Building Serverless Solutions on AWS
Student Guide
Version 1.0

AWS-200-BSS-10-EN

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates. All rights reserved.

This work may not be reproduced or redistributed, in whole or in part, without prior written permission from Amazon Web Services, Inc. Commercial copying, lending, or selling is prohibited.

Corrections or feedback on the course, please email us at:
aws-course-feedback@amazon.com.
For all other questions, contact us at:
https://aws.amazon.com/contact-us/aws-training/.

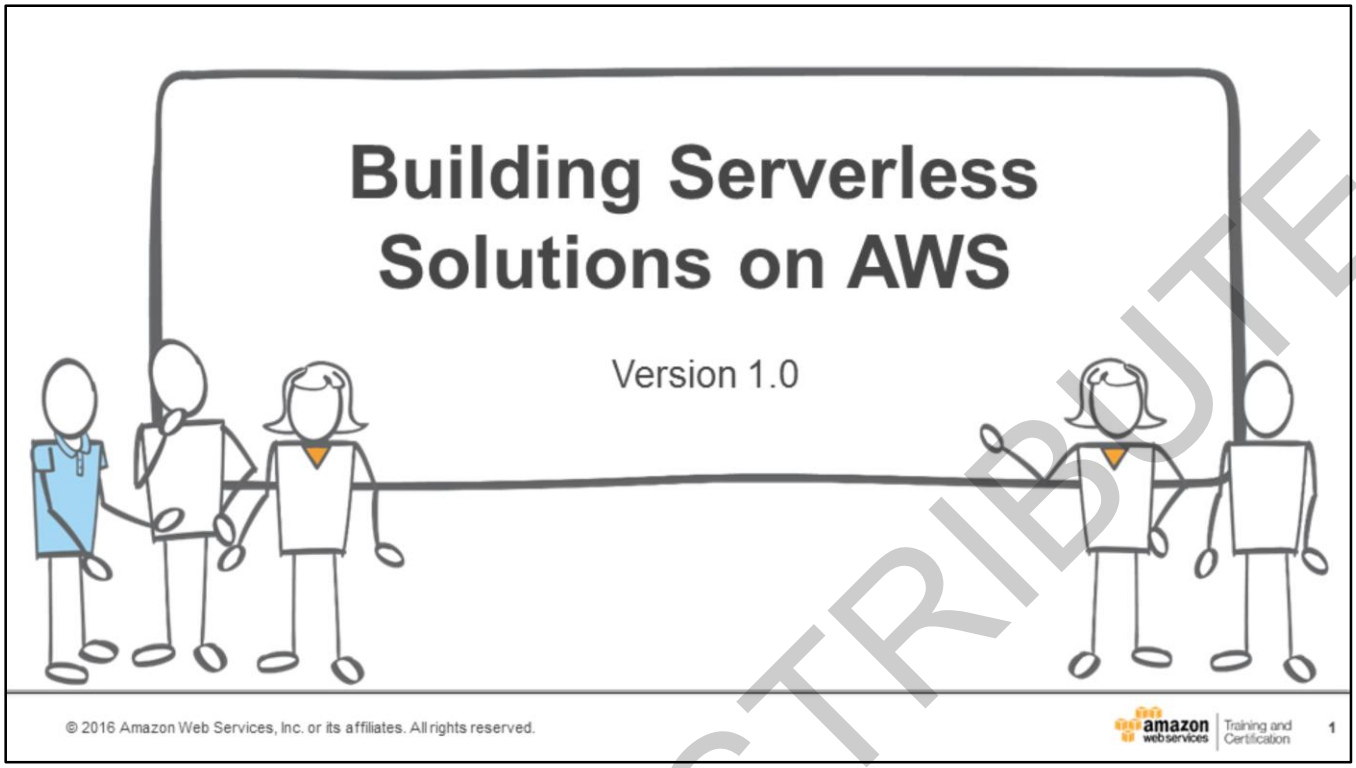
All trademarks are the property of their owners.

DO NOT DISTRIBUTE

Contents

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



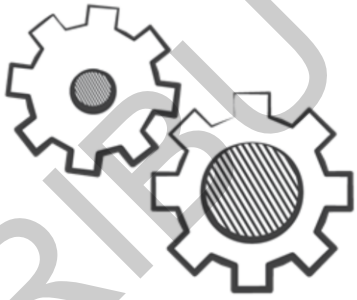
Building Serverless Solutions on AWS is a one-day, advanced-level bootcamp that demonstrates how to create applications using AWS Lambda, Amazon DynamoDB, Amazon API Gateway, and other serverless services. You will get hands-on experience using established patterns and best practices to design and build dynamic web applications, stream processors, and batch data processing systems without servers. This bootcamp will cover design and implementation considerations for building scalable and highly available systems and techniques for integrating these solutions with conventional server-based applications.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

Introductions and Logistics

- Class introductions
- Logistics
- Participation
- Cell phones



© 2016 Amazon Web Services, Inc. or its affiliates. All rights reserved.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

Module Layout

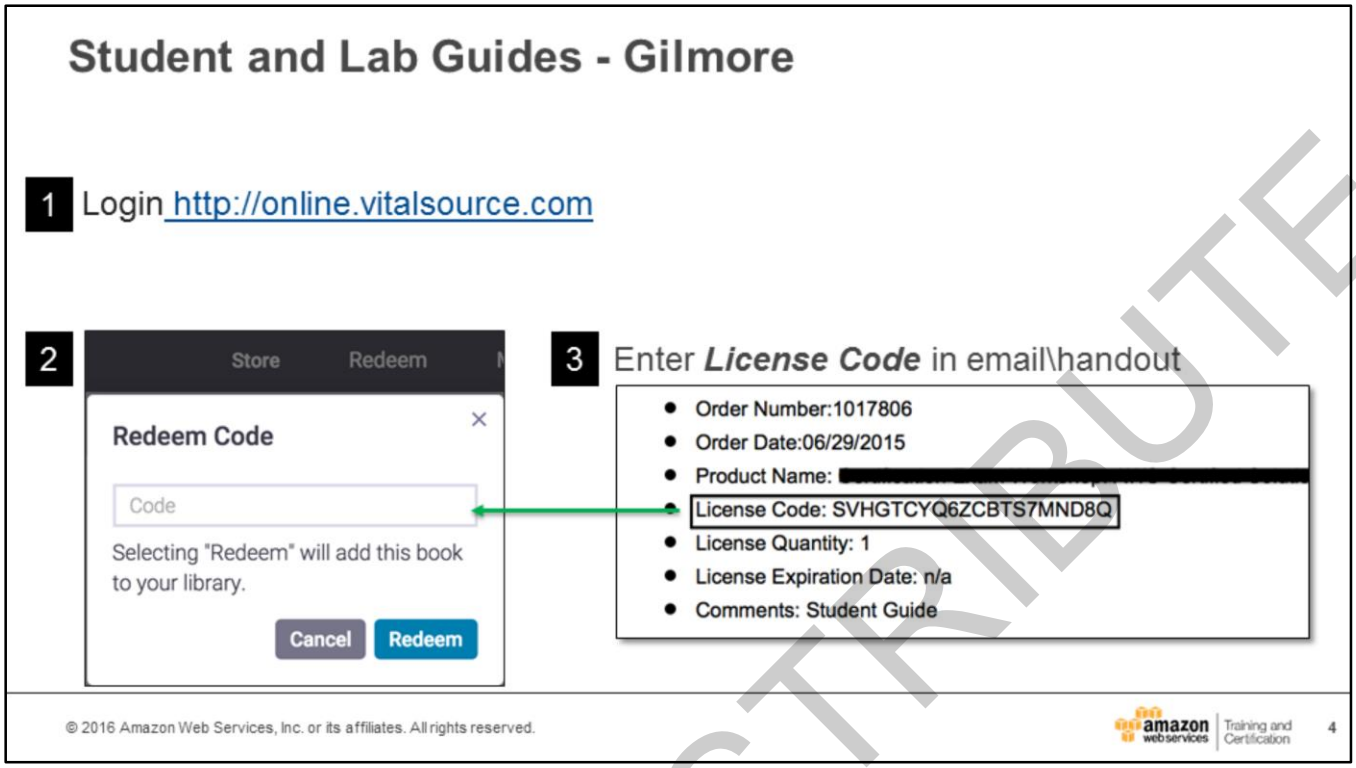
- **Module 1**: Introduction to Serverless
- **Module 2**: Microservices and Web Applications
  - Lab 1: Build a serverless web app with Amazon S3, AWS API Gateway, AWS Lambda and Amazon DynamoDB
- **Module 3**: Streaming Data Processing
  - Lab 2: Build a streaming analytics processor and notification engine
- **Module 4**: Batch Processing and Scheduled Functions
  - Lab 3: Build a parallelized batch data processor
- **Module 5**: VPC Integration and Hybrid Architectures
  - Lab 4: Integrate your serverless web app with VPC resources
- **Module 6**: Course Wrap-Up

© 2016 Amazon Web Services, Inc. or its affiliates. All rights reserved.

amazon web services | Training and Certification 3

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



DO NOT DISTRIBUTE

QwikLABS Access

Login (Safari, Chrome, or Firefox preferred)
https://aws.qwiklab.com

© 2016 Amazon Web Services, Inc. or its affiliates. All rights reserved.
amazon web services | Training and Certification 5

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

Logging into the AWS Management Console from QwikLABS

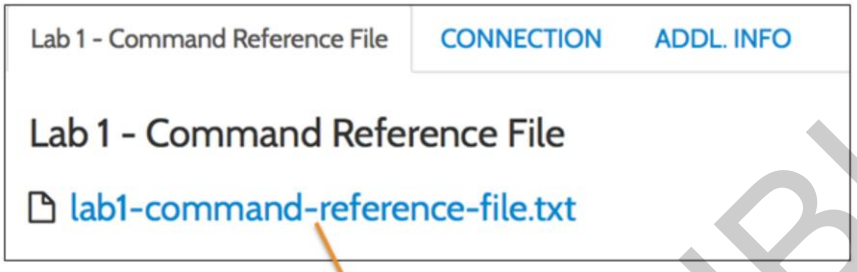
Gilmore VitalSource Bookshelf contains the lab manual.
NOTE: Do not change the default selected region.

© 2016 Amazon Web Services, Inc. or its affiliates. All rights reserved.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

Lab Login – Command Reference File



Download the file to copy the latest scripts used in the lab.
NOTE: This file is not a lab guide.

© 2016 Amazon Web Services, Inc. or its affiliates. All rights reserved.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

Module 2: Microservices and Web Applications

© 2016 Amazon Web Services, Inc. and its affiliates. All rights reserved.

amazon web services | Training and Certification

1

DO NOT DISTRIBUTE

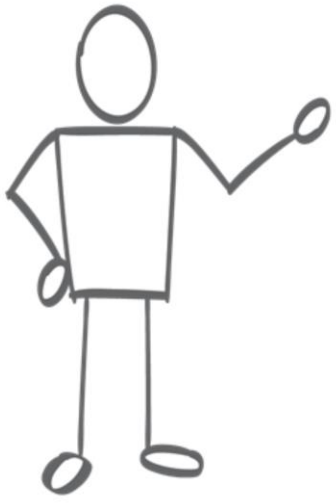
Agenda

- Introduction to Microservices
- Lambda Introduction
- RESTful Web Services and API Gateway
- DynamoDB Introduction
- HTML/AJAX Application Primer
- Lab Architecture Overview

© 2016 Amazon Web Services, Inc. and its affiliates. All rights reserved.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



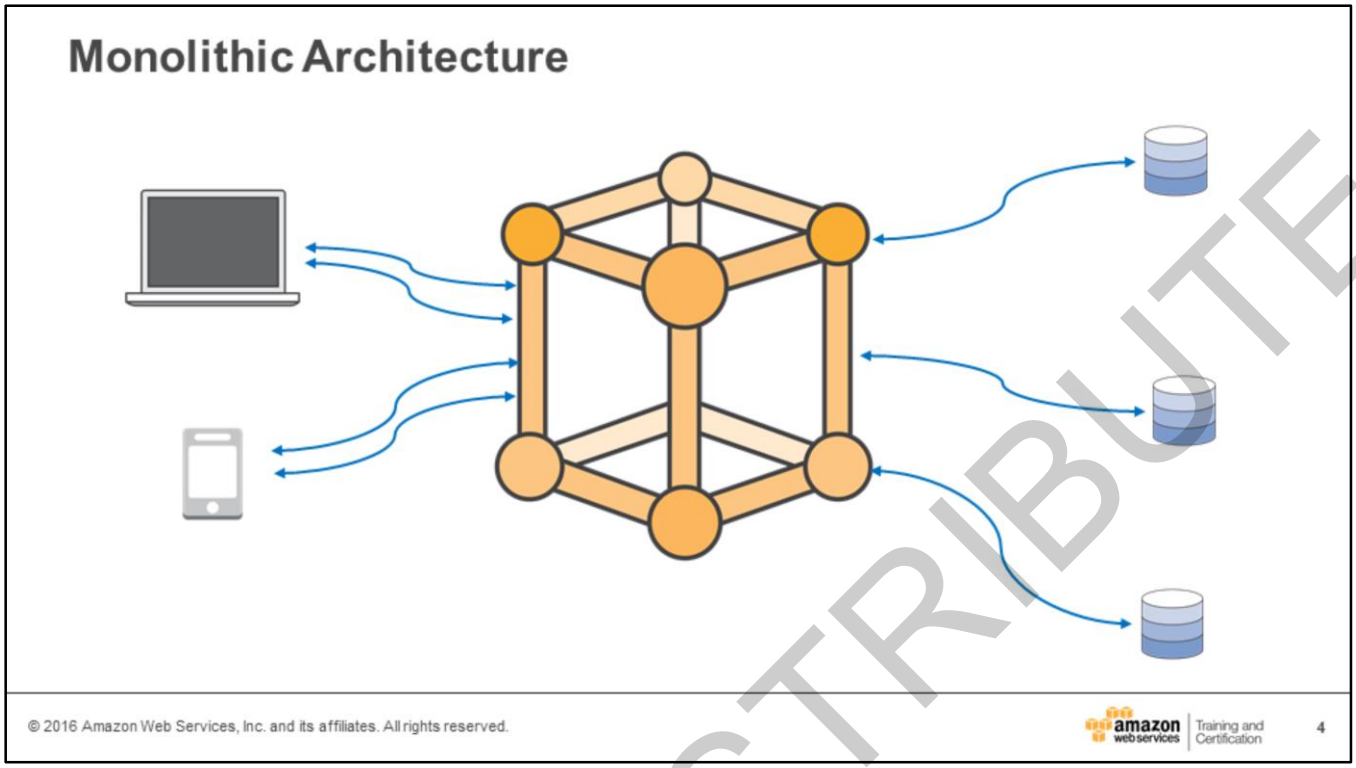
Introduction to Microservices

© 2016 Amazon Web Services, Inc. and its affiliates. All rights reserved.

