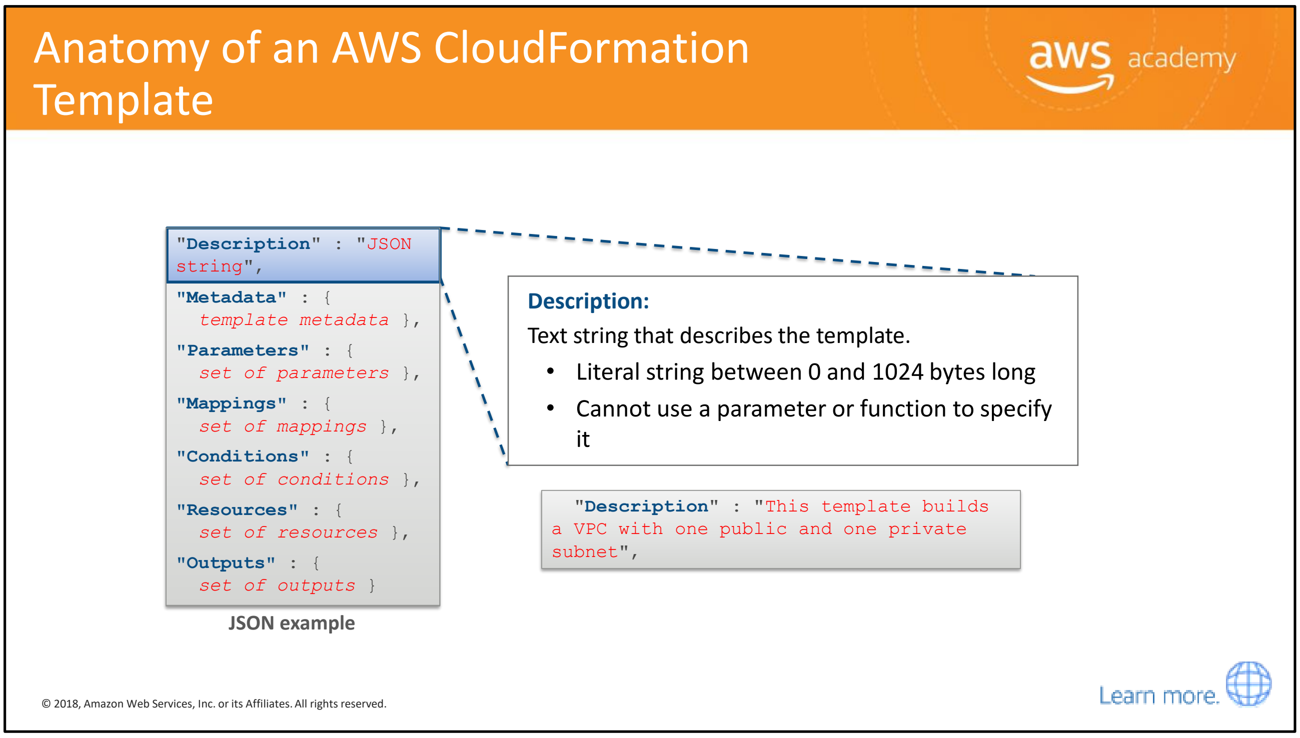
- Simple JSON or YAML text editor
- CloudFormation Designer
  - Is available via the AWS Management Console.
  - Lets you drag and drop resources onto a design area to automatically generate a JSON-formatted or YAML-formatted CloudFormation template.
  - Edit the properties of the JSON or YAML template on the same page.
  - Open and edit existing CloudFormation templates using the CloudFormation Designer tool.

AWS CloudFormation templates can be created several different ways. The most traditional is using a code editor that supports JSON syntax, such as Atom or Sublime Text. However, you can also build templates visually using our own CloudFormation Designer tool, available in the AWS Management Console, or with a third party WYSIWYG editor.

The AWS CloudFormation Designer lets you drag and drop resources onto a design area to automatically generate a JSON-formatted or YAML-formatted CloudFormation template. The properties of the JSON or YAML template can be edited on the same page. Existing CloudFormation templates can be opened and edited using the CloudFormation Designer tool.

Mod 5.3: Cloud Formation Templates

Introducing Part 3: How Should Resources Be Grouped Together into Templates?



Anatomy of an AWS CloudFormation Template

```
"Description" : "JSON
string",
"Metadata" : {
  template metadata },
"Parameters" : {
  set of parameters },
"Mappings" : {
  set of mappings },
"Conditions" : {
  set of conditions },
"Resources" : {
  set of resources },
"Outputs" : {
  set of outputs }
```

JSON example

Description:

Text string that describes the template.

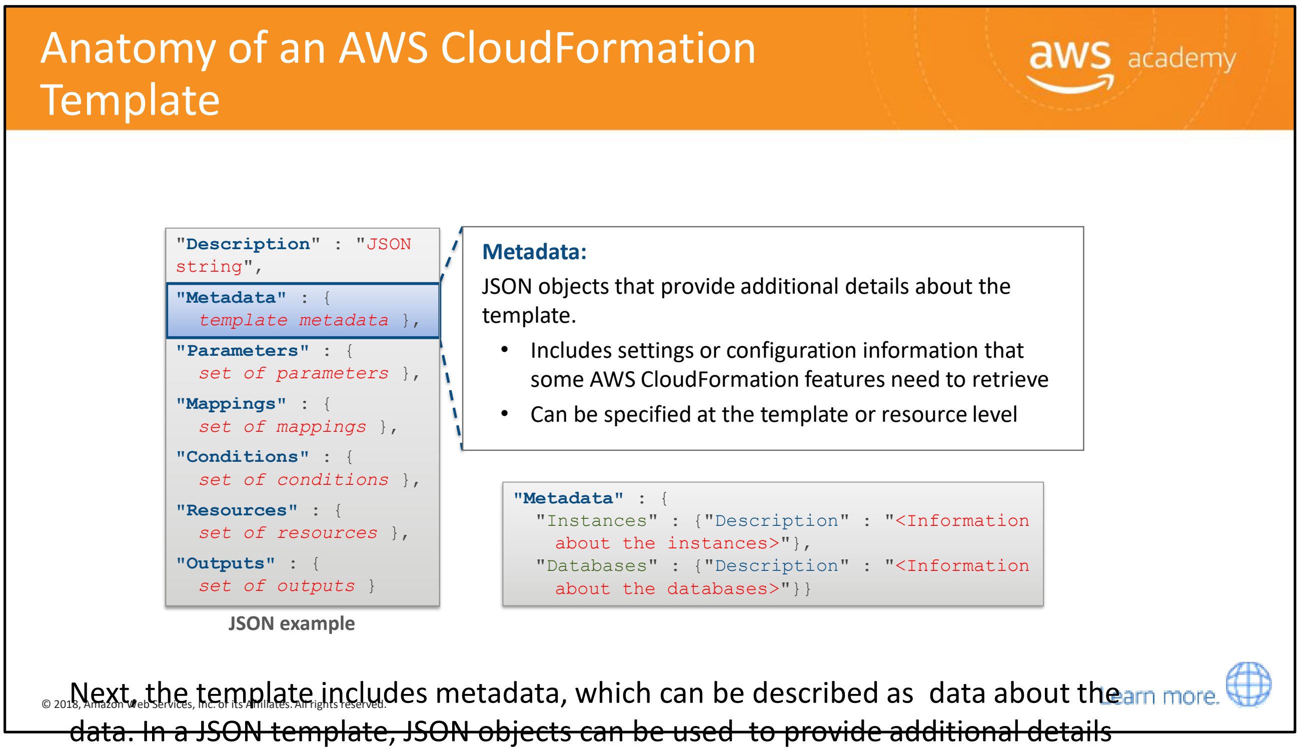
- Literal string between 0 and 1024 bytes long
- Cannot use a parameter or function to specify it

```
  "Description" : "This template builds
a VPC with one public and one private
subnet",
```

Learn more.

This slide shows the anatomy of an AWS CloudFormation template. The top has a description, which is a text string that describes the template.

For more information, go to the AWS CloudFormation documentation about the template Description section.
http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/template-description-structure.html



Anatomy of an AWS CloudFormation Template

```
"Description" : "JSON
string",
"Metadata" : {
   template metadata },
"Parameters" : {
   set of parameters },
"Mappings" : {
   set of mappings },
"Conditions" : {
   set of conditions },
"Resources" : {
   set of resources },
"Outputs" : {
   set of outputs }
```

JSON example

Metadata:

JSON objects that provide additional details about the template.

- Includes settings or configuration information that some AWS CloudFormation features need to retrieve
- Can be specified at the template or resource level

```
"Metadata" : {
  "Instances" : {"Description" : "<Information
    about the instances>"},
  "Databases" : {"Description" : "<Information
    about the databases>"}}
```

Learn more.

Next, the template includes metadata, which can be described as data about the data. In a JSON template, JSON objects can be used to provide additional details about the template.

Some AWS CloudFormation features retrieve settings or configuration information that you define from the Metadata section. You define this information in the following AWS CloudFormation-specific metadata keys:

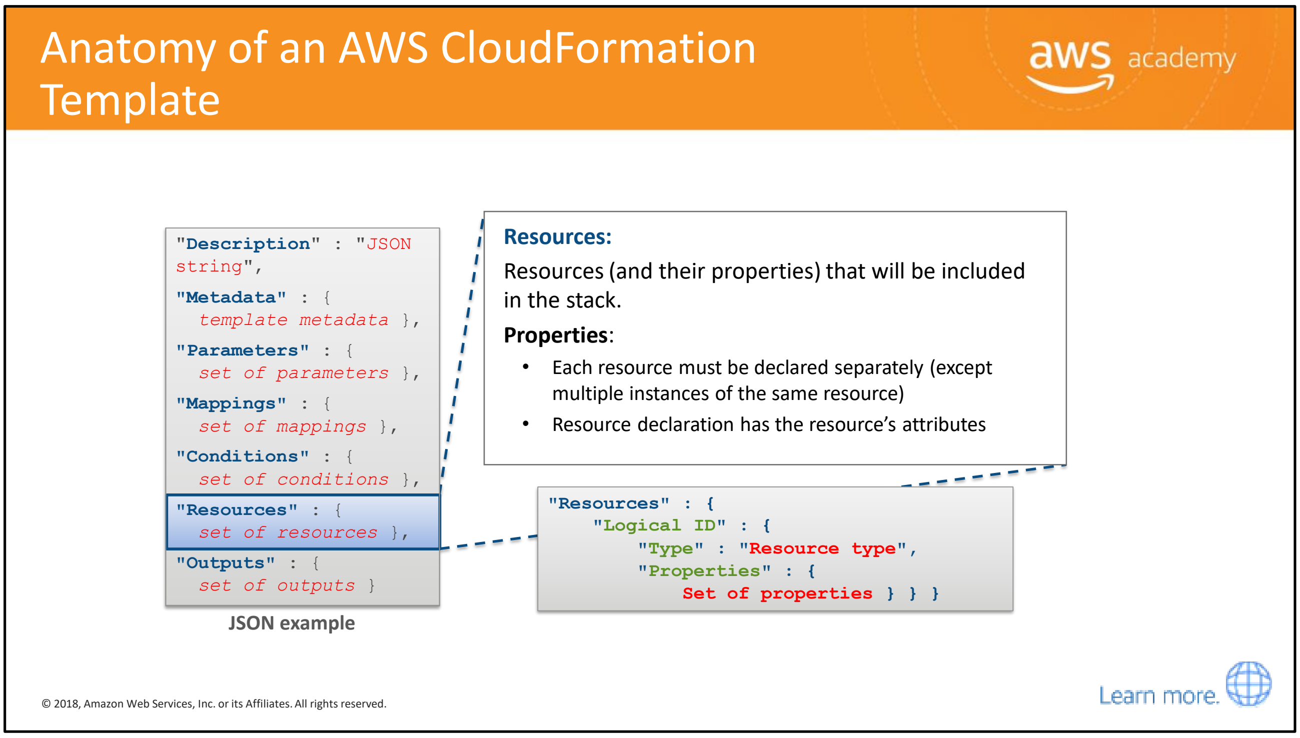
AWS::CloudFormation::Init
Defines configuration tasks for the cfn-init helper script. This script is useful for configuring and installing applications on Amazon EC2 instances. For more information, see AWS::CloudFormation::Init.