amazon web services | Training and Certification

3

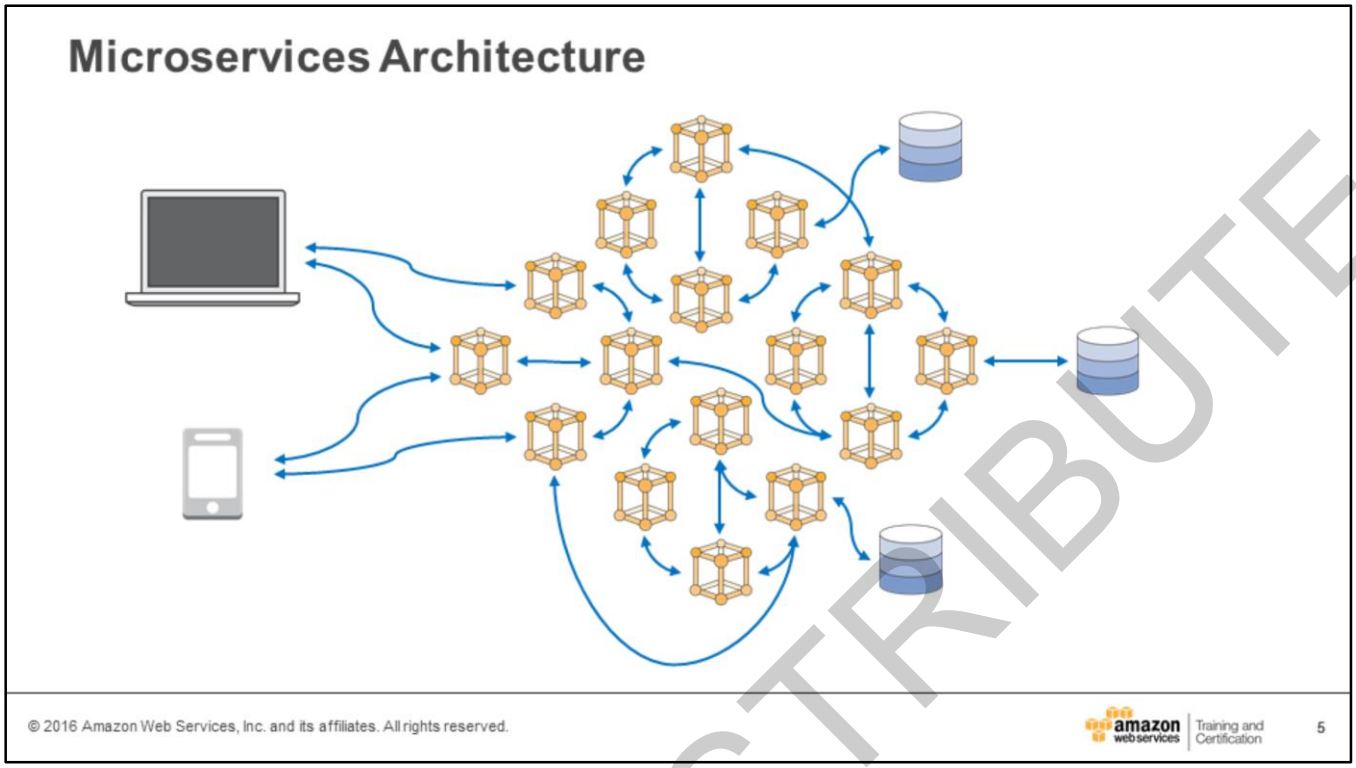
DO NOT DISTRIBUTE



Monolithic architectures combine all of an application's layers, (e.g. user interface, business logic, persistence) into a single deployable unit. Monolithic systems generally have many integration points where data flows in and out of the system and they are responsible for executing every step needed to complete every function exposed by the application.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



In contrast to monolithic applications, each functional component of a larger application is broken down into a smaller, single-function modular web service in a microservices architecture. The *micro* in microservices refers to "Single Responsibility," not size. It is best to start with a few number of coarse-grained but self-contained services and then as the implementation matures over time, refactor into more fine-grained services.

When adopting microservices, standardize service templates to provide boilerplates for development. This will reduce development chatter and the initial ramp-up time for development teams.

In addition, make sure to agree on a standard interface for communication between microservices, your legacy monolithic applications, and the new microservices.

There are common components that will need to be established as core to your architecture and approach. Some of these are:

- Authentication/Authorization
- Service registration, discovery, wiring, administration
- State management
- Service metadata

DO NOT DISTRIBUTE

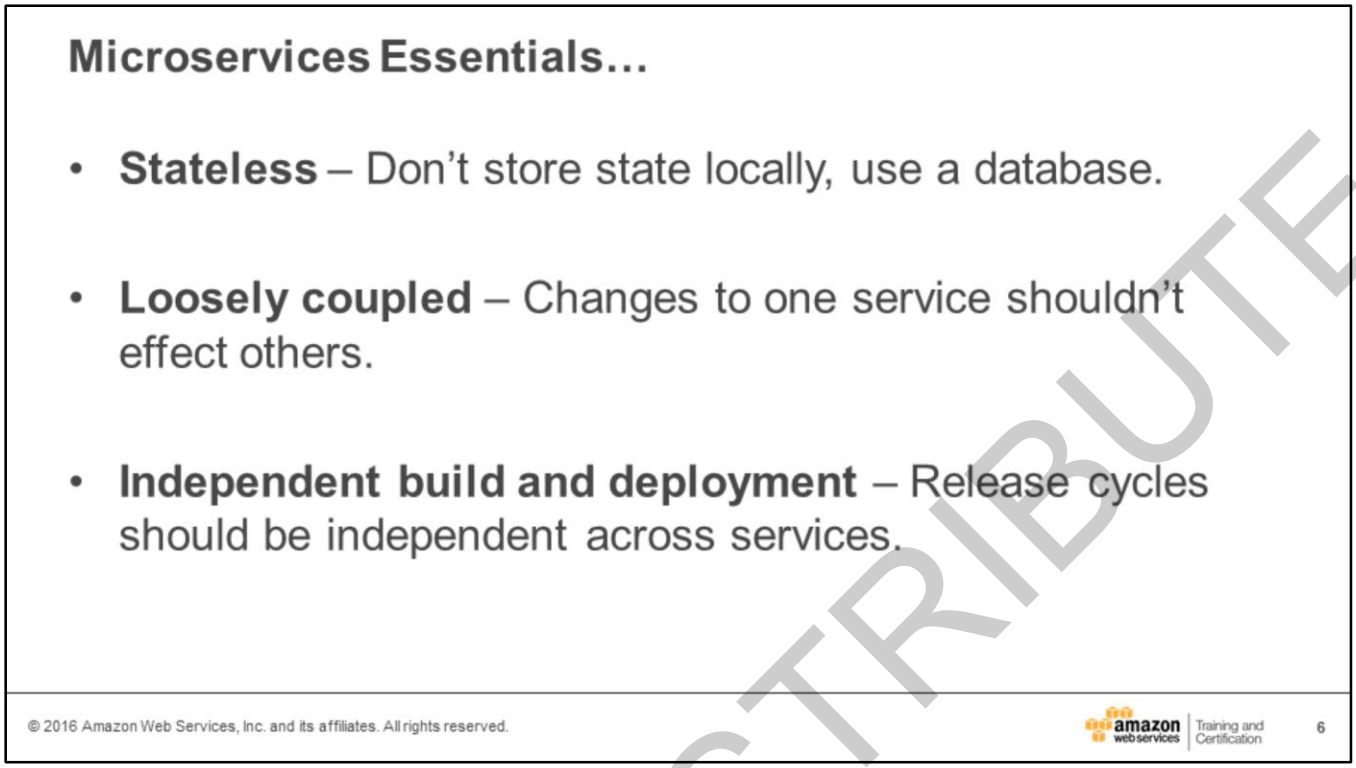
© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

- Service versioning
- Service orchestration/chaining
- Configuration Service
- Caching

You should have a plan for each of these for your architecture.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



Some of the key concepts to keep in mind when architecting your microservices are to:

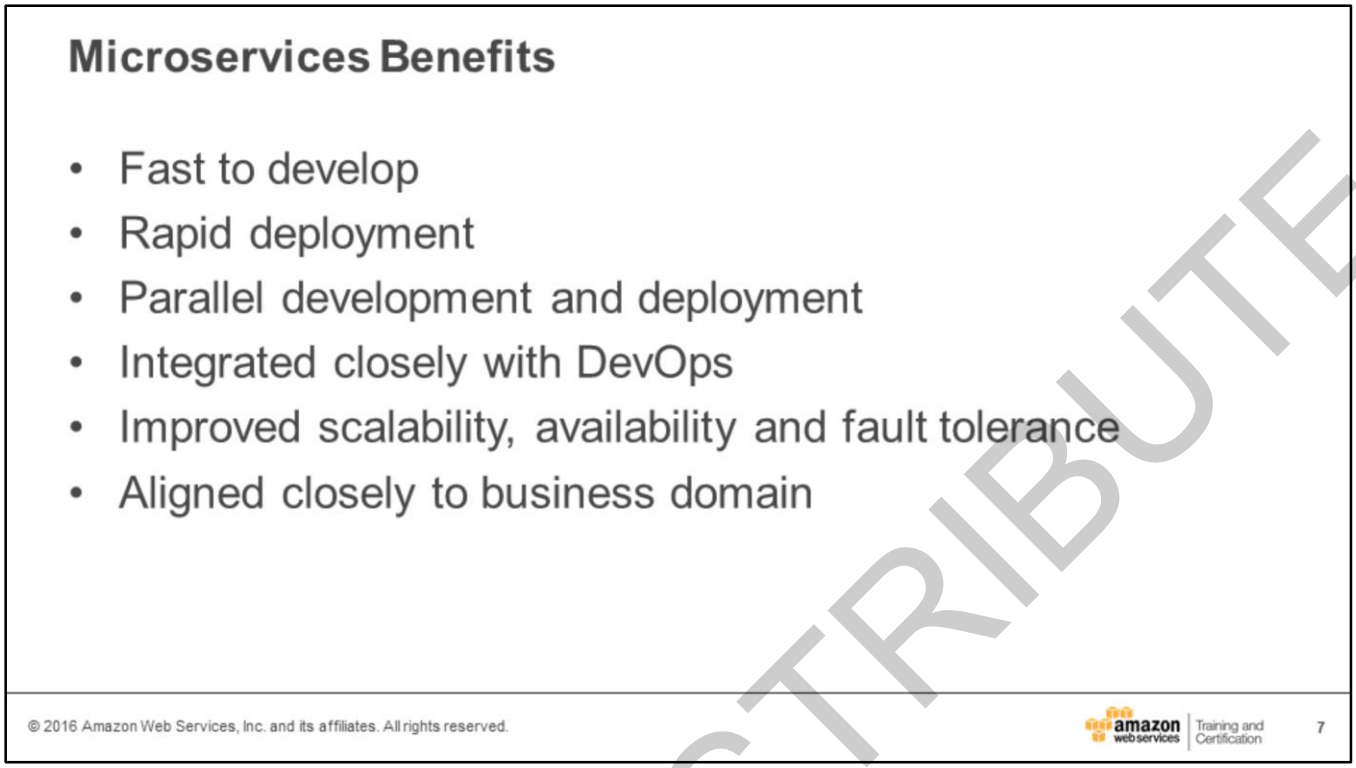
Stateless – Don't store state local to the processes that are running your services. Your services should be able to scale horizontally and any request should be able to be handled by any instance of your service. Any state you do want to persist should be stored in a shared data layer such as a DynamoDB table. This should not be confused with caching. Often times it makes sense to cache data or connection objects locally for performance reasons.

Loosely coupled – Services should expose APIs that abstract consumers from the internal implementation details. Updates to one service should not require redeploying others.

Independent build and deployment – Each service should have its own independent build and deployment process. Build and release cycles across services should be completely separate.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



There are numerous benefits to adopting a microservices approach for your architecture.

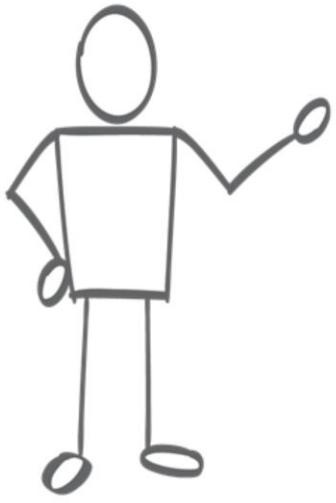
Breaking up your application makes it faster to develop because developing smaller chunks of code makes it easier to code and test those pieces. In addition, it means you can re-use more because of the fine-grained nature of the services. The more re-use you have of your microservices, the faster the subsequent services will be to develop.

Microservices helps make deployments quicker. Your application components are smaller and can be tested more easily and deployed easier. Additionally, when the application is broken into microservices teams or developers can work on the pieces in parallel and deploy in parallel leading to faster time to market. Due to this microservices naturally lend themselves to a DevOps culture.