AWS::CloudFormation::Interface
Defines the grouping and ordering of input parameters when they are displayed in the AWS CloudFormation console. By default, the AWS CloudFormation console alphabetically sorts parameters by their logical ID. For more information, see AWS::CloudFormation::Interface.

AWS::CloudFormation::Designer
Describes how your resources are laid out in AWS CloudFormation Designer. Designer automatically adds this information when you use it create and update templates. For more information, go to the documentation page that explain AWS CloudFormation Designer:
https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/working-with-

templates-cfn-designer.html

To learn more about template metadata, go to the AWS CloudFormation documentation about the Metadata section.
http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/metadata-section-structure.html



Anatomy of an AWS CloudFormation Template

```
"Description" : "JSON
string",
"Metadata" : {
  template metadata },
"Parameters" : {
  set of parameters },
"Mappings" : {
  set of mappings },
"Conditions" : {
  set of conditions },
"Resources" : {
  set of resources },
"Outputs" : {
  set of outputs }
```

JSON example

Resources:

Resources (and their properties) that will be included in the stack.

Properties:

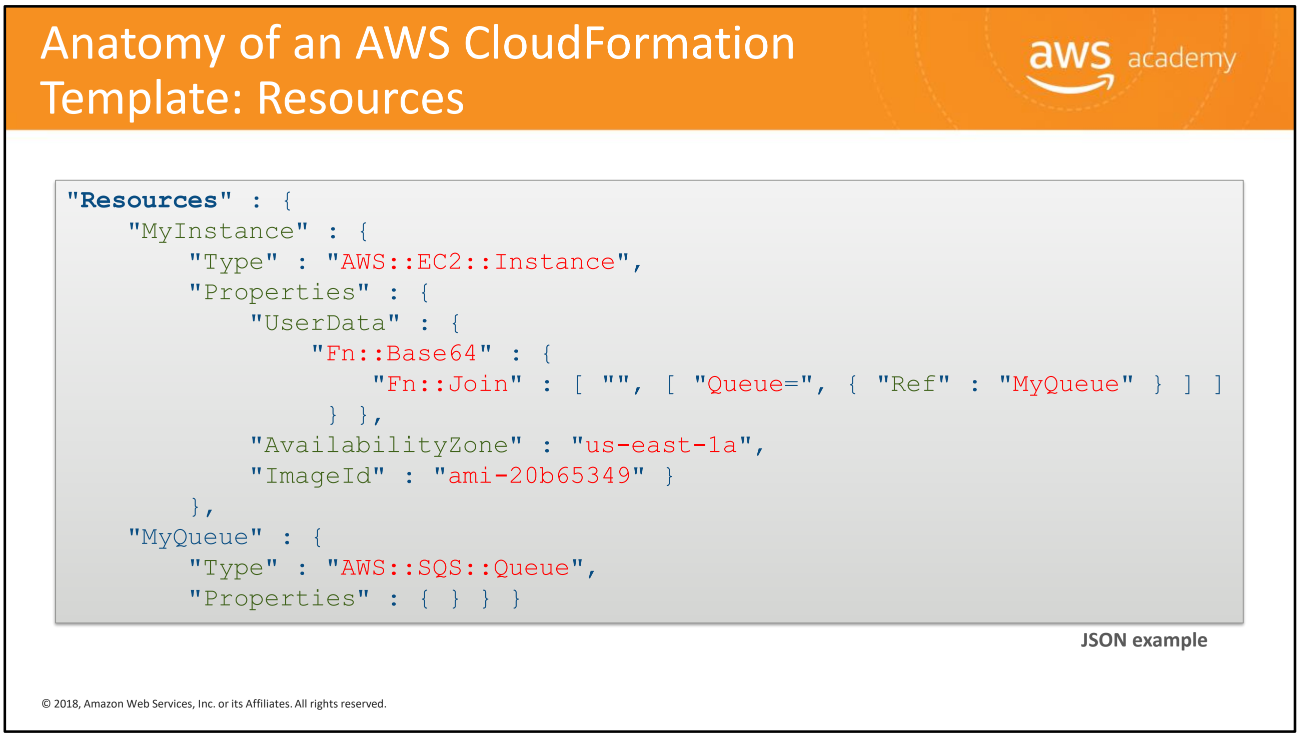
- Each resource must be declared separately (except multiple instances of the same resource)
- Resource declaration has the resource's attributes

```
"Resources" : {
    "Logical ID" : {
        "Type" : "Resource type",
        "Properties" : {
            Set of properties } } }
```

Learn more.

The Resources section is required, and it declares the AWS resources that will be included or created in the stack, such as an Amazon EC2 instance or an Amazon Simple Storage Service –or Amazon S3—bucket. You must declare each resource separately; however, you can specify multiple resources of the same type. If you declare multiple resources, separate them with commas.

For more information, go to the AWS CloudFormation documentation on how to create and use resources. http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/resources-section-structure.html



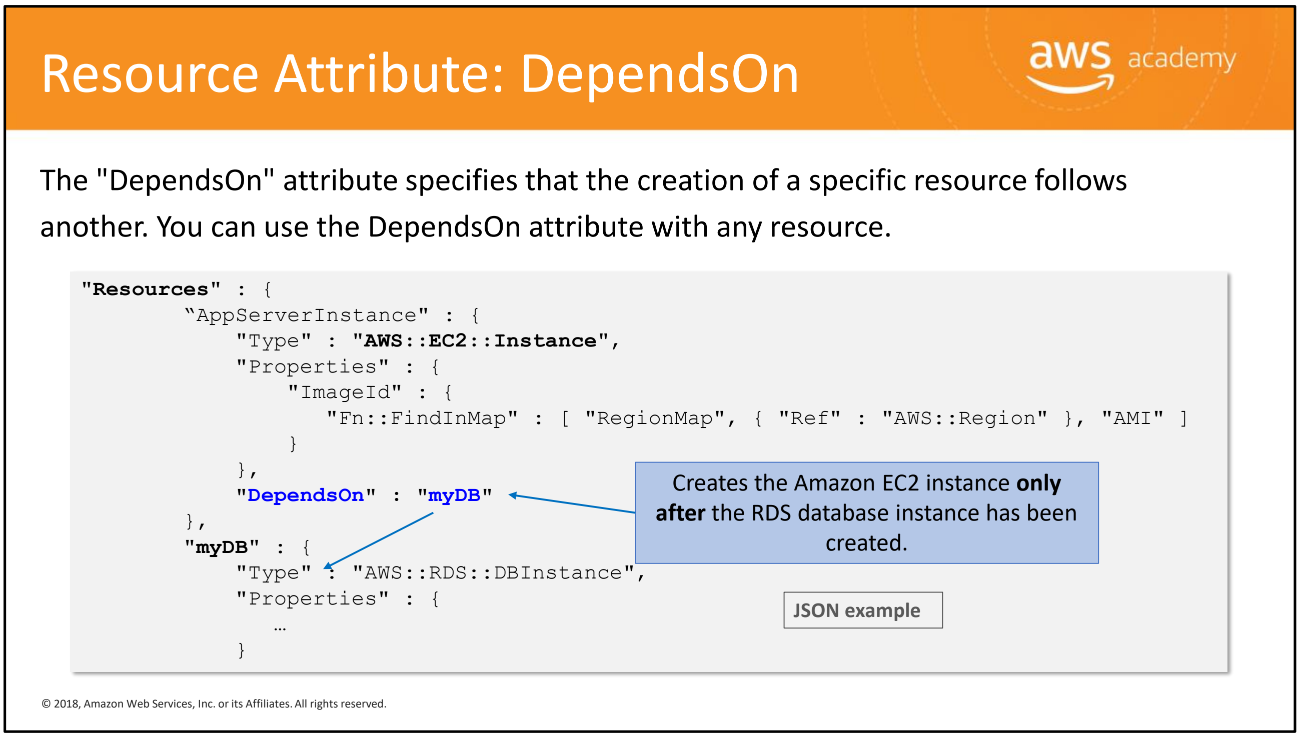
Anatomy of an AWS CloudFormation Template: Resources

```
"Resources" : {
    "MyInstance" : {
        "Type" : "AWS::EC2::Instance",
        "Properties" : {
            "UserData" : {
                "Fn::Base64" : {
                    "Fn::Join" : [ "", [ "Queue=", { "Ref" : "MyQueue" } ] ]
                 } },
            "AvailabilityZone" : "us-east-1a",
            "ImageId" : "ami-20b65349" }
        },
    "MyQueue" : {
        "Type" : "AWS::SQS::Queue",
        "Properties" : { } } }
```

JSON example

In this example, the MyInstance resource includes the MyQueue resource as part of its UserData property, as well as specifications for its AMI ImageId and Availability Zone properties. These properties could also be set in the Parameters or Conditions sections.

This example shows a resource declaration. It defines two resources. The first resource is an Amazon EC2 instance, or "MyInstance". The second resource is an Amazon Simple Queue Service—or Amazon SQS—queue that is called "MyQueue". The AvailabilityZone setting indicates that the Amazon EC2 instance will be hosted in Northern Virginia—or us-east-1a—. The ImageId setting defines the particular AMI that will be used for this Amazon EC2 instance.



Resource Attribute: DependsOn

The "DependsOn" attribute specifies that the creation of a specific resource follows another. You can use the DependsOn attribute with any resource.