Being able to deploy each microservice separately also allows you to scale the components individually.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

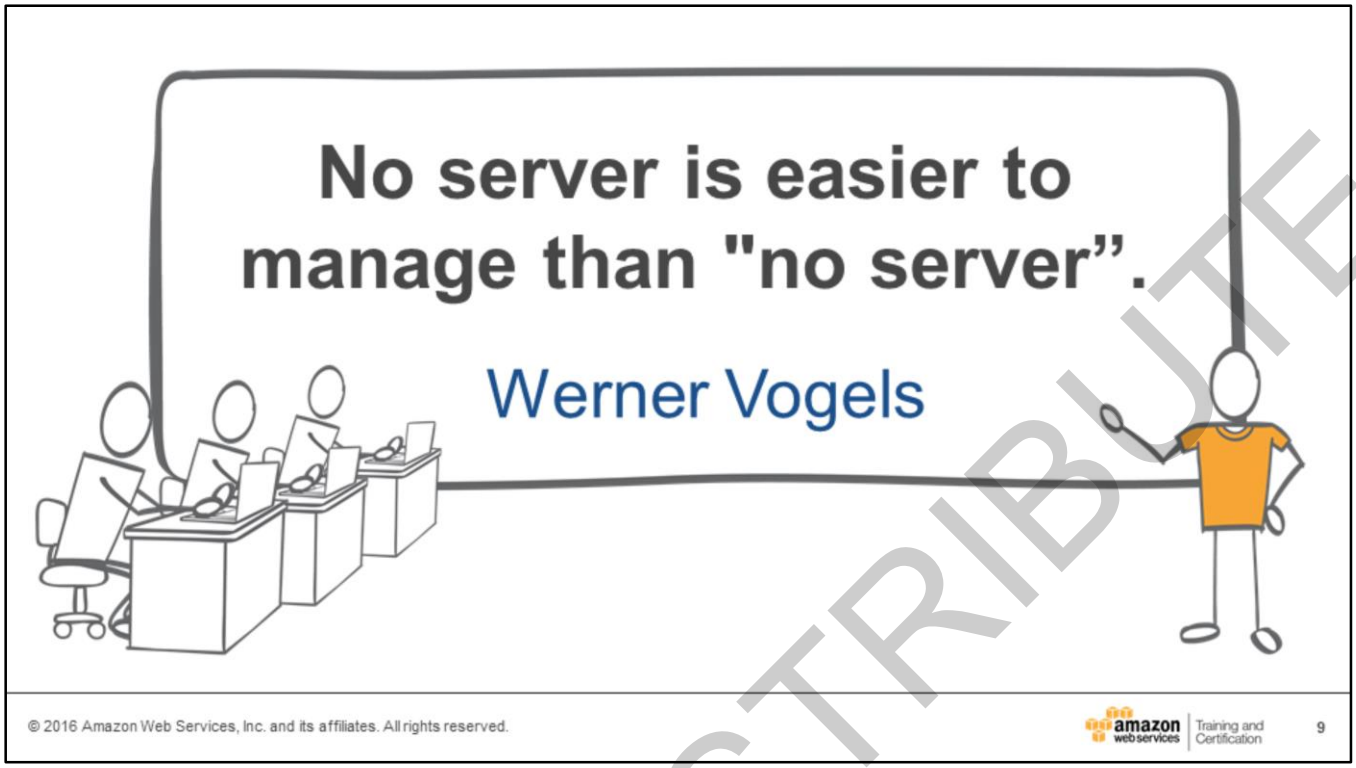


Lambda Introduction

© 2016 Amazon Web Services, Inc. and its affiliates. All rights reserved.

amazon web services | Training and Certification 8

DO NOT DISTRIBUTE



DO NOT DISTRIBUTE

Components of Lambda

- A Lambda Function (that you write)
- An Event Source
- The AWS Lambda Service
- The Function Networking Environment

© 2016 Amazon Web Services, Inc. and its affiliates. All rights reserved.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

The Lambda Function

- Your Code (Java, NodeJS, Python)
- The IAM role that code assumes during execution
- The amount of memory allocated to your code (affects CPU and Network as well) – from 128 MB to 1.5 GB in 64 MB increments

A valid, complete Lambda function

```
var AWS = require('aws-sdk');
var s3 = new AWS.S3();

exports.handler = function(event, context) {
    var params = {
                    Bucket: '[input bucket name here]',
                    Key: "[insert keyname here]",
                    Body: "[object body]"
                };
    s3.putObject(params, function(err, data) {
        if (err) {
            console.log(err, err.stack); // an error occurred
        }
        else {
            console.log(data);
        }
        context.done();
    } );
};
```

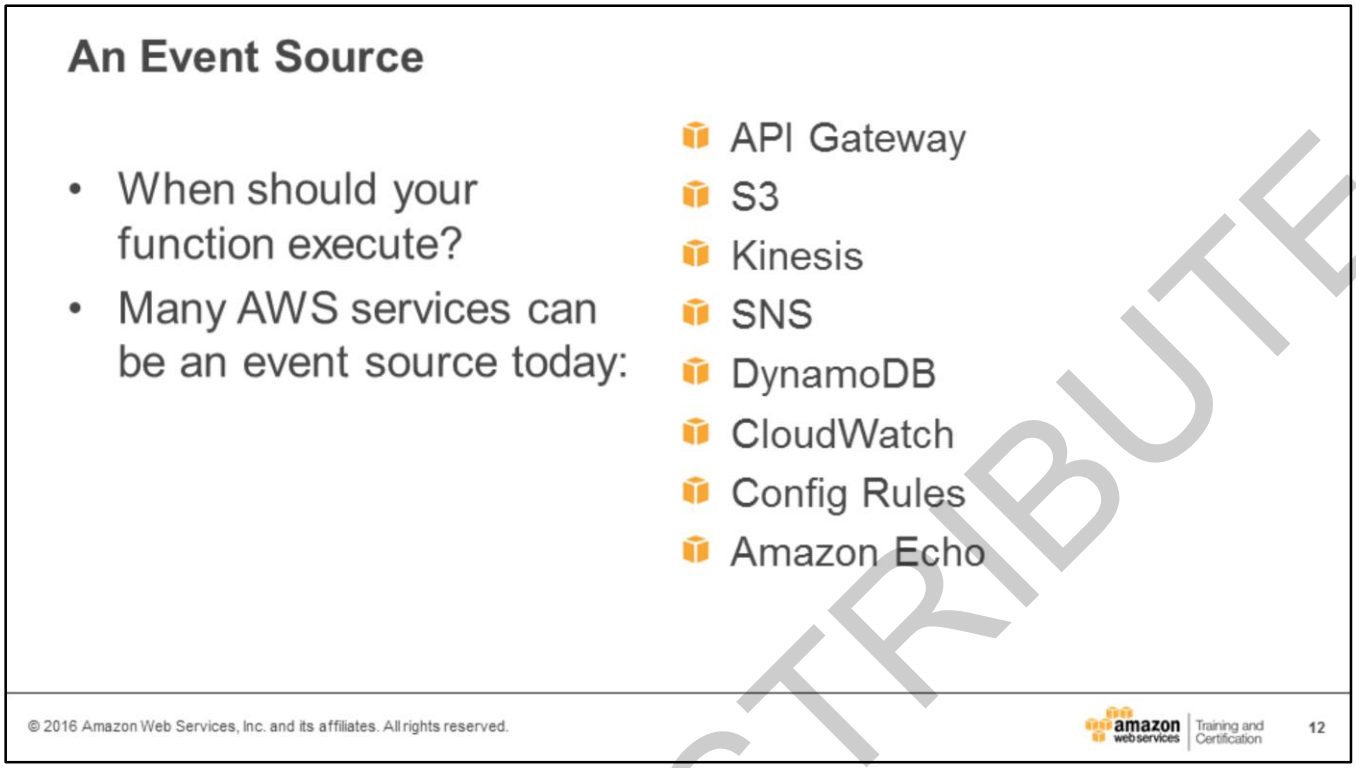
© 2016 Amazon Web Services, Inc. and its affiliates. All rights reserved.

amazon web services | Training and Certification 11

The most important component of a Lambda function is the code itself. Lambda functions can be authored in JavaScript, Python, or JVM-based languages like Java, Scala, or JRuby. You define a single function or method that that serves as the entry point to your code called the handler function. That function can then invoke other local functions, execute arbitrary binaries, or call external services.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

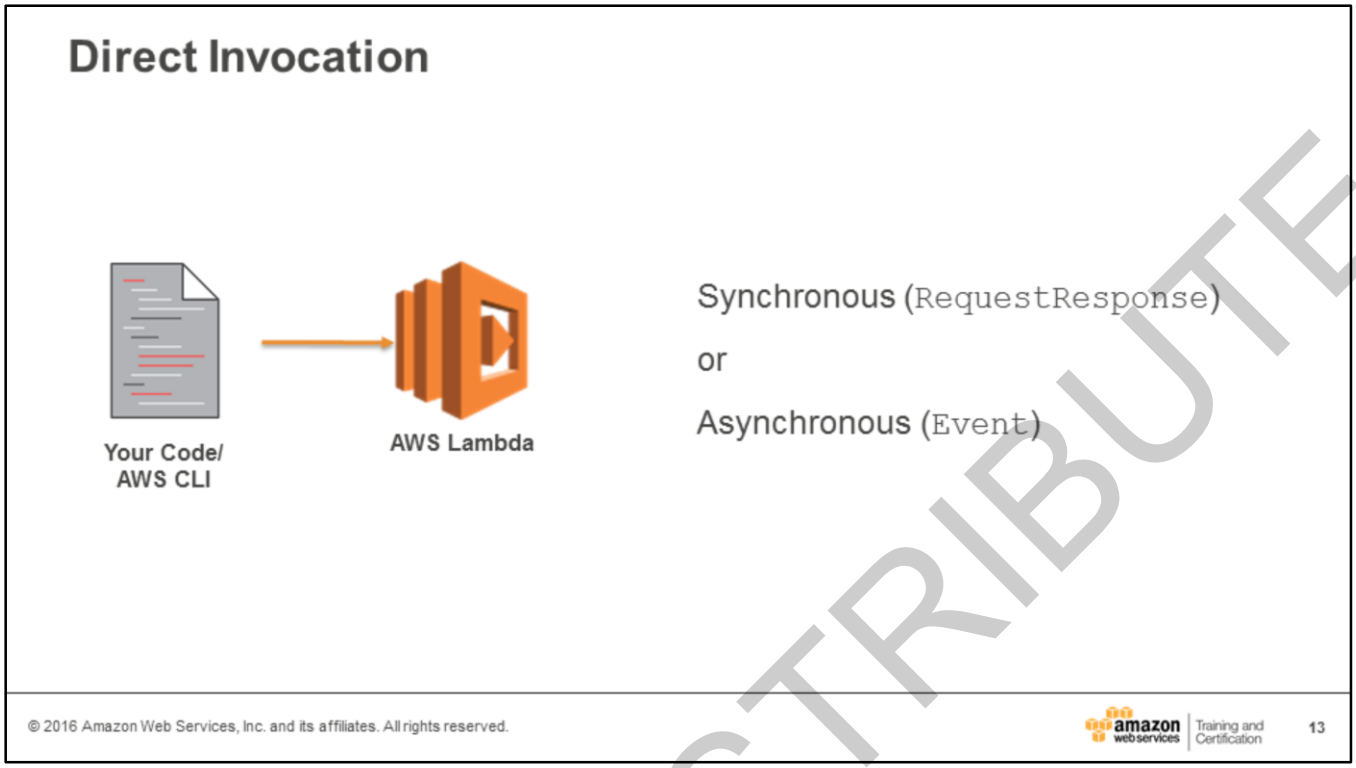


Next is the event source. For the code that you've written and would like to have executed, the event source will define how and when that occurs. There are a number of different event sources available today and that's continuing to grow at a very rapid pace.

Each different event source type will define what data and metadata are passed to your function so that it's able to process with all the context that it needs for your application. So, for example, if you choose Amazon API Gateway as your event source, your Lambda function will receive all of the HTTPS request details it needs to process that API request.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



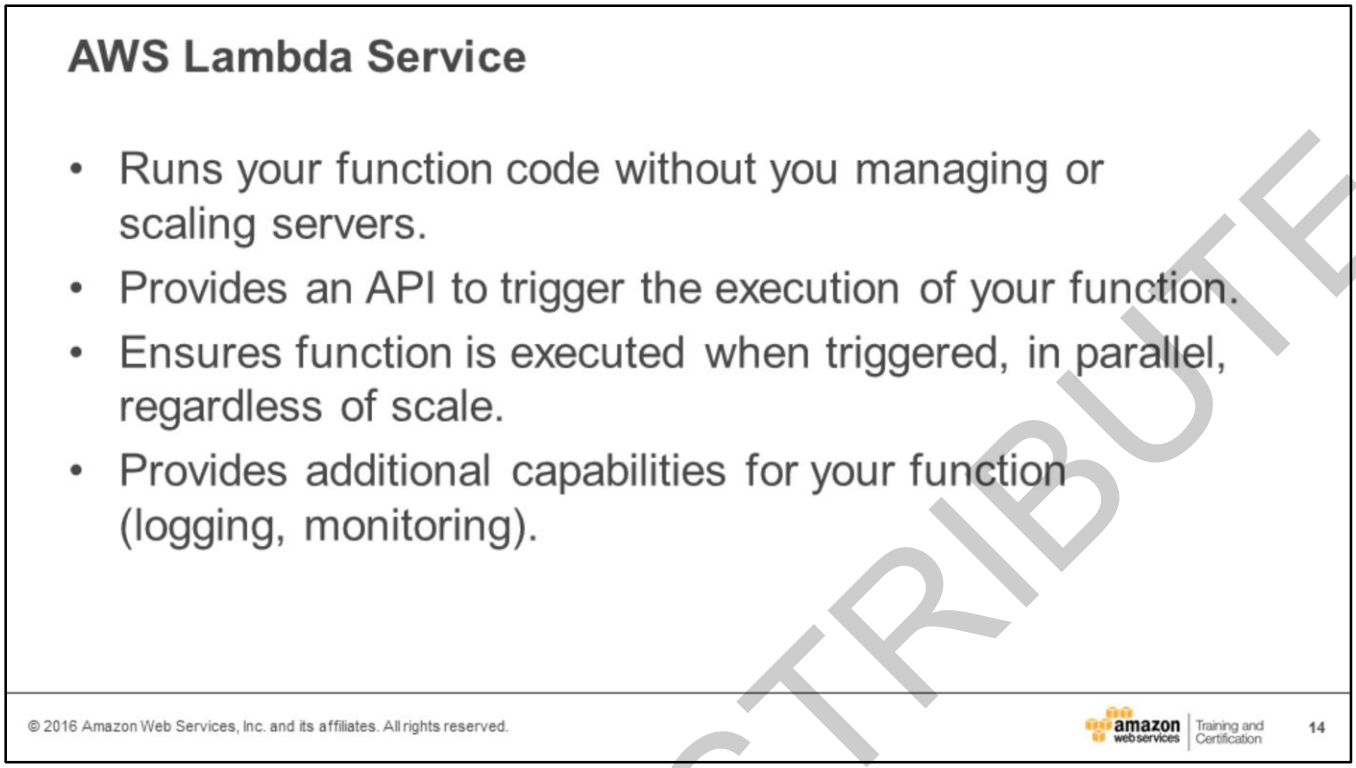
Lambda functions can also be invoked directly from your code or the AWS CLI using the Invoke action of the Lambda API.

The Invoke action exposes a parameter called InvocationType that can be used to control whether the invocation is synchronous or asynchronous. By setting the InvocationType to RequestResponse, the API call to Invoke will not return until the function has completed execution. The object returned by the function will be available in the response when using this InvocationType.

If you set the InvocationType to Event, the function will be executed asynchronously. When using the Event InvocationType the API call will return immediately and there will be no response payload.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



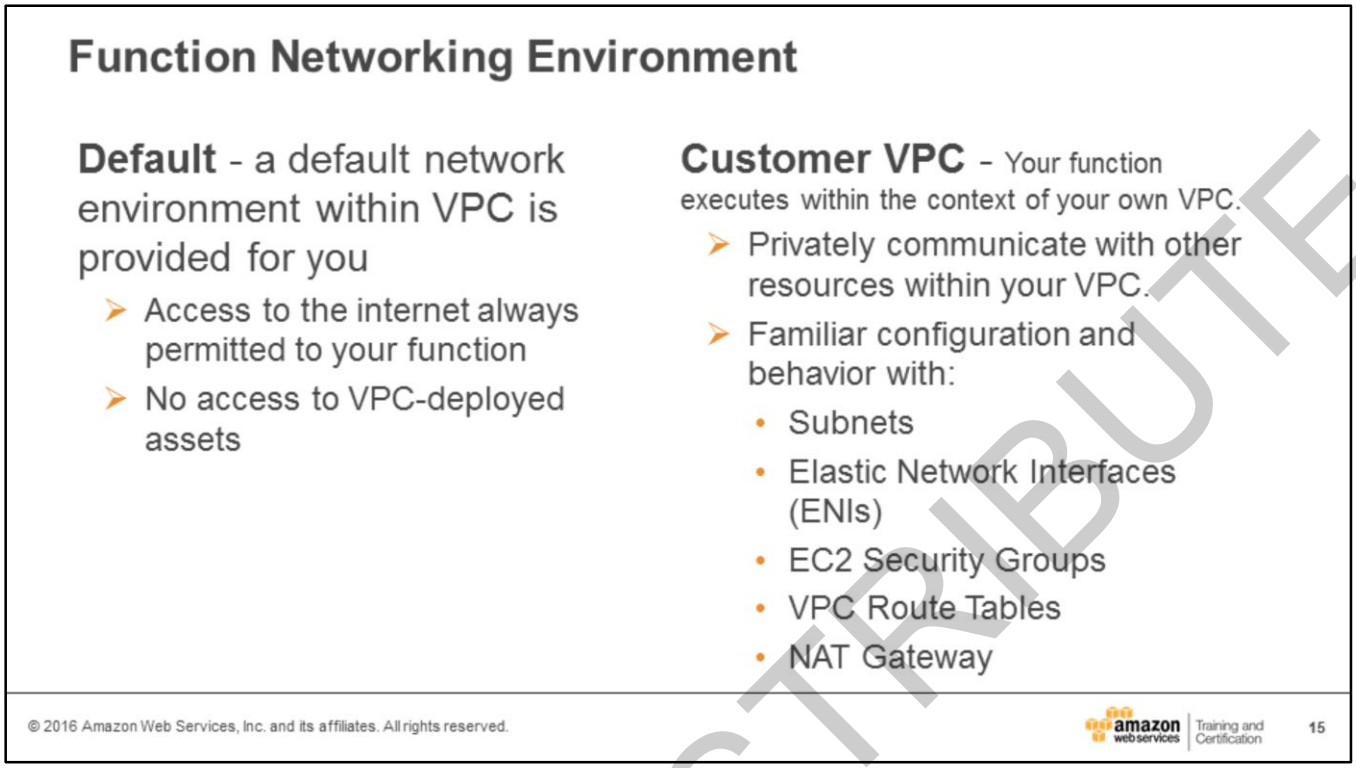
Next is the AWS Lambda Service itself. The Lambda service is responsible for taking the code that you've uploaded, provisioning it onto the Lambda infrastructure (that you don't manage), and providing you an API should you ever need to directly invoke your Lambda function.

The service is responsible for making sure that your code is executed for each and every event that you've configured as event sources for your function. Whether that be once a week when a new report lands inside an S3 bucket every time somebody speaks your application's Invocation Name to Amazon Alexa, or thousands of requests a second that your public API receives.

Finally, it provides you some out of the box operational capabilities such as monitoring your function in CloudWatch and a Logger object that will stream any log statements you make from your code using it to CloudWatch Logs.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



Last, is the function networking environment. This defines what your code's connectivity. We provide two broad options here.

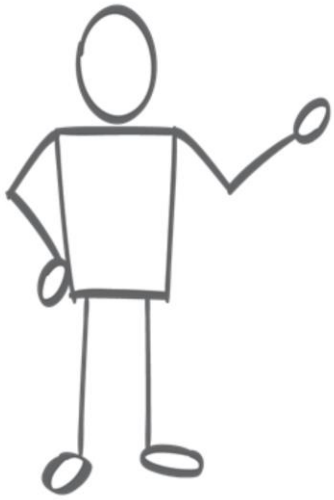
First is a default networking environment, where your function has access to the internet but no private connectivity to any resources running inside of your VPCs.

Second, your Lambda functions can be provisioned within your own customer-created Virtual Private Cloud. This option will be explained in Module 4.

If your function does not require any private network connectivity to VPC deployed resources and internet access is permissible, the default option is good. You can change this configuration for an existing Lambda function later, if needed.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

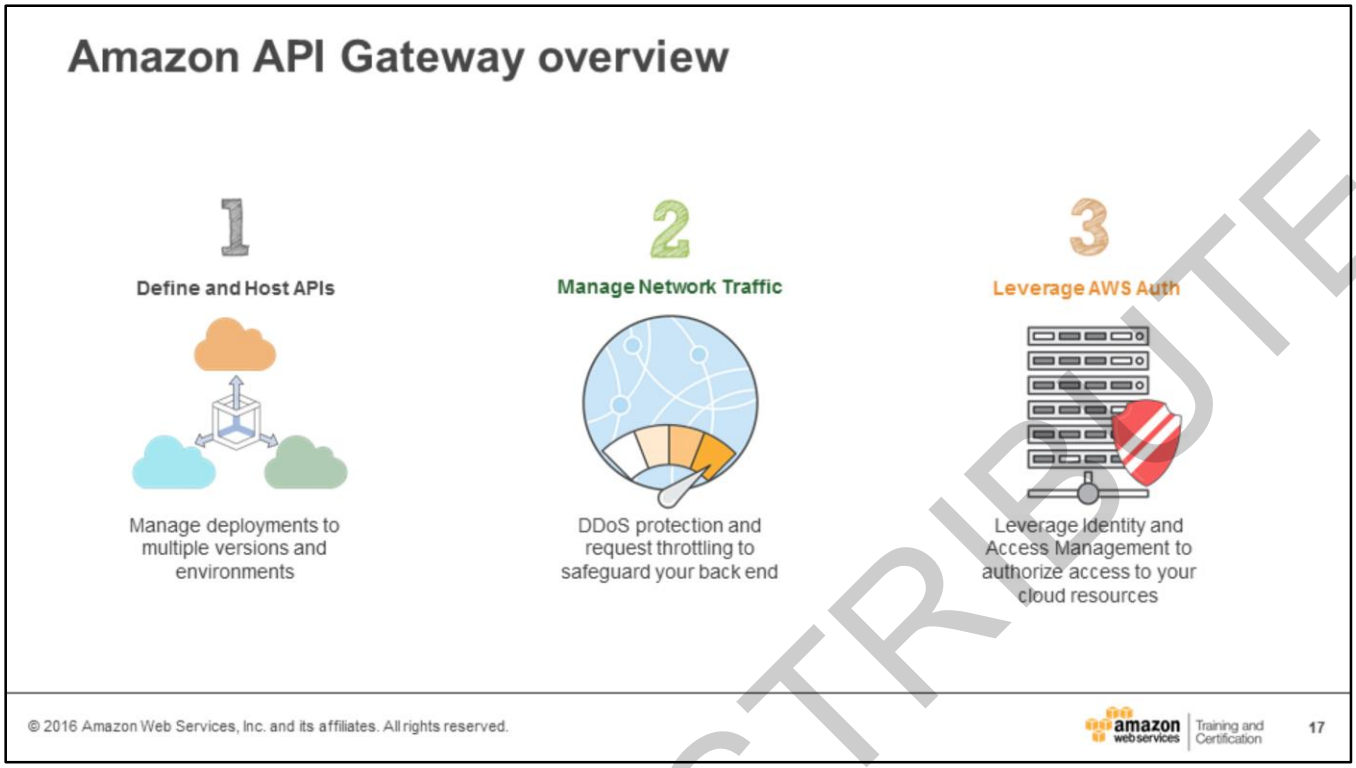


RESTful Web Services and API Gateway

© 2016 Amazon Web Services, Inc. and its affiliates. All rights reserved.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



Amazon API Gateway is a fully managed service for hosting HTTPS APIs on top of AWS.

You can create APIs:

- Support for standard HTTP methods
- Console, API, CLI support
- Swagger Import/Export
- Custom domains

Configure:

- Choose what your APIs integrate with:
- **AWS Lambda**
- AWS Service APIs
- Any other accessible web service
- Add an optional managed cache layer
- Stage variables for dynamic routing

Publish:

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

- Test new API versions pre-release
- Click-button or single API call deployment
- Create multiple versions and stages of your API
- Start letting developers integrate via mock responses

Maintain

- Managed scaling
- Usage-based pricing
- Ability to create and require API keys for developer integration
- Generate client SDK programmatically

Monitor:

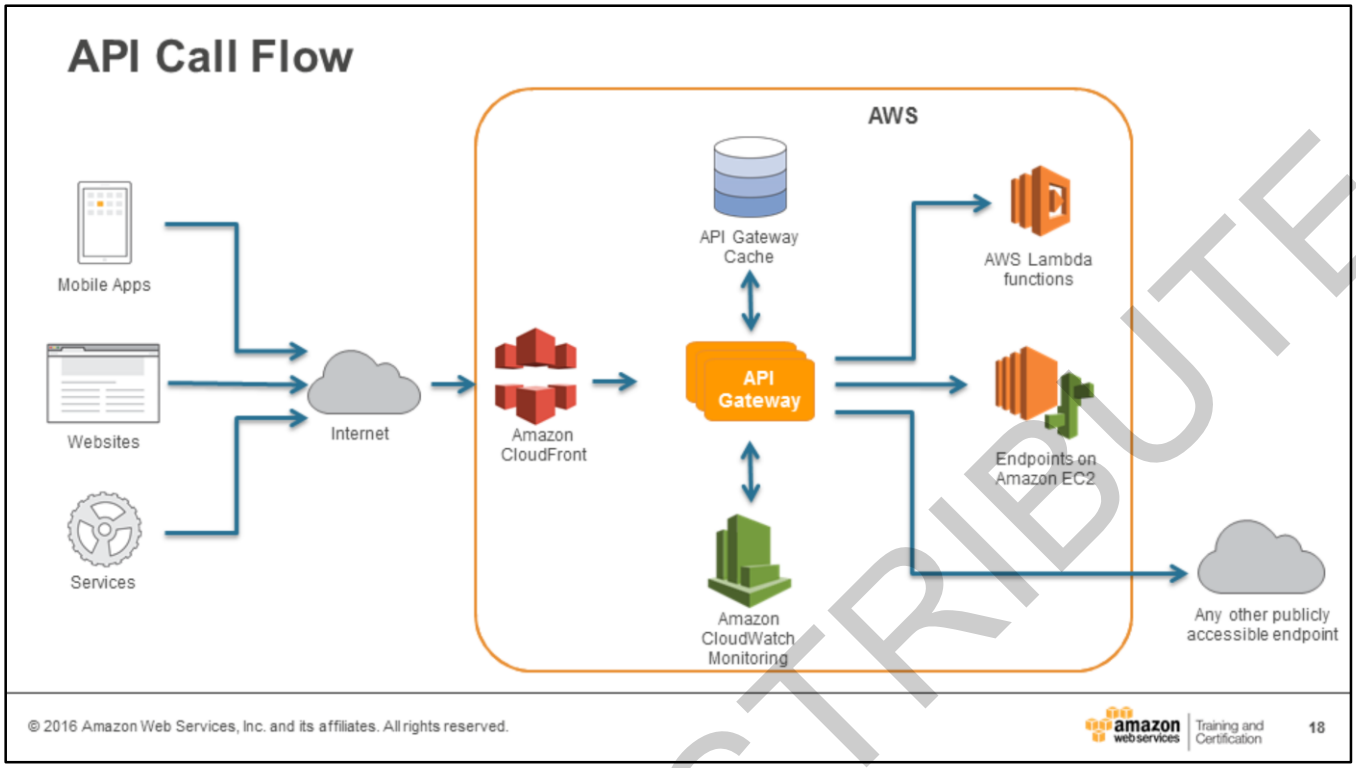
- Native CloudWatch metrics and CloudWatch Logs integration
- CloudTrail integration to track changes to your API

Secure:

- Native integration with IAM and AWS Sigv4 to authorize access to APIs
- Custom Authorization
- Mutual SSL with backend web services
- Integration with Amazon Cloudfront for DDoS protection
- Throttle and monitor requests to protect your backend