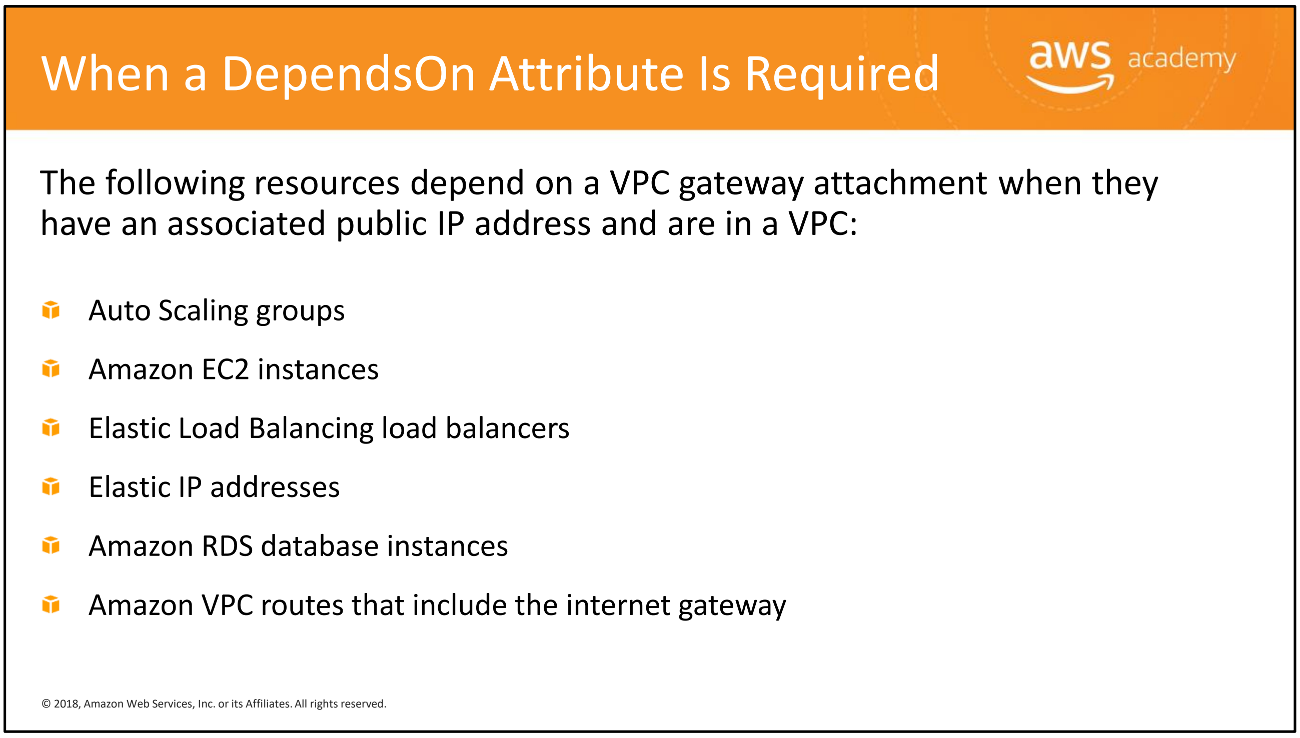
```
"Resources" : {
        “AppServerInstance" : {
            "Type" : "AWS::EC2::Instance",
            "Properties" : {
                "ImageId" : {
                   "Fn::FindInMap" : [ "RegionMap", { "Ref" : "AWS::Region" }, "AMI" ]
                }
            },
            "DependsOn" : "myDB"
        },
        "myDB" : {
            "Type" : "AWS::RDS::DBInstance",
            "Properties" : {
               …
            }
```

Creates the Amazon EC2 instance **only after** the RDS database instance has been created.

JSON example

The DependsOn attribute is an important attribute. DependsOn is how you specify that AWS CloudFormation should wait to launch a resource until a specific, different resource has already finished being created.

In this case, there is an Amazon EC2 instance that can only be created after the database has been established. So, the creation of the Amazon EC2 instance depends on when the database is created.

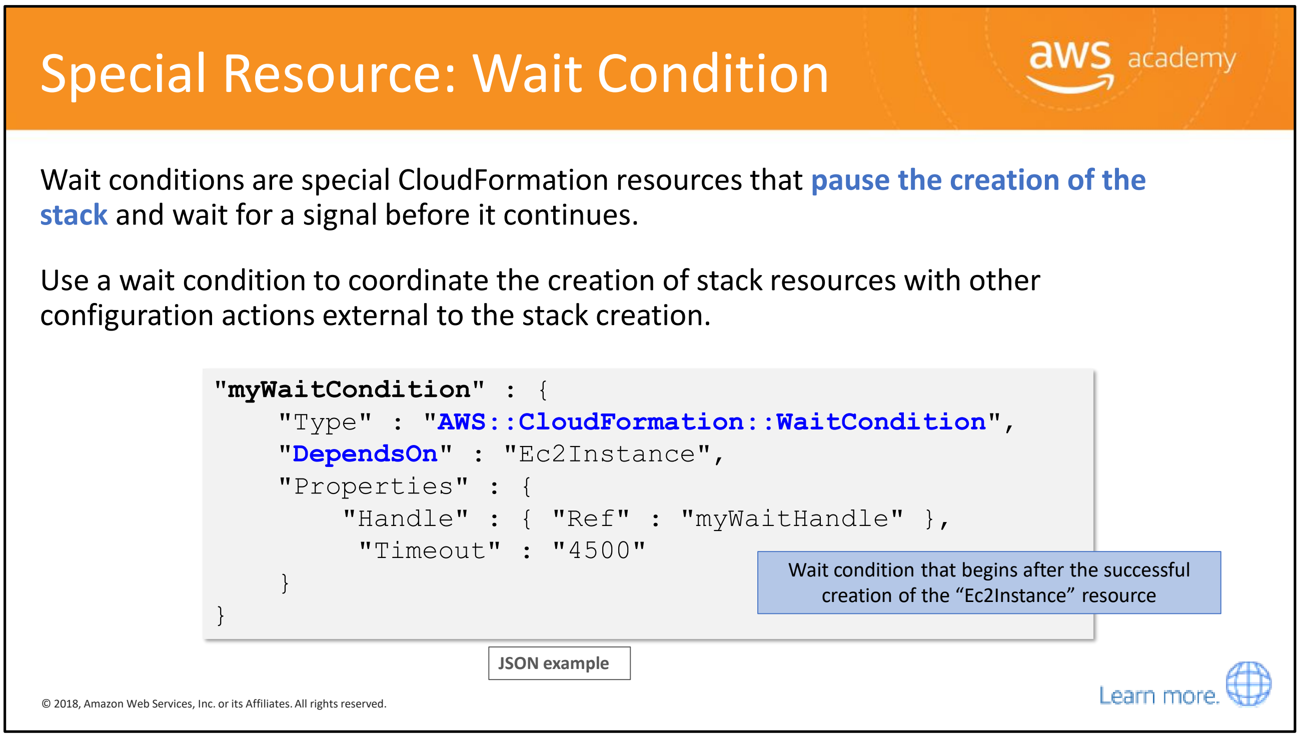


When a DependsOn Attribute Is Required

The following resources depend on a VPC gateway attachment when they have an associated public IP address and are in a VPC:

- Auto Scaling groups
- Amazon EC2 instances
- Elastic Load Balancing load balancers
- Elastic IP addresses
- Amazon RDS database instances
- Amazon VPC routes that include the internet gateway

The DependsOn attribute should be used when you need to wait for something. Some resources in a VPC require a gateway—either an internet gateway or a VPN gateway. If your AWS CloudFormation template defines a VPC, a gateway, and a gateway attachment, any resources that require the gateway depend on the gateway attachment. For example, an Amazon EC2 instance with a public IP address depends on the VPC gateway attachment if the VPC and internet gateway resources are also declared in the same template. Other VPC-dependent resources include Auto Scaling groups, Amazon EC2 instances, Elastic Load Balancing load balancers, Elastic IP addresses, Amazon Relational Database Service –or Amazon RDS—database instances, and Amazon Virtual Private Cloud—or Amazon VPC—routes that include the internet gateway.



Special Resource: Wait Condition

Wait conditions are special CloudFormation resources that **pause the creation of the stack** and wait for a signal before it continues.

Use a wait condition to coordinate the creation of stack resources with other configuration actions external to the stack creation.

```
"myWaitCondition" : {
    "Type" : "AWS::CloudFormation::WaitCondition",
    "DependsOn" : "Ec2Instance",
    "Properties" : {
        "Handle" : { "Ref" : "myWaitHandle" },
         "Timeout" : "4500"
    }
}
```

Wait condition that begins after the successful creation of the "Ec2Instance" resource

JSON example

Learn more.

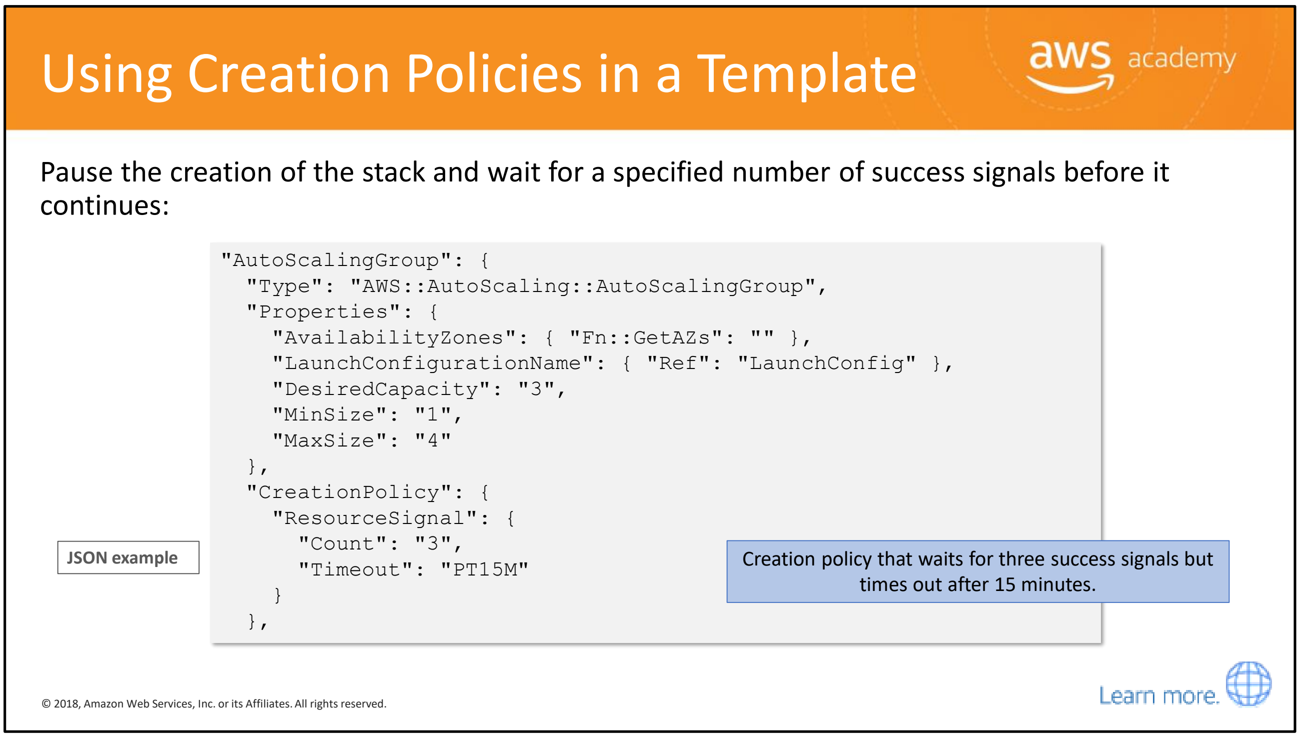
Another special condition that can be used to wait or pause and receive a signal to continue is the wait condition.

The AWS::CloudFormation::WaitConditionHandle type has no properties. When you reference the WaitConditionHandle resource by using the Ref function, AWS CloudFormation returns a pre-signed URL. You pass this URL to applications or scripts that are running on your Amazon EC2 instances to send signals to that URL. An associated AWS::CloudFormation::WaitCondition resource checks the URL for the required number of success signals or for a failure signal.

The timeout value is in seconds—for example, 4,500 seconds.

This example shows an AWS CloudFormation template with a wait condition that is tied to an Amazon EC2 instance. It will wait for that Amazon EC2 instance or it will time out after 4,500 seconds.