DO NOT DISTRIBUTE

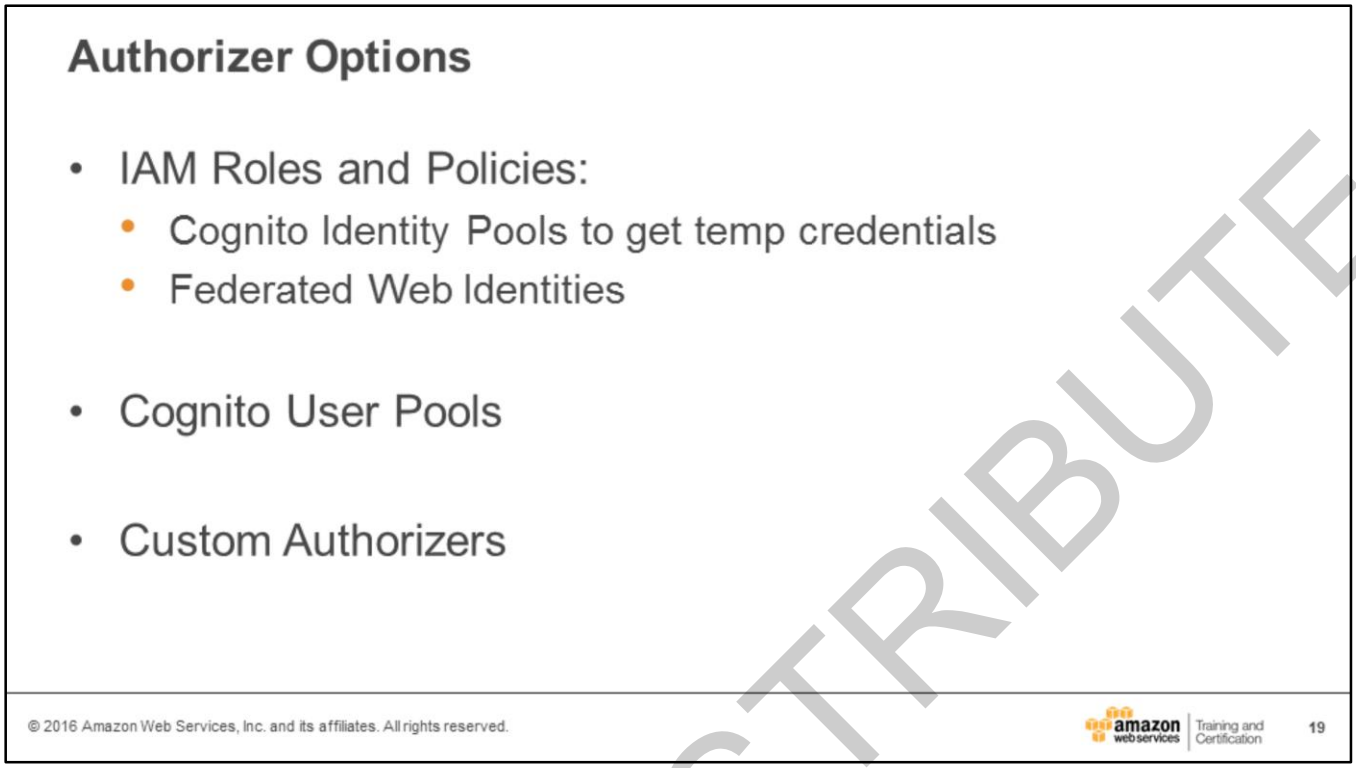
© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



- The first thing we want to look at is the standard flow of an API call, including all components in the system.
- A request comes from a client - this could be a mobile device, a web application or a backend service.
- The requests arrive at one of our CloudFront PoP locations, it's accepted and routed through to the API Gateway in the customer's region.
- The API Gateway receives the request and checks for records in the dedicated cache (if it is configured). If there are no cached records available, it will forward the request to backend for processing.
- Backend can be a Lambda function, a web service running on Amazon EC2, or any other publicly accessible web service.
- Once backend has processed the request, the API call metrics are logged in Amazon CloudWatch and the content is returned to the client.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



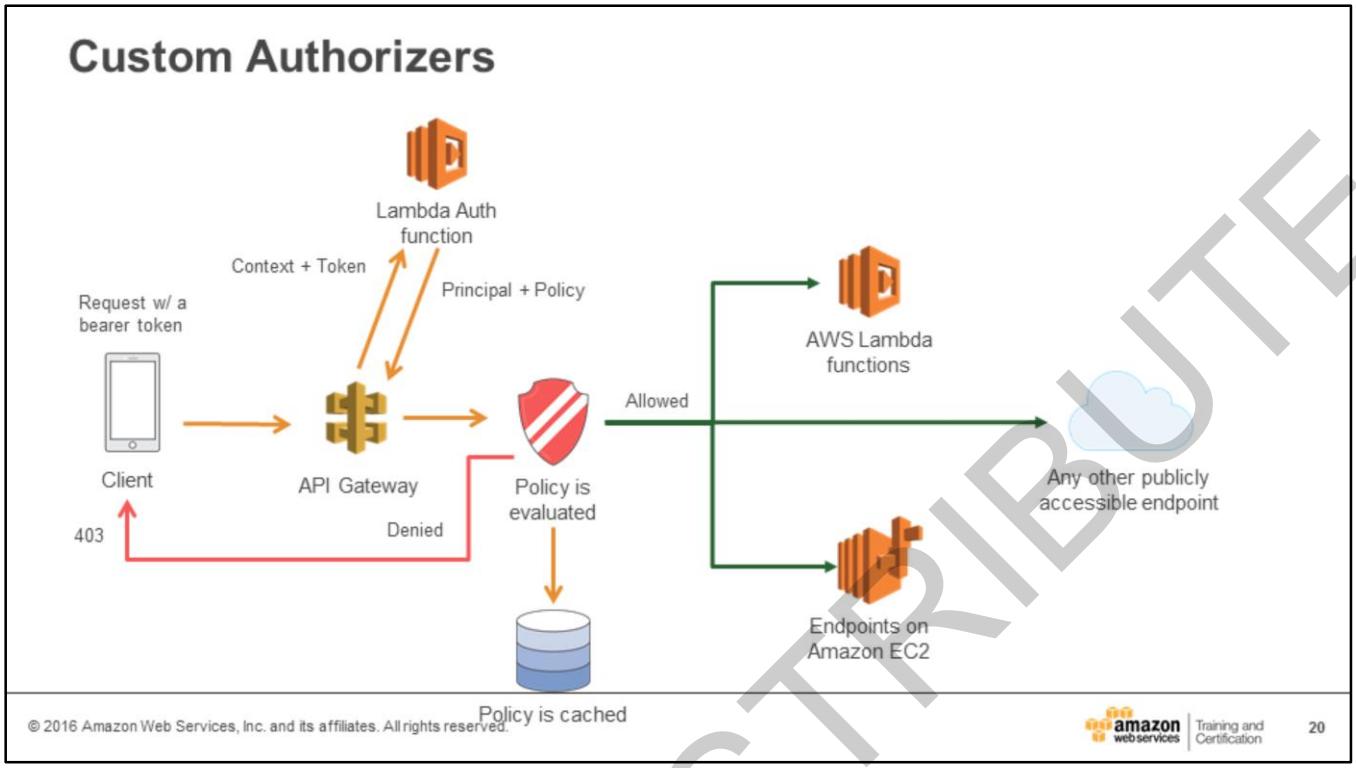
You can access API Gateway methods via Sigv4 by signing the request to the API call with valid IAM credentials. **It is** best practice to get temporary credentials that have permissions to call the API instead of baking in credentials to your client applications. There are different ways to gain temporary IAM credentials such as Cognito Identity **Pool,** which allow you to integrate with Amazon, Facebook, Google, **and so on,** as your Identity Provider. Additionally, you could use STS and Federated Web Identities to obtain temporary IAM credentials.

A user pool is integrated with an API as a method authorizer. When calling the methods with such an authorizer enabled, an API client includes in the request headers the user's identity token provisioned from the user pool. API Gateway then validates the token to ensure it belongs to the configured user pool and authenticates the caller before passing the request to **backend**.

Custom Authorizers are another option for authorizing calls to your API Gateway. Custom Authorizers allow you to integrate your APIs with OAuth providers or JWT verification tokens.

DO NOT DISTRIBUTE

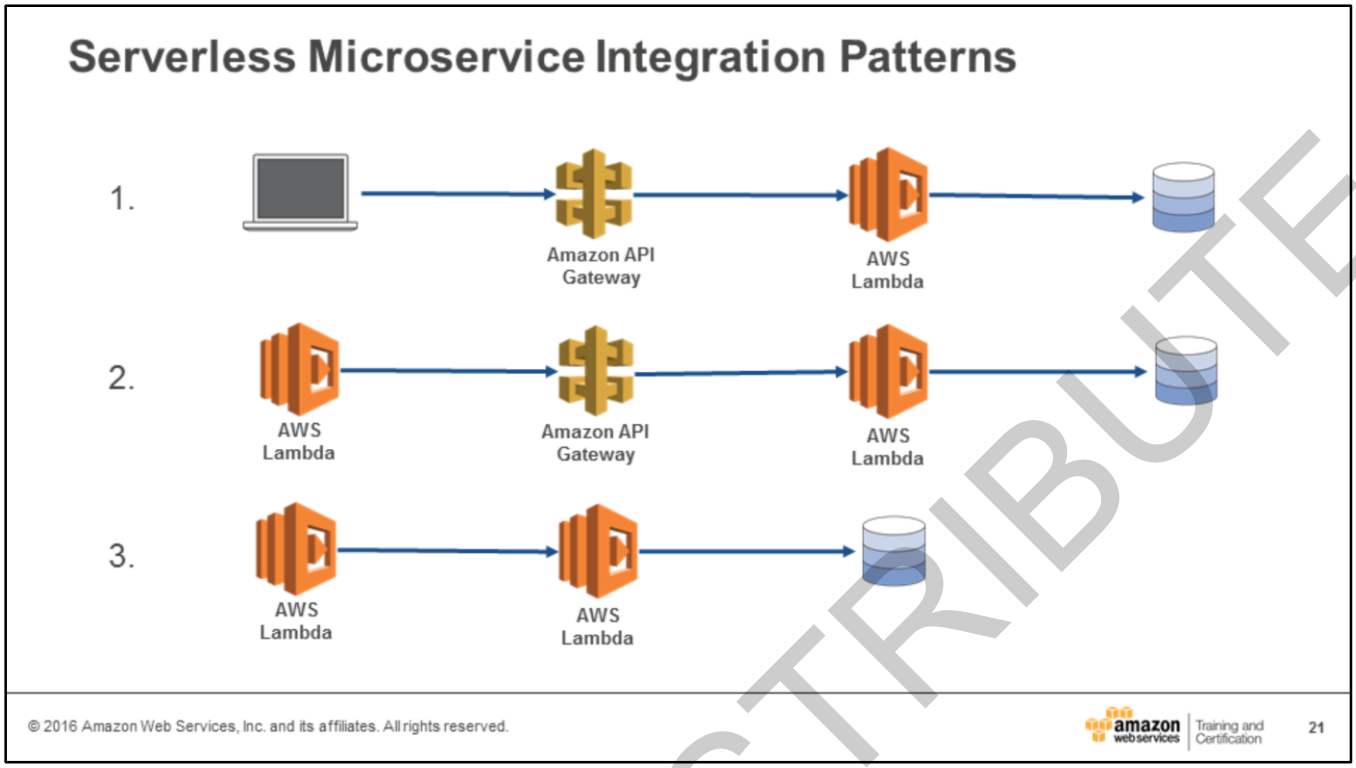
© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



With custom request authorizers, you can authorize your APIs using bearer token authorization strategies, such as OAuth using an AWS Lambda function. For each incoming request, API Gateway verifies whether a custom authorizer is configured, and if so, API Gateway calls the Lambda function with the authorization token. You can use Lambda to implement various authorization strategies, for example, JWT verification, OAuth provider callout. Custom authorizers must return AWS Identity and Access Management (IAM) policies. These policies are used to authorize the request. If the policy returned by the authorizer is valid, API Gateway caches the returned policy associated with the incoming token for up to 1 hour so that your Lambda function doesn't need to be invoked again.