For more information, see the AWS CloudFormation documentation on wait conditions in templates.
http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/using-cfn-waitcondition.html



Using Creation Policies in a Template

aws academy

Pause the creation of the stack and wait for a specified number of success signals before it continues:

JSON example

```
"AutoScalingGroup": {
  "Type": "AWS::AutoScaling::AutoScalingGroup",
  "Properties": {
    "AvailabilityZones": { "Fn::GetAZs": "" },
    "LaunchConfigurationName": { "Ref": "LaunchConfig" },
    "DesiredCapacity": "3",
    "MinSize": "1",
    "MaxSize": "4"
  },
  "CreationPolicy": {
    "ResourceSignal": {
      "Count": "3",
      "Timeout": "PT15M"
    }
  },
```

Creation policy that waits for three success signals but times out after 15 minutes.

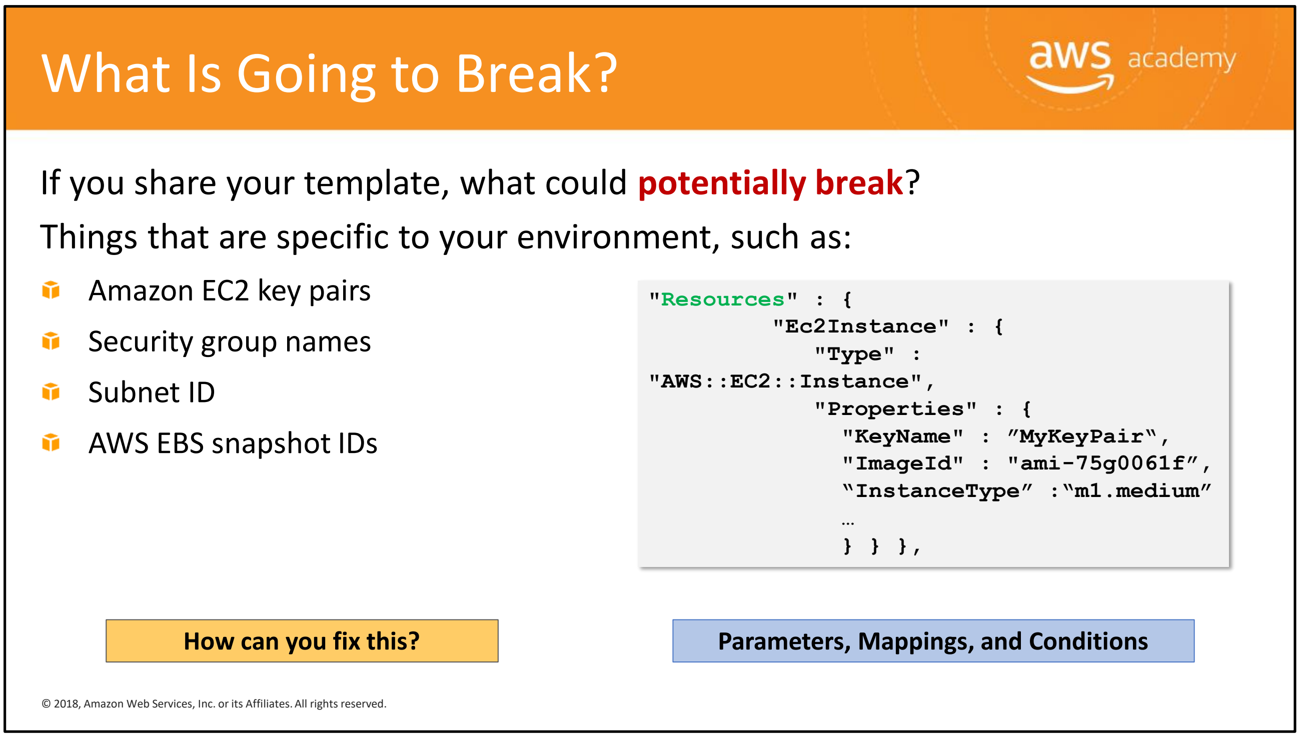
Learn more.

By using creation policies in a template, you can pause stack creation and wait for a specified number of successful signals.

This creation policy is associated with the creation of an Auto Scaling group. The default count is 1 and the default timeout period is 5 minutes, or PT5M. The value for the count must be an integer, and the value for the timeout must be a string that is in ISO8601 duration format, which has the form “PT#H#M#S” where # is the number of hours, minutes, and seconds, respectively. With this policy, three successful signals within fifteen minutes are required or it will time out.

Set your timeouts so that they give your resources enough time to get up and running. When the timeout period expires, or a failure signal is received, the creation of the resource fails, and AWS CloudFormation rolls the stack back.

For more information, go to the AWS CloudFormation documentation about the CreationPolicy attribute.
https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-attribute-creationpolicy.html



What Is Going to Break?

aws academy

If you share your template, what could **potentially break**?

Things that are specific to your environment, such as:

- Amazon EC2 key pairs
- Security group names
- Subnet ID
- AWS EBS snapshot IDs

```
"Resources" : {
          "Ec2Instance" : {
             "Type" :
"AWS::EC2::Instance",
             "Properties" : {
               "KeyName" : "MyKeyPair",
               "ImageId" : "ami-75g0061f",
               "InstanceType" :"m1.medium"
               …
               } } },
```

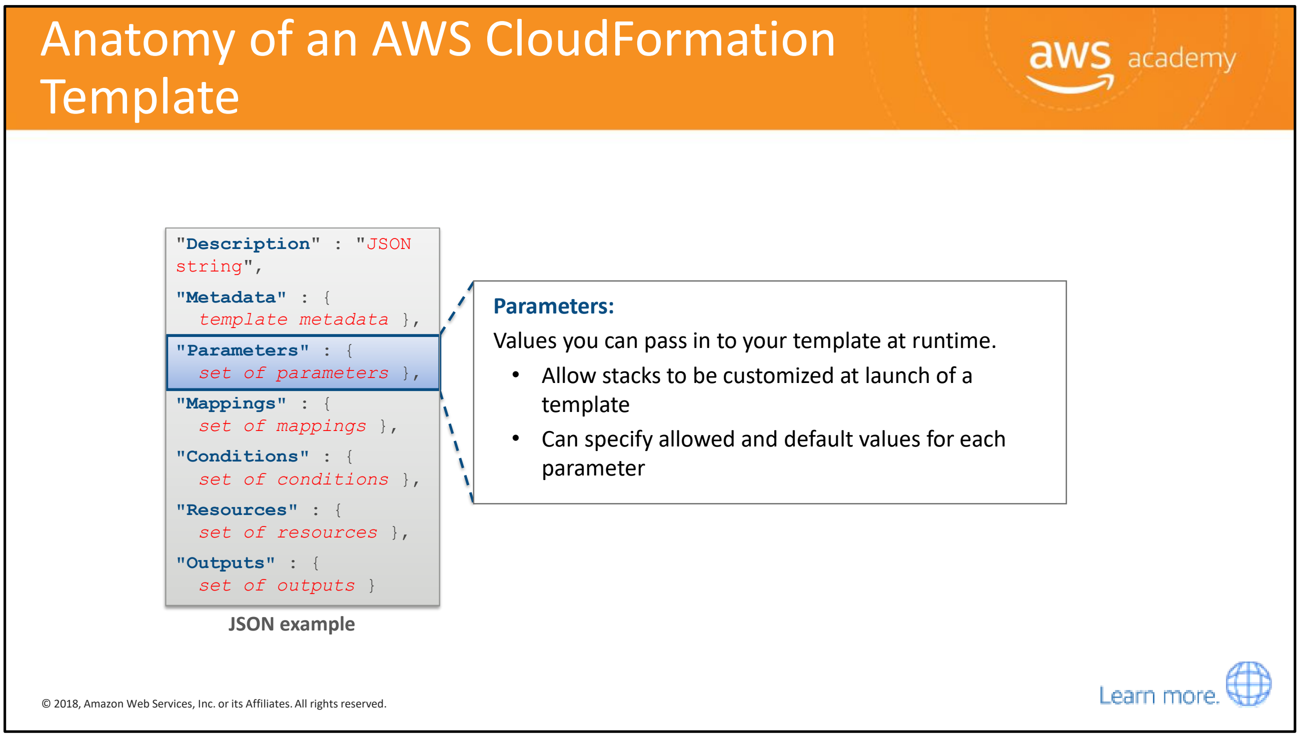
How can you fix this?

Parameters, Mappings, and Conditions

If you share your template, what could potentially break if you use a template from one Region in a different Region?

The things that could break are things that are specific to your environment, such as Amazon EC2 key pairs, security group names, subnet IDs, and Amazon Elastic Block Store—or Amazon EBS—snapshot IDs.

How can you fix this situation? It can be fixed by using parameters, mappings, and conditions.



Anatomy of an AWS CloudFormation Template

```
"Description" : "JSON
string",
"Metadata" : {
  template metadata },
"Parameters" : {
  set of parameters },
"Mappings" : {
  set of mappings },
"Conditions" : {
  set of conditions },
"Resources" : {
  set of resources },
"Outputs" : {
  set of outputs }
```

JSON example

Parameters:

Values you can pass in to your template at runtime.

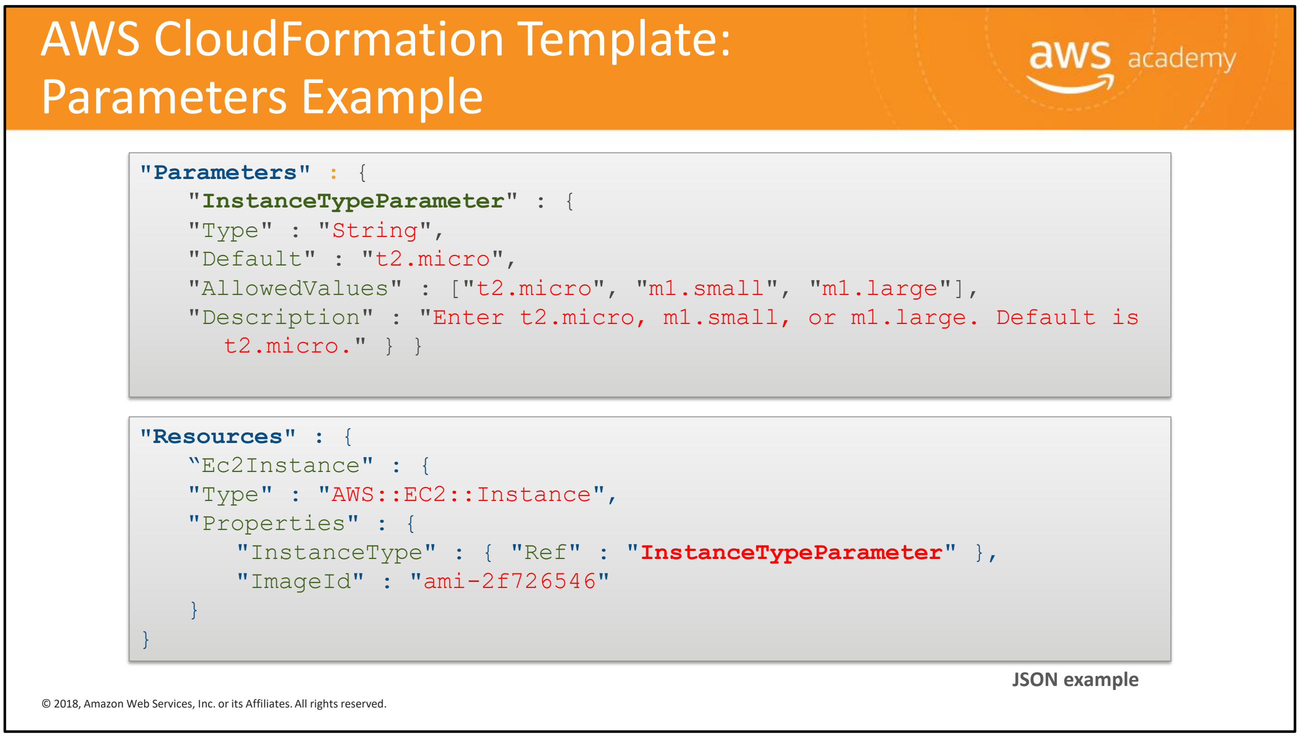
- Allow stacks to be customized at launch of a template
- Can specify allowed and default values for each parameter

Learn more.

In this example, a parameter allows you to pass the value of your template at runtime, which allows you to customize things in the stack. You can specify allowed and default values for each parameter.

You declare parameters in a template's Parameters object. A parameter contains a list of attributes that define its value, and also define constraints against its value. The only required attribute is Type, which can be String, Number, or CommaDelimitedList. You can also add a Description attribute that tells a user more about what kind of value they should specify. The parameter's name and description appear in the Specify Parameters page when a user uses the template in the Create Stack wizard.

For more information, go to the AWS CloudFormation documentation about creating and using parameters.
https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/parameters-section-structure.html



AWS CloudFormation Template: Parameters Example

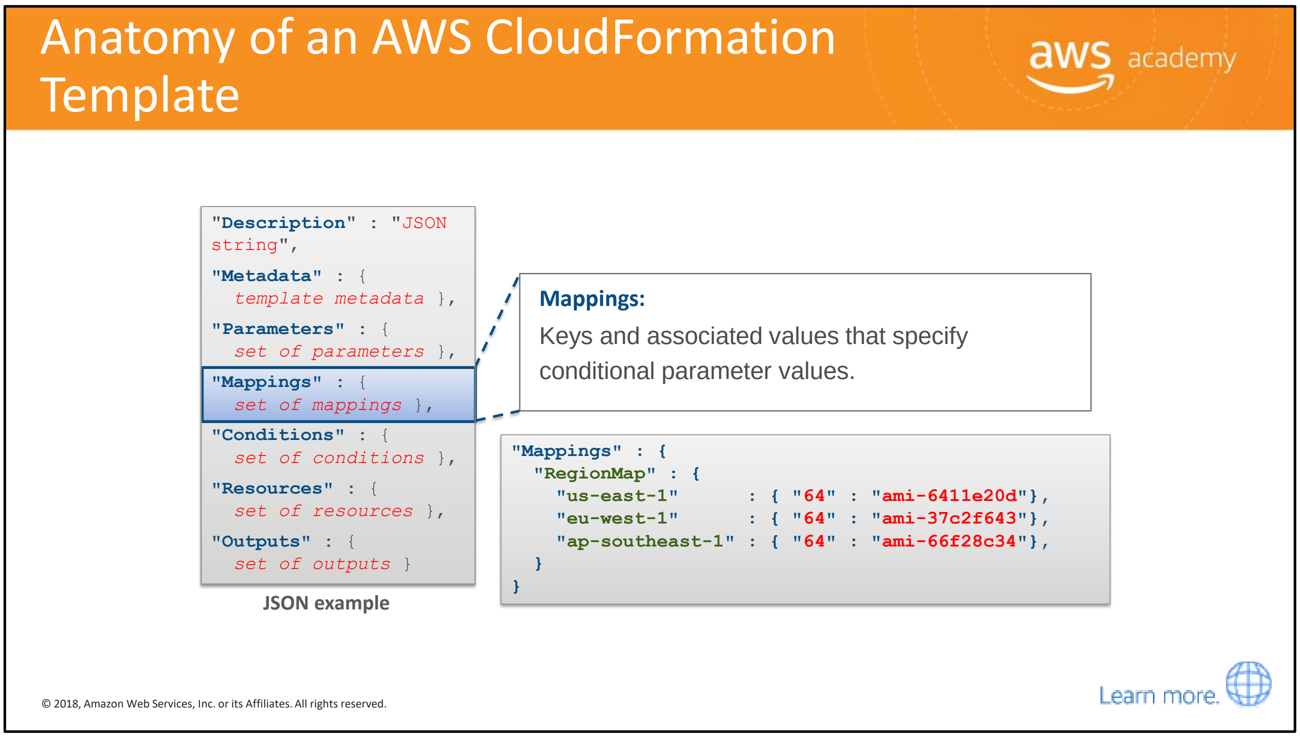
```
"Parameters" : {
    "InstanceTypeParameter" : {
    "Type" : "String",
    "Default" : "t2.micro",
    "AllowedValues" : ["t2.micro", "m1.small", "m1.large"],
    "Description" : "Enter t2.micro, m1.small, or m1.large. Default is
       t2.micro." } }
```

```
"Resources" : {
    “Ec2Instance" : {
    "Type" : "AWS::EC2::Instance",
    "Properties" : {
        "InstanceType" : { "Ref" : "InstanceTypeParameter" },
        "ImageId" : "ami-2f726546"
    }
}
```

JSON example

In this example, the InstanceTypeParameter specifies a default Amazon EC2 instance type of t2.micro, but users can choose from a t2.micro, m1.small, or m1.large instance type when they invoke the template. It also provides a description, which appears in the AWS CloudFormation Console when the template is launched.

Then, when an Amazon EC2 instance is launched in the Resources section of the template, the Properties section of the instance can reference the InstanceTypeParameter specification. In this example, the “Ec2Instance" resource—which is an Amazon EC2 instance—references the InstanceTypeParameter specification for its instance type. You could also specify details like the range of acceptable AMI ImageId numbers, key pairs, subnets, or any properties that must be specified for a resource.



Anatomy of an AWS CloudFormation Template

```
"Description" : "JSON
string",
"Metadata" : {
  template metadata },
"Parameters" : {
  set of parameters },
"Mappings" : {
  set of mappings },
"Conditions" : {
  set of conditions },
"Resources" : {
  set of resources },
"Outputs" : {
  set of outputs }
```

JSON example

Mappings:

Keys and associated values that specify conditional parameter values.

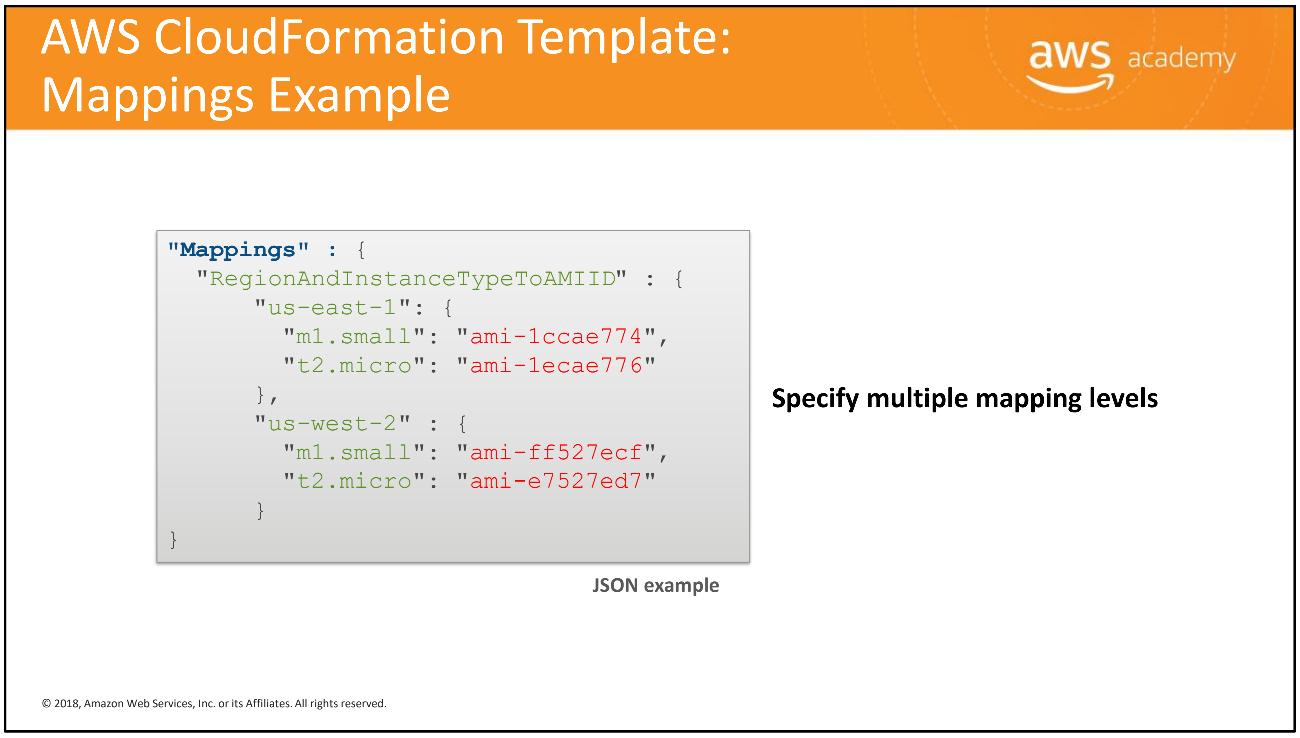
```
"Mappings" : {
  "RegionMap" : {
    "us-east-1"      : { "64" : "ami-6411e20d"},
    "eu-west-1"      : { "64" : "ami-37c2f643"},
    "ap-southeast-1" : { "64" : "ami-66f28c34"},
  }
}
```

Learn more.

Mappings are keys and their associated values, and they specify conditional parameter values. Mappings allow you to customize the properties of a resource based on certain conditions, which enables you to have fine-grained control over how your templates are launched. For example, an AMI ImageId number is unique to a Region, and the person who received your template might not necessarily know which AMI to use. You can thus provide the AMI lookup list using the Mappings parameter.

This example contains a map for Regions. The mapping lists the AMI that should be used, based on the Region that the EC2 instance will launch in.