DO NOT DISTRIBUTE

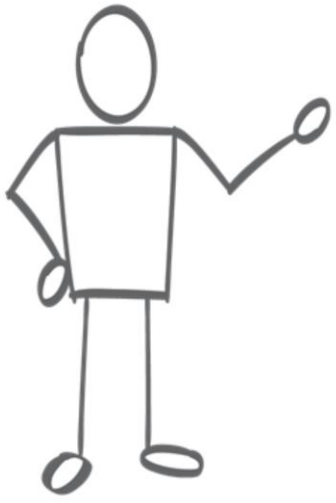
© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



When implementing serverless microservices you can use a combination of API Gateway and Lambda to implement small, loosely coupled web services that don't require you to manage any infrastructure. Keep in mind that for internally consumed services, you don't always need to front your Lambda functions with API Gateway. It's a perfectly valid design to have your functions invoke one another directly.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

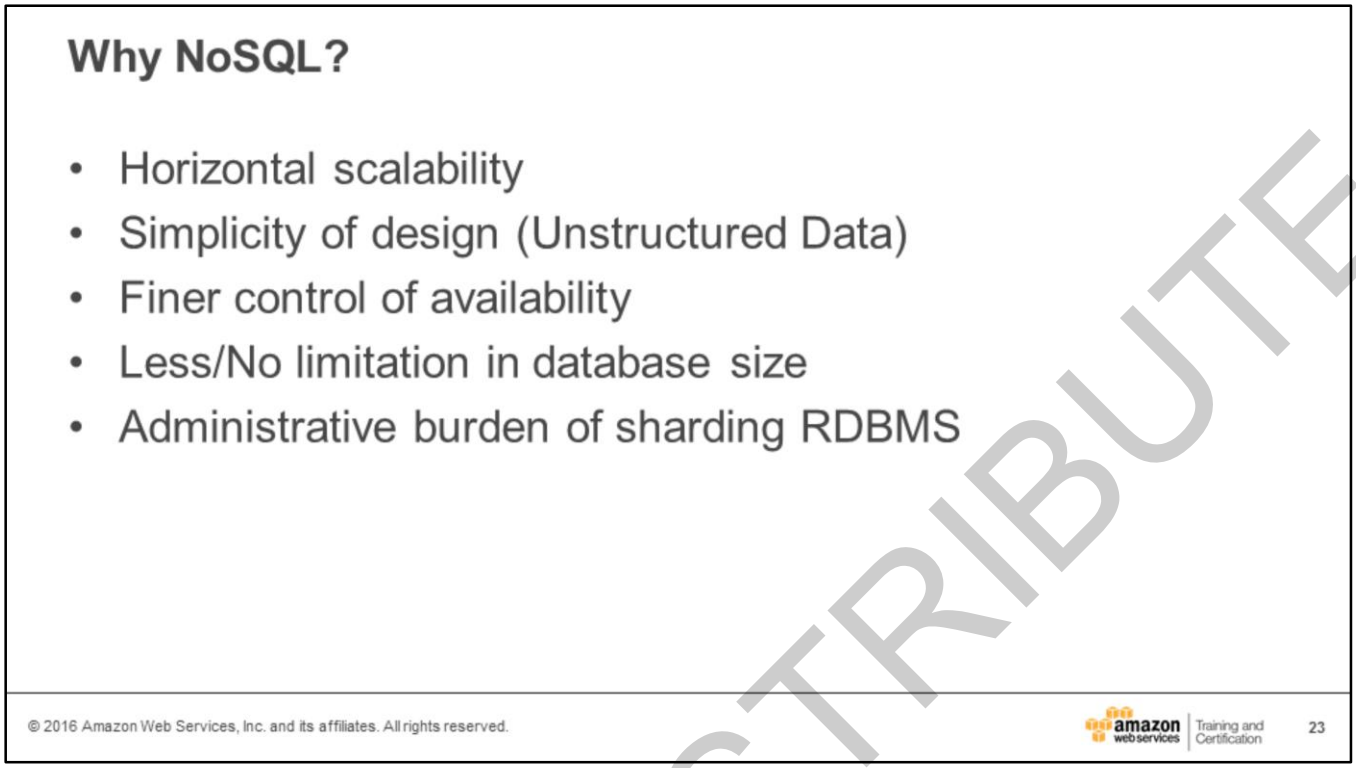


DynamoDB Introduction

© 2016 Amazon Web Services, Inc. and its affiliates. All rights reserved.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



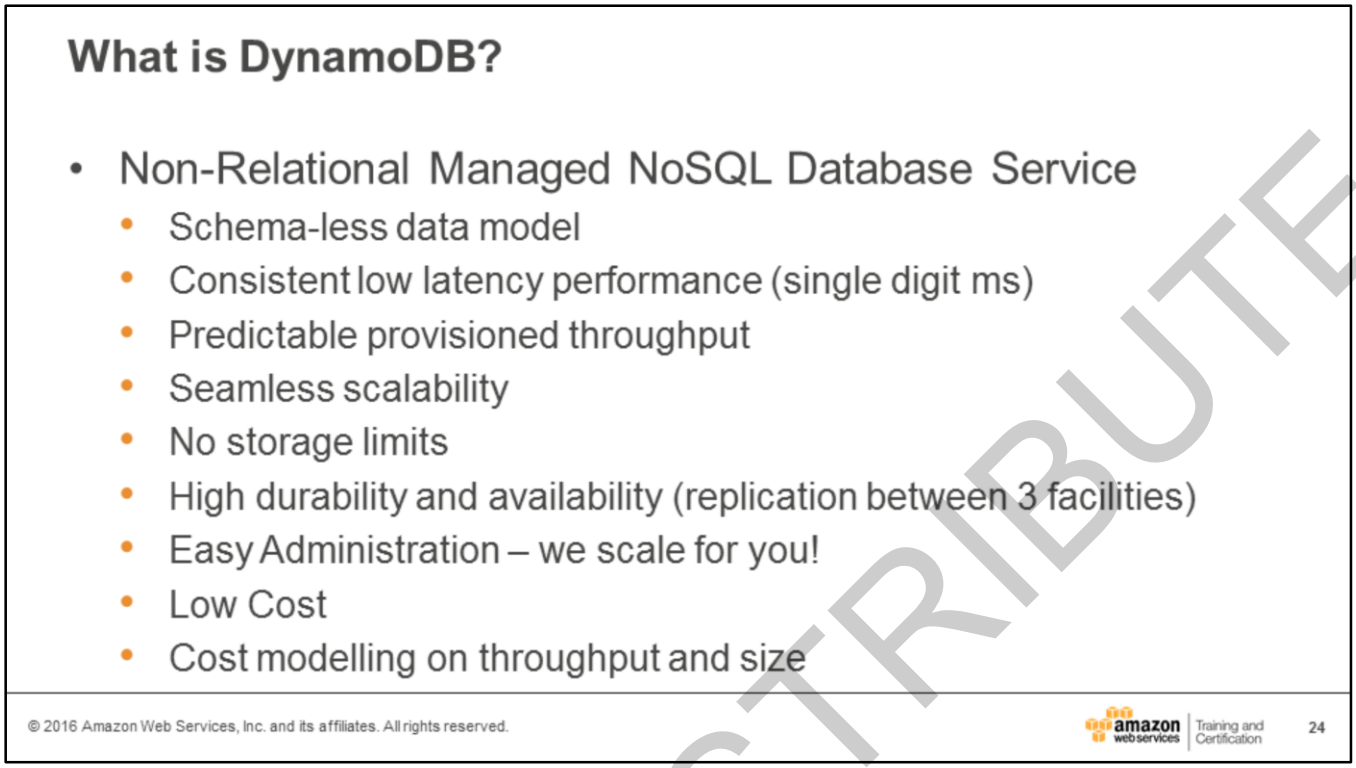
Why use NoSQL?

Traditional relational databases can be a choke point for applications as they start to scale. They are difficult to scale and require either adding read replicas or sharding your data which can be complex and difficult to maintain and operate. NoSQL databases allow for almost limitless scale and scale horizontally. This makes it easier for your data tier to keep up with the growth of your application.

Additionally, the ability to store unstructured data without a rigid schema defined means as your data changes overtime you are not stuck in an old data model.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



So, what is DynamoDB?

Amazon DynamoDB is a fully managed NoSQL database service that provides fast and predictable performance with seamless scalability. DynamoDB lets you offload the administrative burdens of operating and scaling a distributed database, so that you don't have to worry about hardware provisioning, setup and configuration, replication, software patching, or cluster scaling.

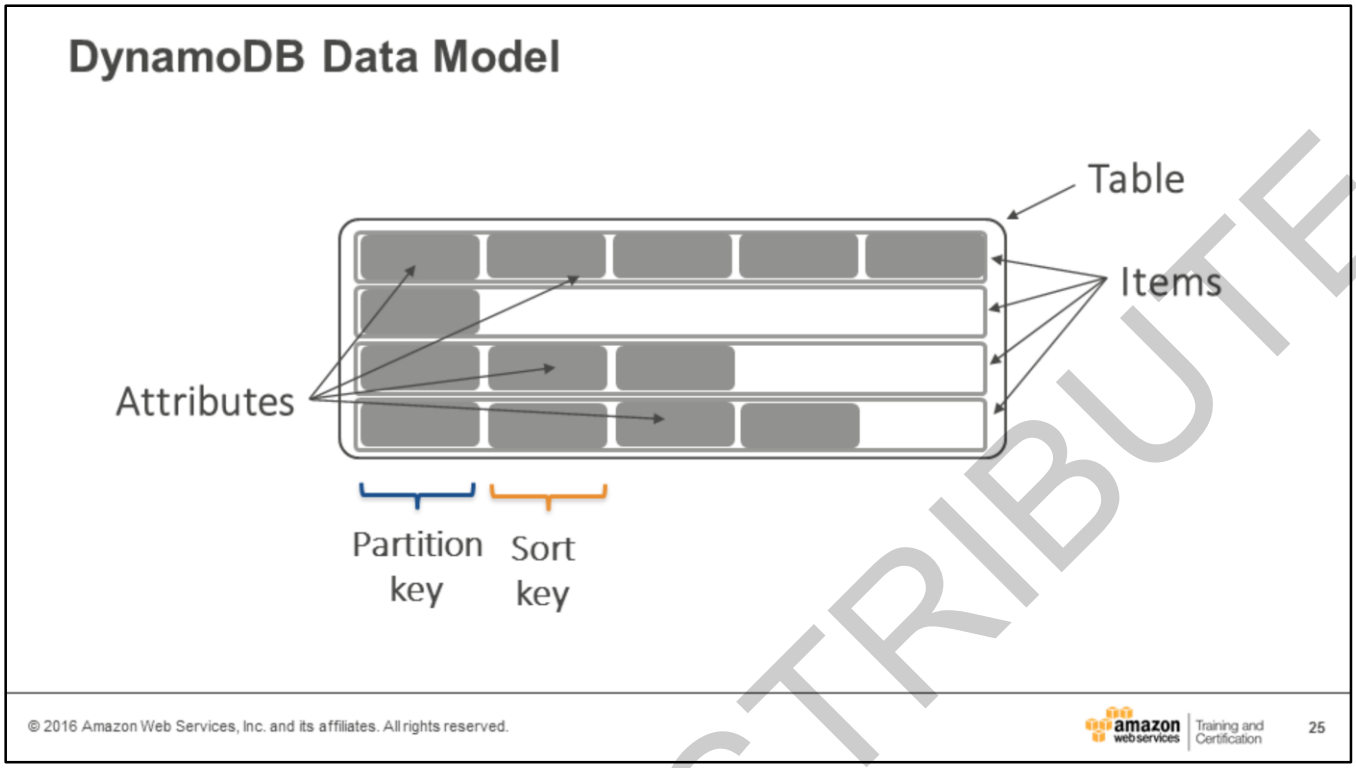
With DynamoDB, you can create database tables that can store and retrieve any amount of data, and serve any level of request traffic. You can scale up or scale down your tables' throughput capacity without downtime or performance degradation, and use the AWS Management Console to monitor resource utilization and performance metrics.

You specify the IOPS you need for read and write capacity and DynamoDB scales to meet those needs assuming you have a good key distribution for your queries. There is no limit to how much data you can store in a table and the data is automatically replicated to 3 different facilities to give you high availability and durability. All of this is handled for you with no need for you to manage any infrastructure.

Additionally, you can easily model your costs based on the amount of data you expect and your needed throughput.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



In DynamoDB, tables, items, and attributes are the core components that you work with. A table is a collection of items and each item is a collection of attributes. Items are identified by a unique key that consists of a partition key and a sort key. Partition keys are required while sort keys are optional.

Notice that items within a table do not have to share the same set of attributes although in practice you should generally use a consistent structure for items within a given table. There is no schema defined for a table beyond the key structure. Attribute values can be strings, numbers, binary data, boolean values, maps (similar to JSON objects), lists, or sets.

DO NOT DISTRIBUTE

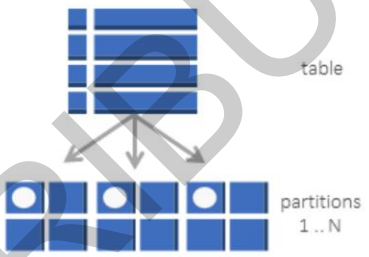
© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

Scalability

- No limit in terms of throughput (reads/writes per second)
- No limit in terms of storage
- Automatically partitions data by the hash key

Auto-Partitioning occurs when:

- Data set growth
- Provisioned capacity increase



© 2016 Amazon Web Services, Inc. and its affiliates. All rights reserved.
amazon web services | Training and Certification 26

DynamoDB stores data in partitions. A *partition* is an allocation of storage for a table, backed by solid-state drives (SSDs) and automatically replicated across multiple Availability Zones within an AWS Region. Partition management is handled entirely by DynamoDB—customers never need to manage partitions themselves.

DynamoDB will allocate additional partitions to a table in the following situations:

- If you increase the table's provisioned throughput settings beyond what the existing partitions can support.

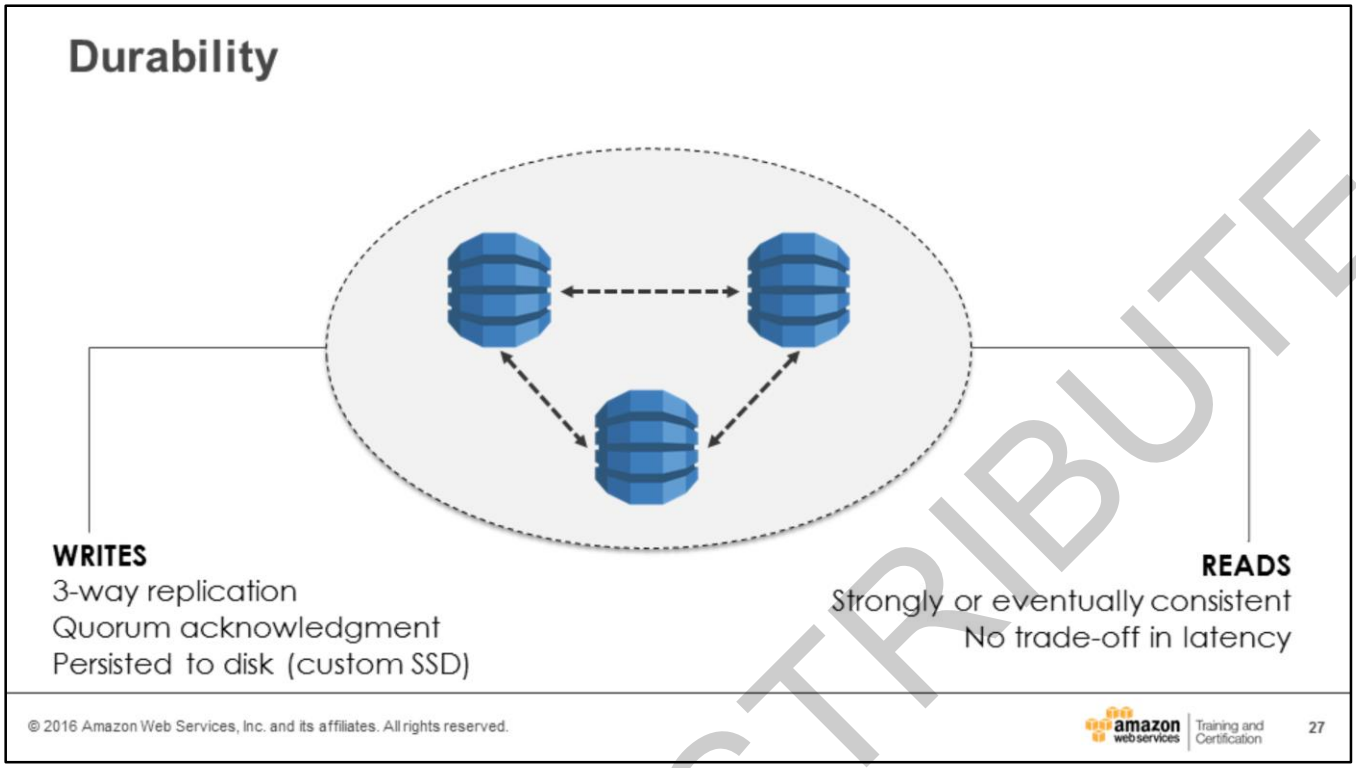
- If an existing partition fills to capacity and more storage space is required.