For more information, go to the AWS CloudFormation documentation on how to create and use Mappings.
https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/mappings-section-structure.html



AWS CloudFormation Template: Mappings Example

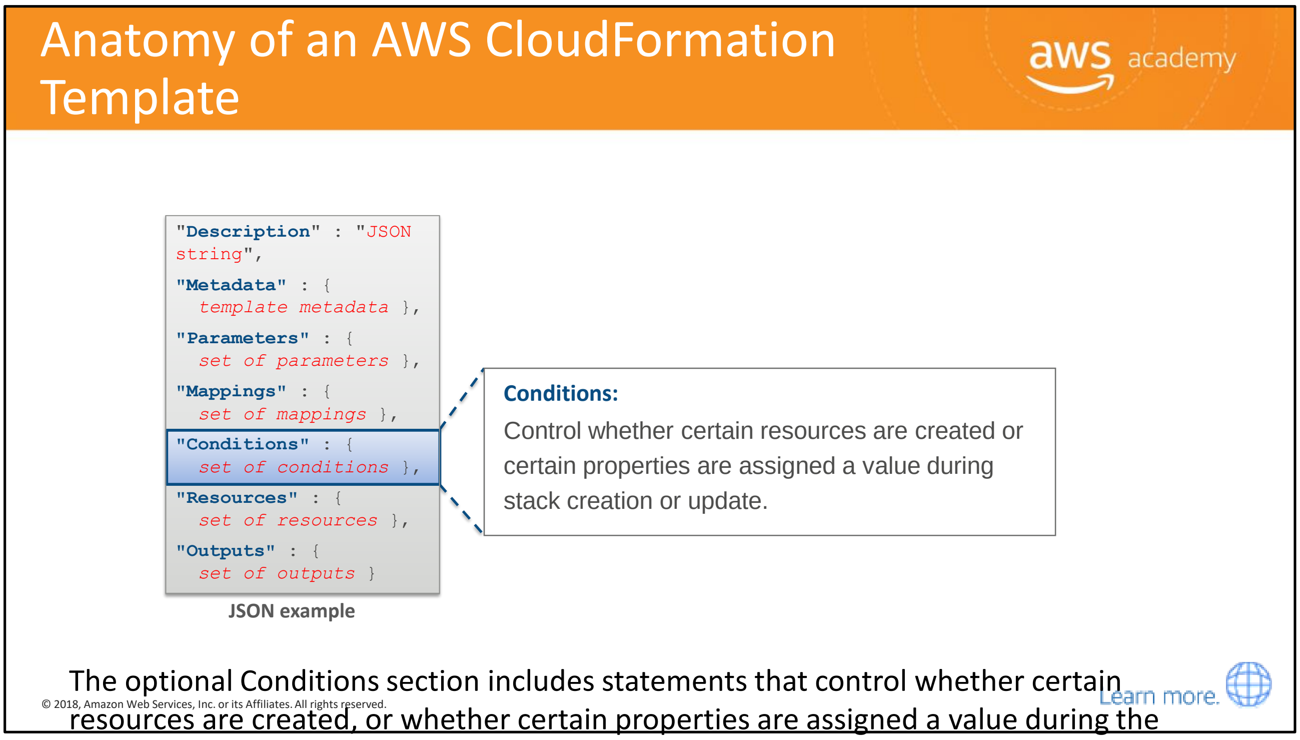
```
"Mappings" : {
  "RegionAndInstanceTypeToAMIID" : {
      "us-east-1": {
        "m1.small": "ami-1ccae774",
        "t2.micro": "ami-1ecae776"
      },
      "us-west-2" : {
        "m1.small": "ami-ff527ecf",
        "t2.micro": "ami-e7527ed7"
      }
}
```

JSON example

Specify multiple mapping levels

In the template, you can use Regions and specify multiple mapping levels.

In the example on the slide, this mapping specifies an AMI based on the type of instance that is launched within a specific Region. For example, if an m1.small instance is used, the AMI that will be used is ami-1ccae774. This mapping ties specific machine images to instances.



Anatomy of an AWS CloudFormation Template

```
"Description" : "JSON
string",
"Metadata" : {
  template metadata },
"Parameters" : {
  set of parameters },
"Mappings" : {
  set of mappings },
"Conditions" : {
  set of conditions },
"Resources" : {
  set of resources },
"Outputs" : {
  set of outputs }
```

JSON example

Conditions:
Control whether certain resources are created or certain properties are assigned a value during stack creation or update.

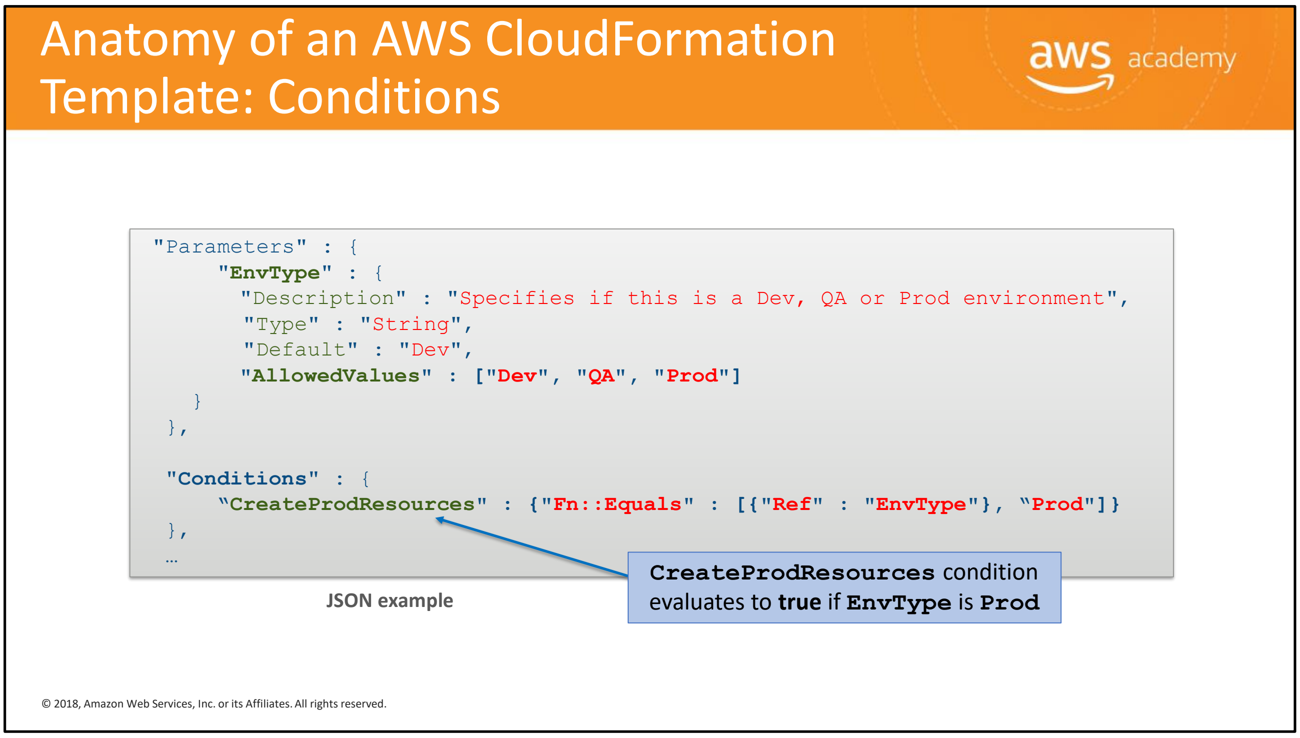
Learn more.

The optional Conditions section includes statements that control whether certain resources are created, or whether certain properties are assigned a value during the creation or update of a stack. For example, you can compare whether a value is equal to another value. Based on the result of that condition, you can conditionally create resources. If you have multiple conditions, separate them with commas.

You might use conditions when you want to reuse a template that can create resources in different contexts, such as a test environment versus a production environment. In your template, you can add an EnvironmentType input parameter, which accepts either “prod” or “test” as inputs. For the production environment, you might include Amazon EC2 instances with certain capabilities; however, for the test environment, you want to use reduced capabilities to save money. With conditions, you can define which resources are created, and how they're configured for each environment type.

Conditions are evaluated based on input parameter values that you specify when you create or update a stack. Within each condition, you can reference another condition, a parameter value, or a mapping. After you define all your conditions, you can associate them with resources and resource properties in the Resources and Outputs sections of a template.

For more information, see the AWS CloudFormation documentation on how to create and use conditions.
http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/conditions-section-structure.html



Anatomy of an AWS CloudFormation Template: Conditions

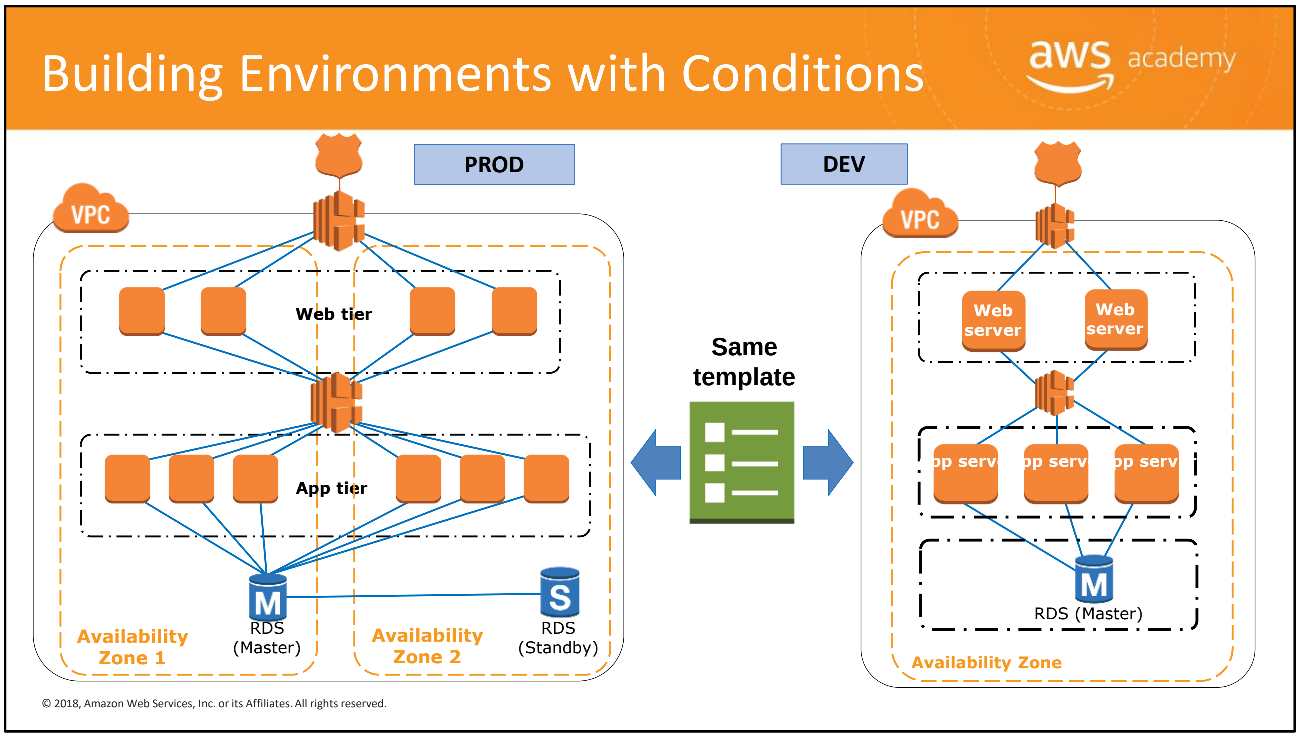
```
"Parameters" : {
     "EnvType" : {
       "Description" : "Specifies if this is a Dev, QA or Prod environment",
       "Type" : "String",
       "Default" : "Dev",
       "AllowedValues" : ["Dev", "QA", "Prod"]
   }
 },

 "Conditions" : {
     "CreateProdResources" : {"Fn::Equals" : [{"Ref" : "EnvType"}, "Prod"]}
 },
 …
```

JSON example

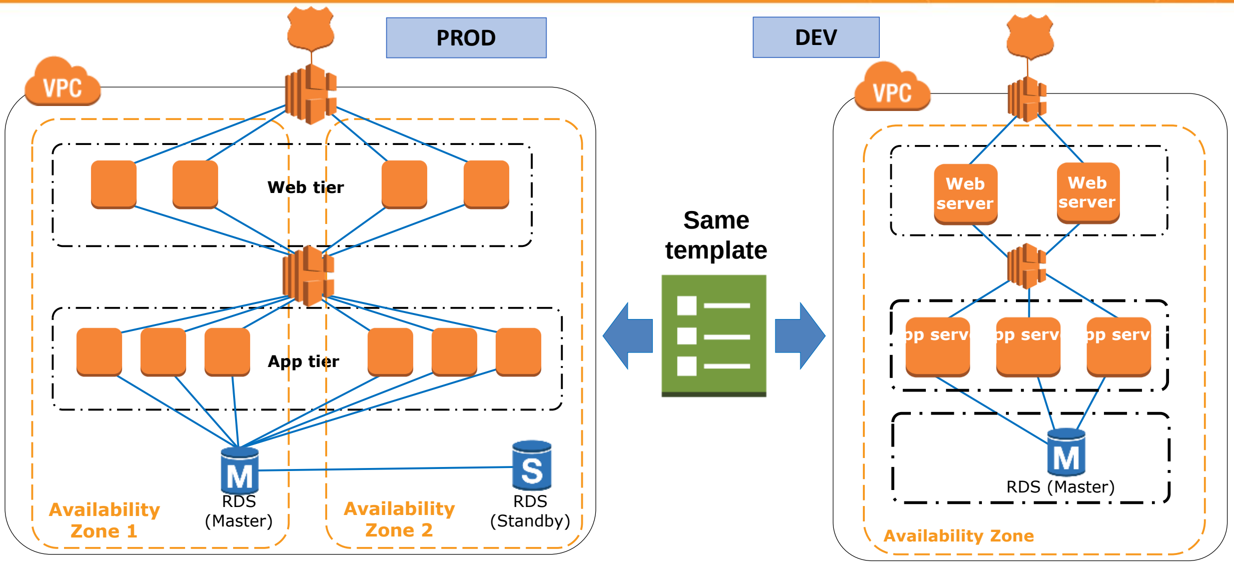
CreateProdResources condition evaluates to **true** if **EnvType** is **Prod**

This is an important one to pay attention to because if values or tags have been assigned, the template will do something different based on the assigned value.

In this example, the EnvType parameter specifies whether you want to create a Dev environment, a QA—environment, or a Prod environment. Depending on the environment, you might want to specify different configurations, such as which database it points to. You can use “Condition” to evaluate this, and specify appropriate resources for each environment.

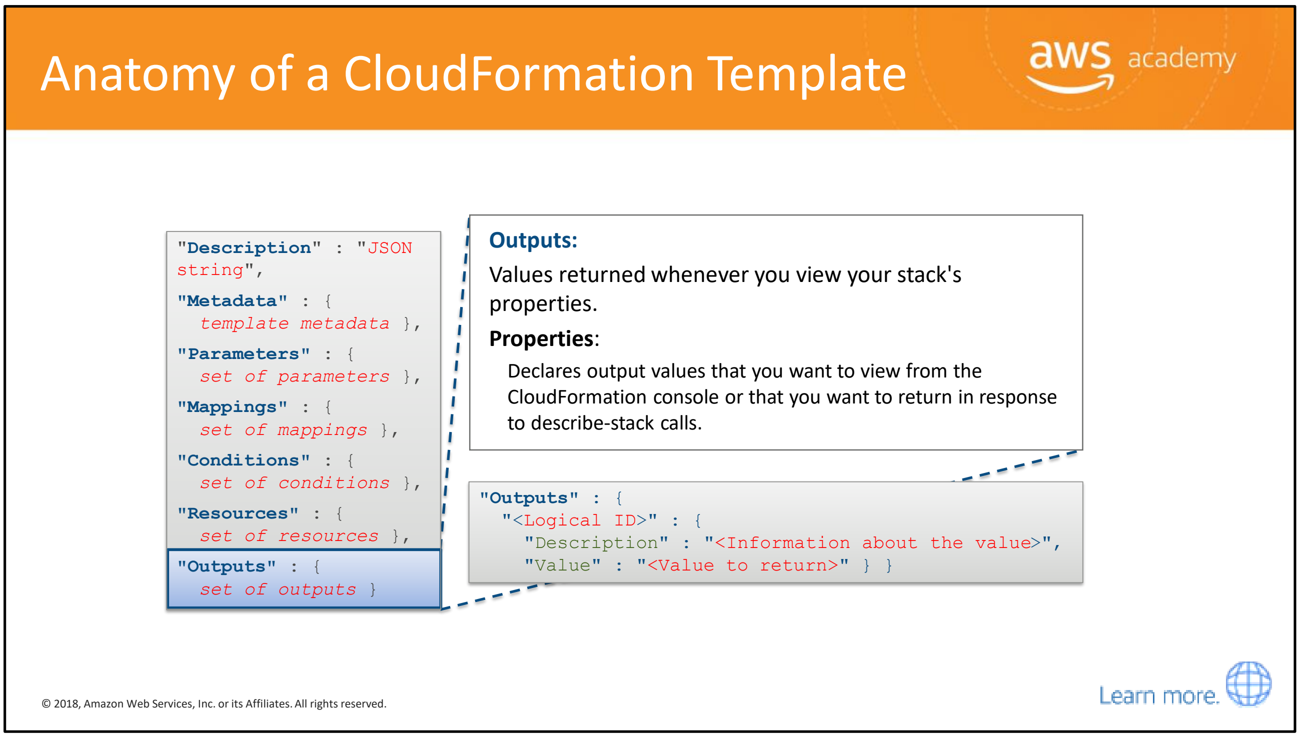


Building Environments with Conditions



When the previous template is applied to this example, only one set of resources in one Availability Zone is launched when the target environment is development, or DEV. When this template is used in production--or PROD—the solution launches two sets of resources in two different Availability Zones. So, without making a single change, you can get a redundant environment from the same template.

Your production environment and development environment must have the same stack in order to ensure that your application works the way that it was designed. Your DEV environment and QA environment must have the same stack of applications and the same configuration. You might have several QA environments for functional testing, user acceptance testing, load testing, and so on. The process of creating those environments manually can be -prone. You can use a Conditions statement in the template to solve this problem.



Anatomy of a CloudFormation Template

```
"Description" : "JSON
string",
"Metadata" : {
  template metadata },
"Parameters" : {
  set of parameters },
"Mappings" : {
  set of mappings },
"Conditions" : {
  set of conditions },
"Resources" : {
  set of resources },
"Outputs" : {
  set of outputs }
```

Outputs:

Values returned whenever you view your stack's properties.

Properties:

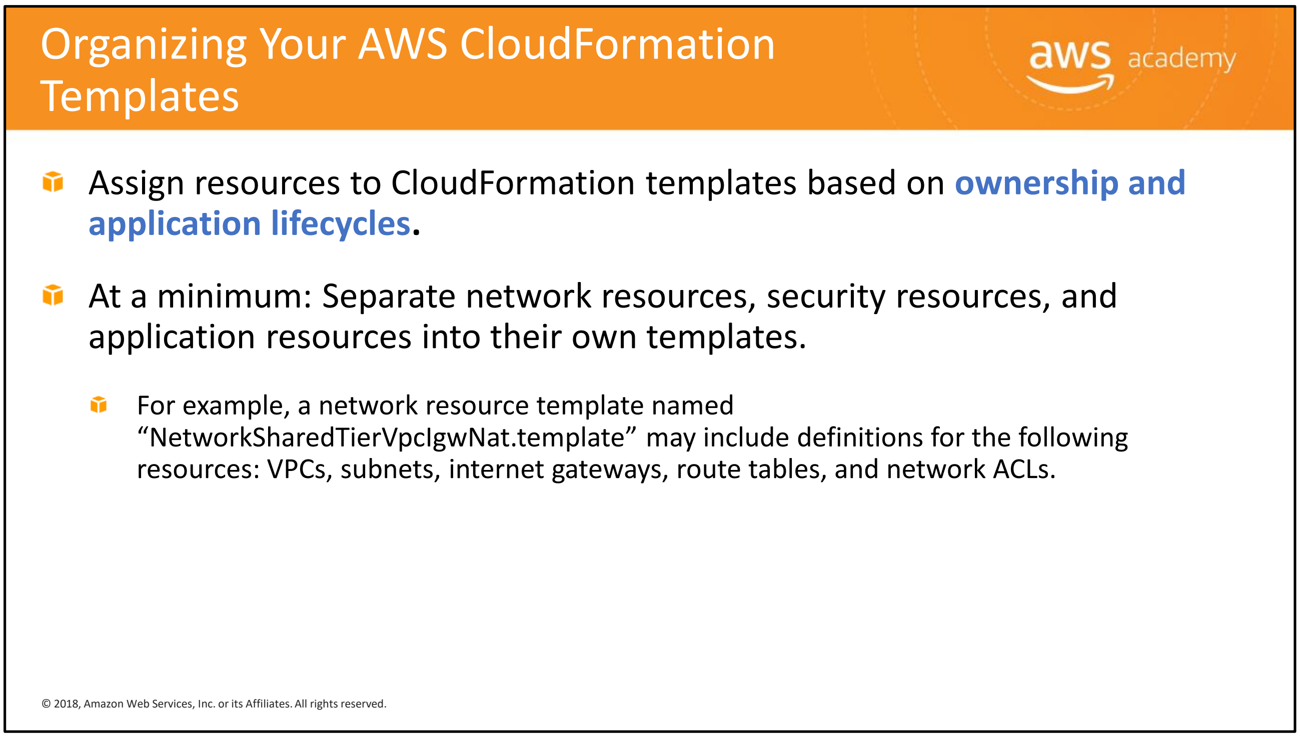
Declares output values that you want to view from the CloudFormation console or that you want to return in response to describe-stack calls.

```
"Outputs" : {
  "<Logical ID>" : {
    "Description" : "<Information about the value>",
    "Value" : "<Value to return>" } }
```

With an AWS CloudFormation template, you can specify an output. Outputs are values that are returned whenever you view the properties of your stack. For example, if something executes properly, it is helpful to provide an indication that the execution completed and was successful.

Outputs can specify the string output of any logical identifier that is available in the template. It's a convenient way to capture important information about your resources or input parameters.