Partition management occurs automatically in the background and is transparent to your applications. Your table remains available throughout and fully supports your provisioned throughput requirements.

DynamoDB does not have limits in terms of throughput (IOPS) you can provision or on the amount of data you are storing in a table. In the backend it is automatically partitioning the data by the hash key. This makes it imperative to have a good hash key scheme so as not to get hot partitions. To read more about partitions, visit: http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.Partitions.html

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



DynamoDB automatically spreads the data and traffic for your tables over a sufficient number of servers to handle your throughput and storage requirements, while maintaining consistent and fast performance. All of your data is stored on solid-state drives (SSDs) and are automatically replicated across multiple Availability Zones in an AWS region, providing built-in high availability and data durability.

To support varied application requirements, DynamoDB supports both *eventually consistent* and *strongly consistent* reads.

Eventually Consistent Reads

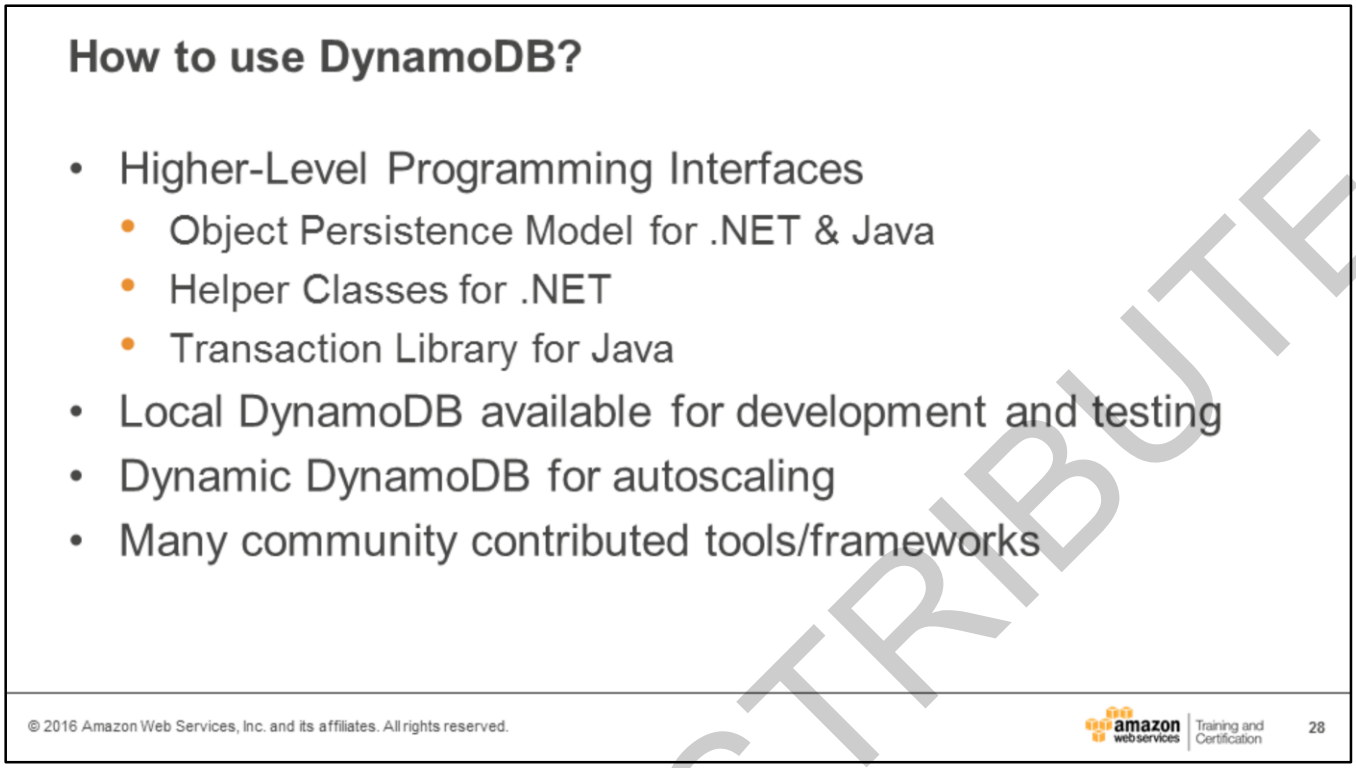
When you read data from a DynamoDB table, the response might not reflect the results of a recently completed write operation. The response might include some stale data. However, if you repeat your read request after a short time, the response should return the latest data.

Strongly Consistent Reads

When you request a strongly consistent read, DynamoDB returns a response with the most up-to-date data, reflecting the updates from all prior write operations that were successful. Note that a strongly consistent read might not be available in the case of a network delay or outage.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



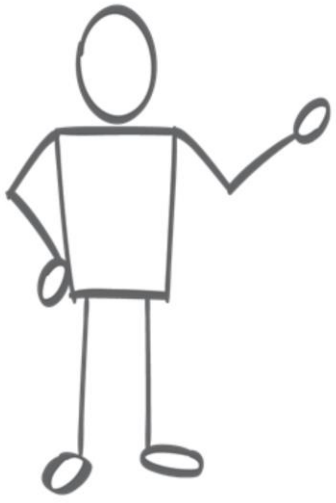
Interactions with DynamoDB is easiest through higher level programming interfaces that interact with the DynamoDB APIs such as those documented here: http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HigherLevelInterfaces.html. There is also AWS SDK support for interacting with DynamoDB via the APIs.

There is a local version you can run for testing your applications that interact with DynamoDB locally. Dynamic DynamoDB is a tool provided out on GitHub that helps you scale your DDB throughput according to demand on your table. Take a look here for more information: https://aws.amazon.com/blogs/aws/auto-scale-dynamodb-with-dynamic-dynamodb/

Some additional example tools are noted here: http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Tools.html

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

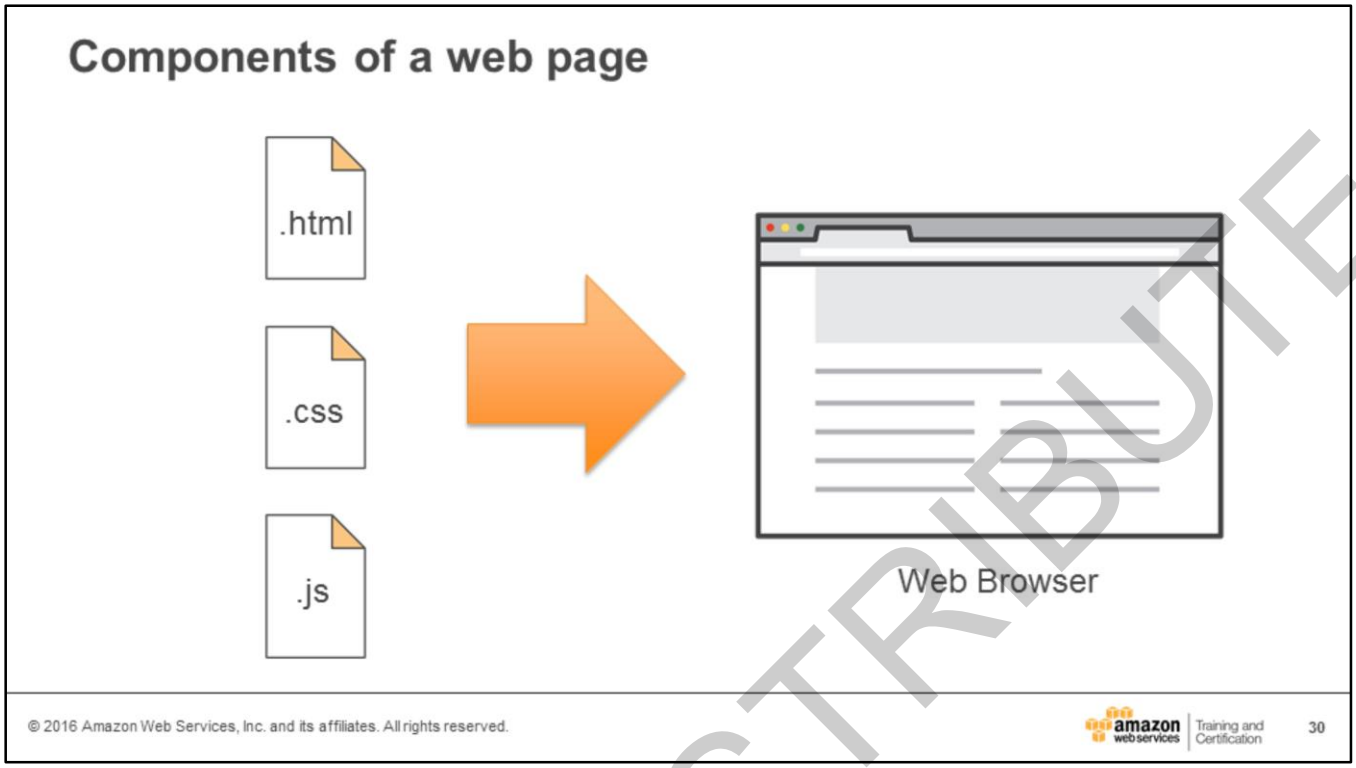


Browser-based Application Primer

© 2016 Amazon Web Services, Inc. and its affiliates. All rights reserved.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

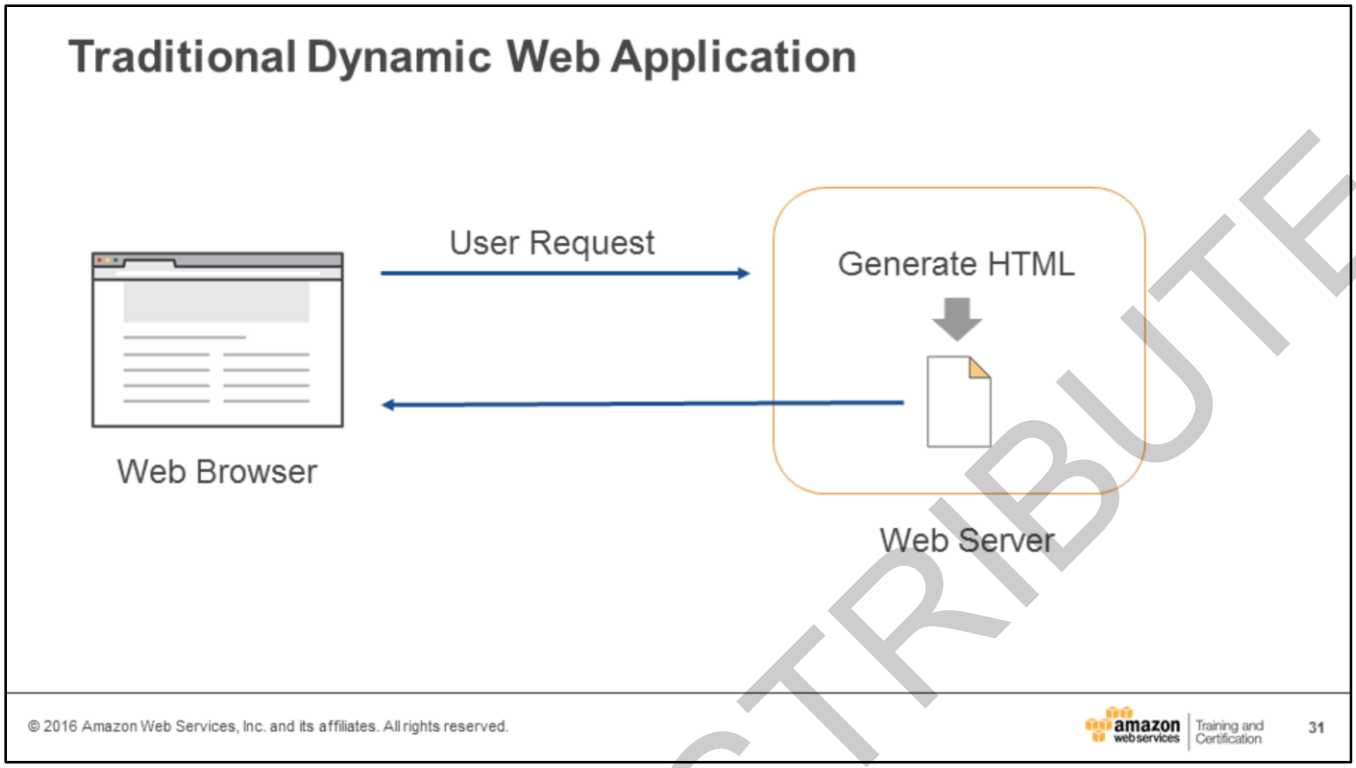


Static web pages consist of an HTML document that references Cascading Style Sheets (CSS) and JavaScript files. The HTML defines the elements and structure of the page, the CSS defines styles such as fonts, colors, and element dimensions and JavaScript are used to respond to events on the page and update the elements and styles dynamically.