For more information, go to the AWS CloudFormation documentation on how to create and use outputs.
http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/outputs-section-structure.html



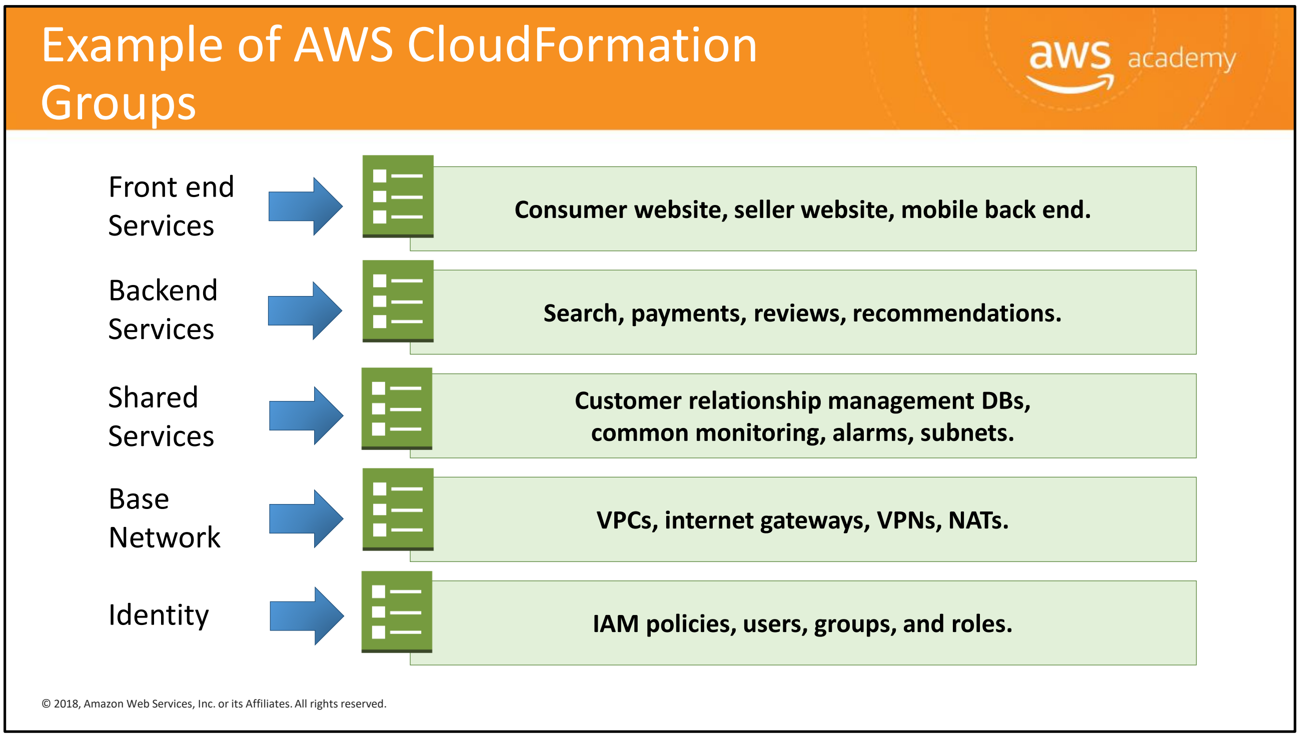
Organizing Your AWS CloudFormation Templates

- Assign resources to CloudFormation templates based on **ownership and application lifecycles**.
- At a minimum: Separate network resources, security resources, and application resources into their own templates.
  - For example, a network resource template named “NetworkSharedTierVpcIgwNat.template” may include definitions for the following resources: VPCs, subnets, internet gateways, route tables, and network ACLs.

While templates can be reused to create multiple environments or parts of environments, we do not recommend building all of an application's within one template .

Resources should be grouped into templates based on their ownership and their place in the application lifecycle. At a minimum, you should separate network resources, security resources, and application resources into their own templates. With security resources, you might want to lock them down by separating them from the rest of your templates. For example, a network resource template named “NetworkSharedTierVpcIgwNat.template” might include definitions for the following resources: VPCs, subnets, internet gateways, route tables, and network access control lists, or ACLs.

Also, a test environment and a production environment should probably not share the same templates in most cases. Resources in a test environment will need to change frequently, while resources in a production environment should be relatively stable. In addition, we do not recommend sharing templates across management teams because different needs and standards can impact teams inappropriately.

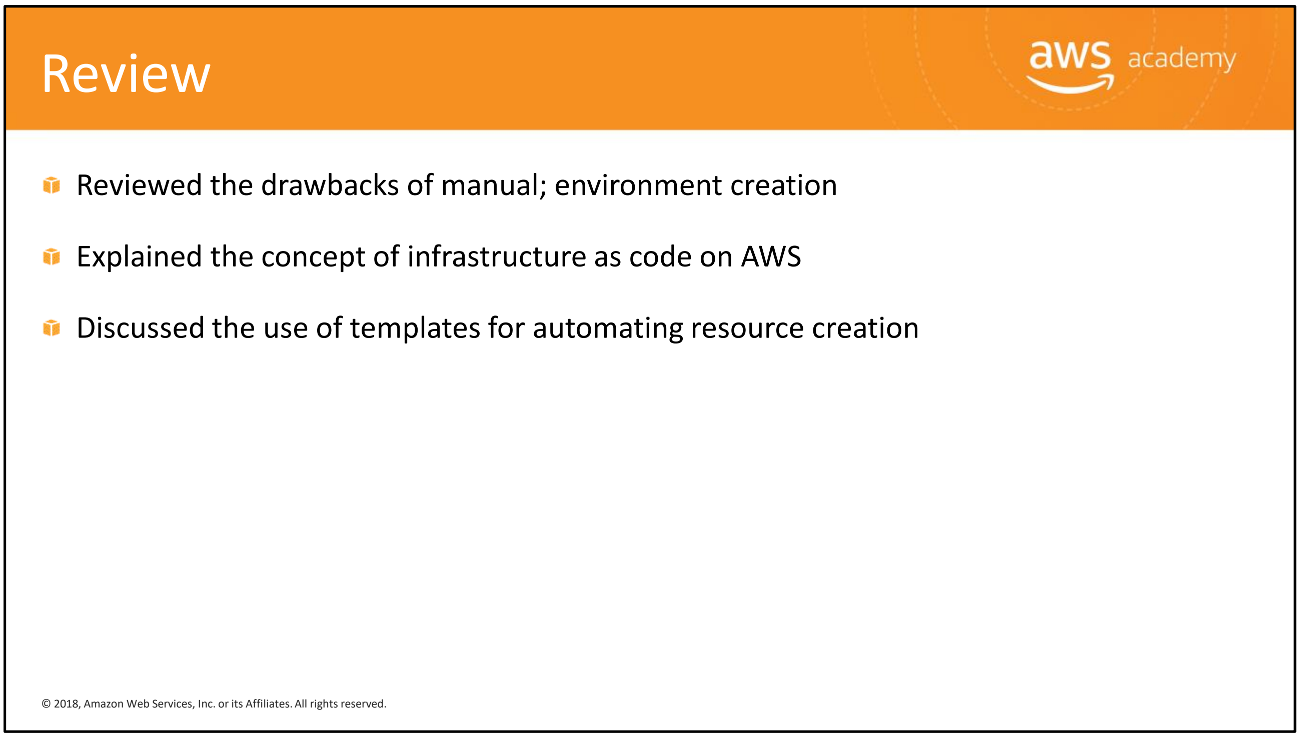


Example of AWS CloudFormation Groups

aws academy
Front end Services
Consumer website, seller website, mobile back end.
Backend Services
Search, payments, reviews, recommendations.
Shared Services
Customer relationship management DBs, common monitoring, alarms, subnets.
Base Network
VPCs, internet gateways, VPNs, NATs.
Identity
IAM policies, users, groups, and roles.

Here's an example of how stack can be grouped with AWS CloudFormation groups. You could have front-end services like a consumer website, a seller website, or a mobile backend. There might be backend services for search, payments, reviews or recommendations. Shared services could include customer relationship management databases, common monitoring, alarms, and subnets. The base network could include VPCs, internet gateways, virtual private networks or VPNs, and network address translation—or NAT. Finally, identity could include IAM policies, users, groups, and roles.

When you think about how to bundle resources into your AWS CloudFormation templates, a good guideline is to organize the resources like they are software. Think about the tightly connected components to your infrastructure, and put them in the same templates. In this example, AWS CloudFormation resources are grouped into five different templates: front-end services, backend services, shared services, base network services, and identity resources.



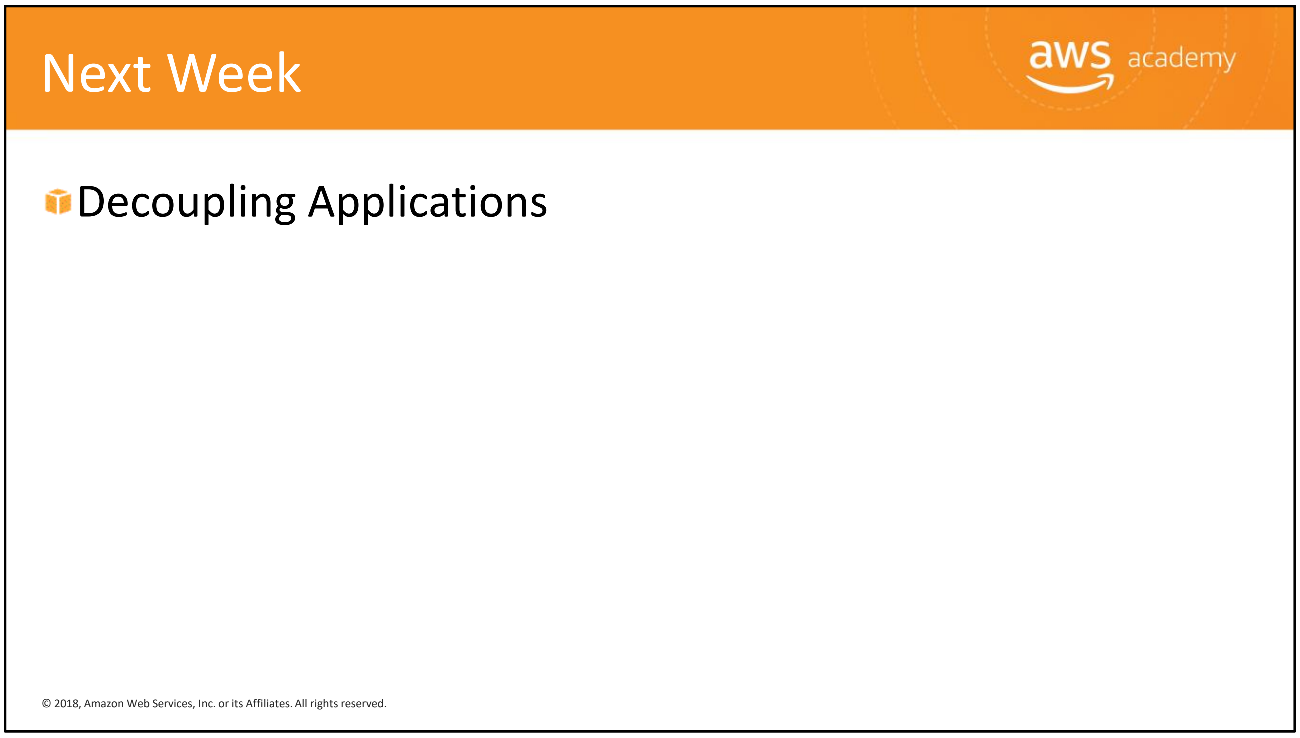
Review

- Reviewed the drawbacks of manual; environment creation
- Explained the concept of infrastructure as code on AWS
- Discussed the use of templates for automating resource creation

In review, we:

- Reviewed the drawbacks of manual; environment creation
- Explained the concept of infrastructure as code on AWS
- Discussed the use of templates for automating resource creation
- Reviewed other automation resources and the convenience versus control of each tool

To finish this module, please complete the lab and the corresponding knowledge assessment.



Next Week

- Decoupling Applications

In review, we:

- Reviewed the drawbacks of manual; environment creation
- Explained the concept of infrastructure as code on AWS
- Discussed the use of templates for automating resource creation
- Reviewed other automation resources and the convenience versus control of each tool

To finish this module, please complete the lab and the corresponding knowledge assessment.