All the files that are loaded by the browser are static. You can use JavaScript to modify the state of the page after it has been loaded, but the HTML, CSS, and JavaScript files do not change.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



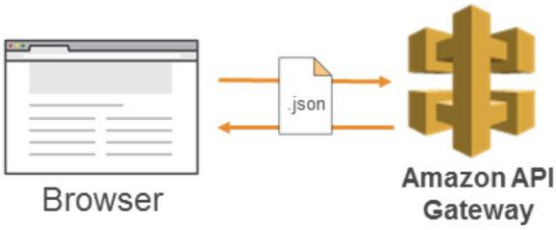
Static web pages consist of an HTML document that references cascading style sheets (CSS) and JavaScript files. The HTML defines the elements and structure of the page, the CSS defines styles such as fonts, colors, and element dimensions, and JavaScript is used to respond to events on the page and update the elements and styles dynamically.

All of the files that are loaded by the browser are themselves static. You can use JavaScript to modify the state of the page after it has been loaded, but the HTML, CSS and JavaScript files themselves do not change.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

Making Static Pages Dynamic



Static page structure defined by HTML.

JavaScript event listeners use AJAX to call external REST services and update the HTML document.

```
$('#my-button').click(function() {
    var oReq = new XMLHttpRequest();
    oReq.addEventListener("load", reqListener);
    oReq.open("GET", "http://www.example.org/myMethod");
    oReq.send();
});
```

© 2016 Amazon Web Services, Inc. and its affiliates. All rights reserved.

Amazon S3 static web hosting will return the same static content for every request to a given resource. In order to enable users to interact with the page and retrieve data from the cloud, you can use a JavaScript executed in the user's browser to respond to events, for example, a button click, and retrieve data from a RESTful web service. This technique uses the XMLHttpRequet object to execute the remote web service calls, but it doesn't require that your service or application use XML to encode the data. In fact, most modern applications use JSON.

When you use Amazon API Gateway to host your web service API, you don't have to write a code against the XMLHttpRequest object directly. Amazon API Gatway will generate a JavaScript SDK for your API that can be used within your application and it simplifies the process of making API calls from the browser.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

Static Website Hosting on S3

- Specify an index document (i.e. index.html)
- Specify an error document
- Objects publicly readable
- Supports redirects
  - All Requests
  - Conditional


```
<RoutingRules>
    <RoutingRule>
    <Condition>
      <KeyPrefixEquals>docs/</KeyPrefixEquals>
    </Condition>
    <Redirect>
      <ReplaceKeyPrefixWith>documents/</ReplaceKeyPrefixWith>
    </Redirect>
    </RoutingRule>
  </RoutingRules>
```

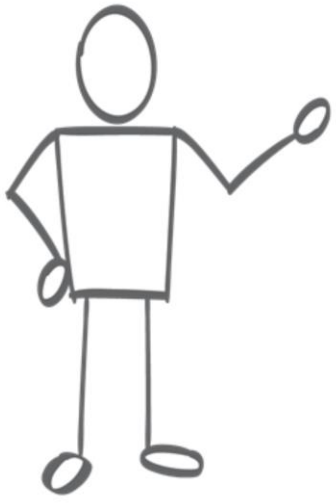
© 2016 Amazon Web Services, Inc. and its affiliates. All rights reserved.

You can serve static web content from an Amazon S3 bucket by enabling website hosting in the bucket configuration. Amazon S3 website hosting enables you to specify redirection rules as well as default index and error pages, but you cannot run any server side scripts.

When configuring a bucket for static website hosting make sure to add a policy for all objects to be publicly readable.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

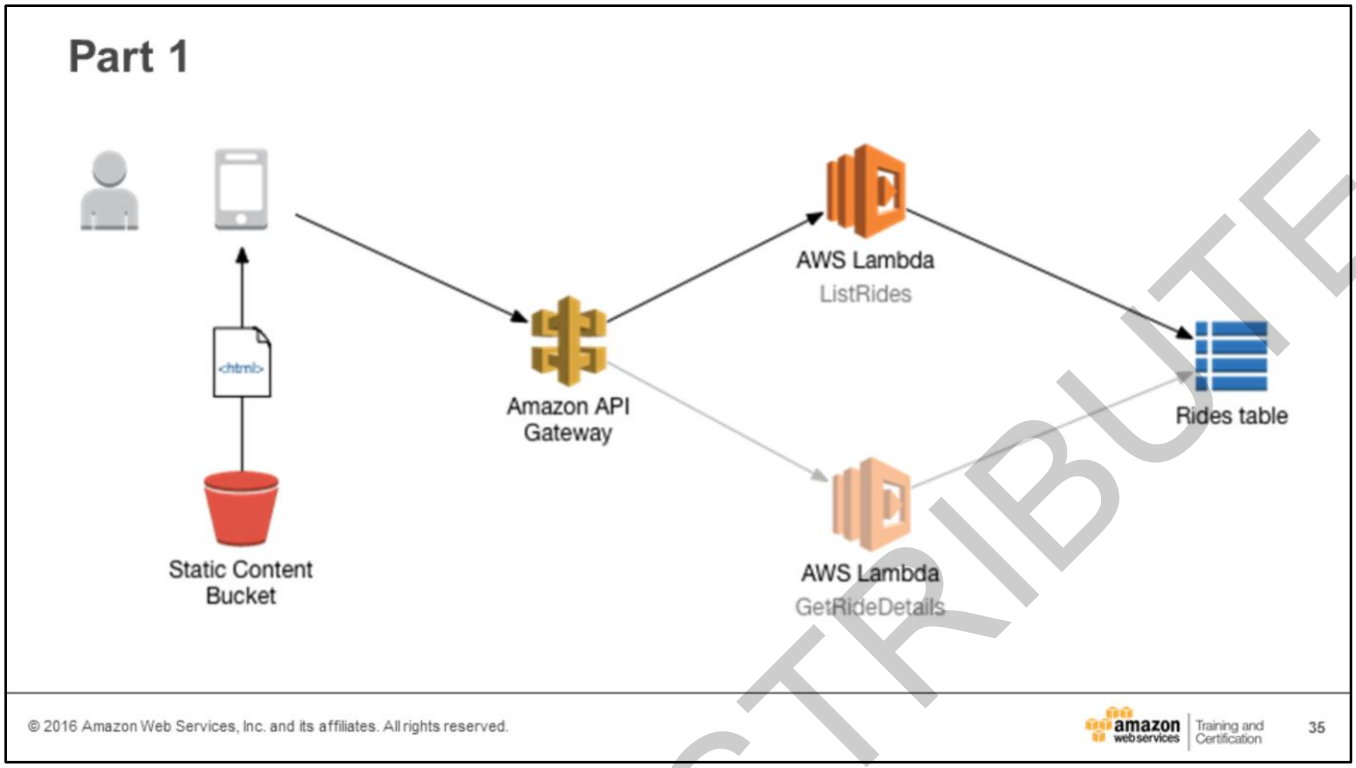


Lab Architecture Overview

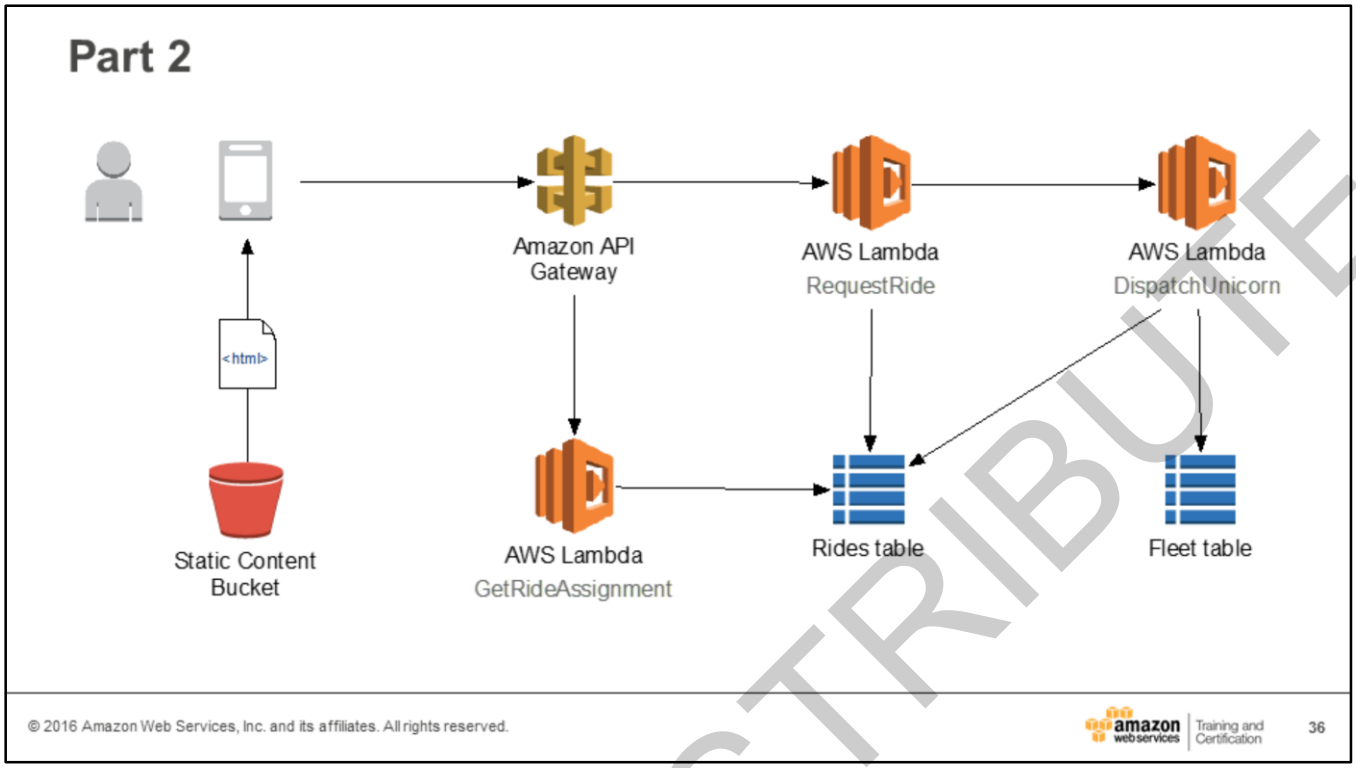
© 2016 Amazon Web Services, Inc. and its affiliates. All rights reserved.

DO NOT DISTRIBUTE

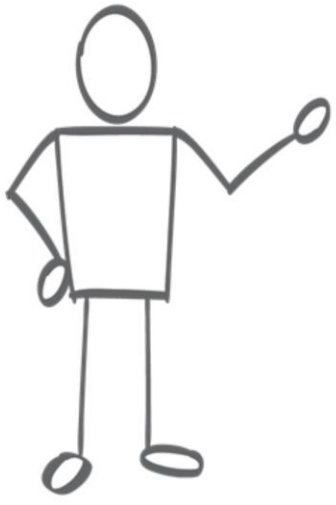
© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



DO NOT DISTRIBUTE



DO NOT DISTRIBUTE

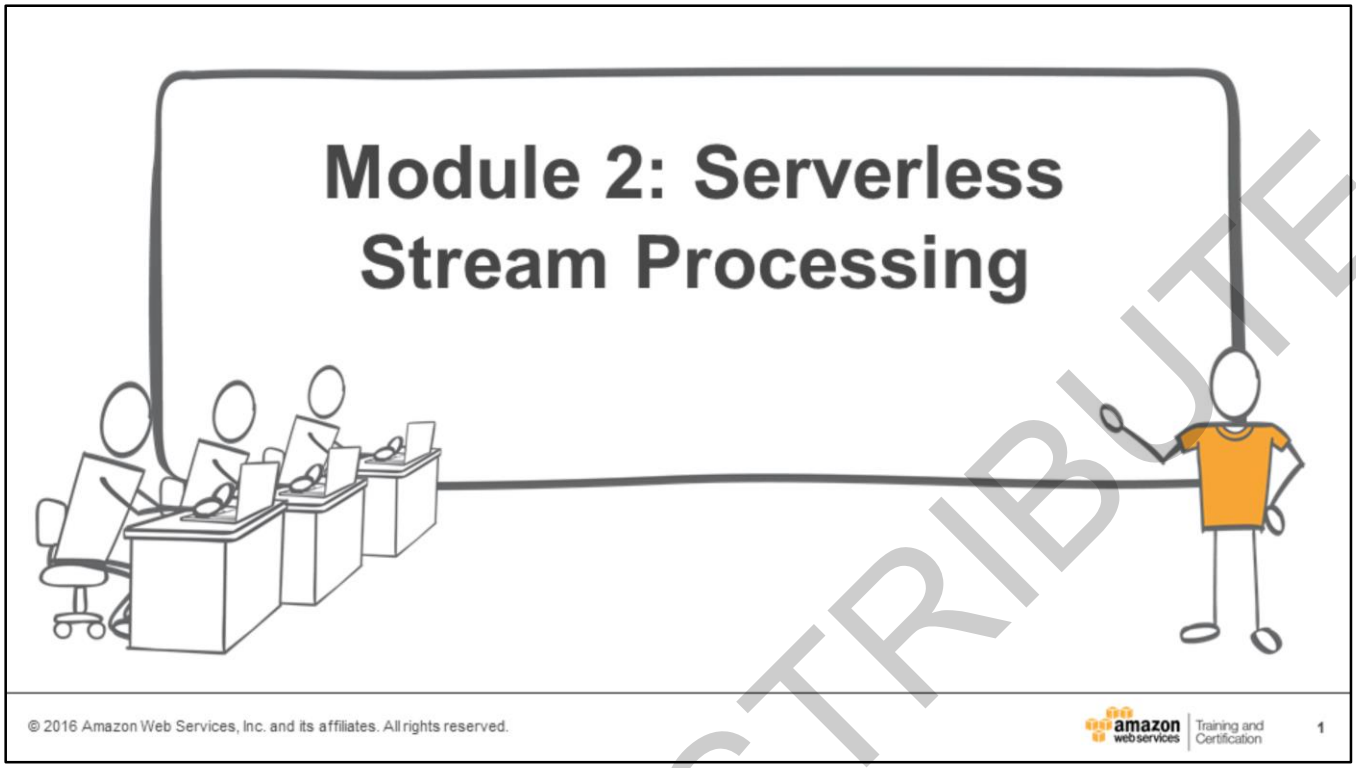


Lab 1

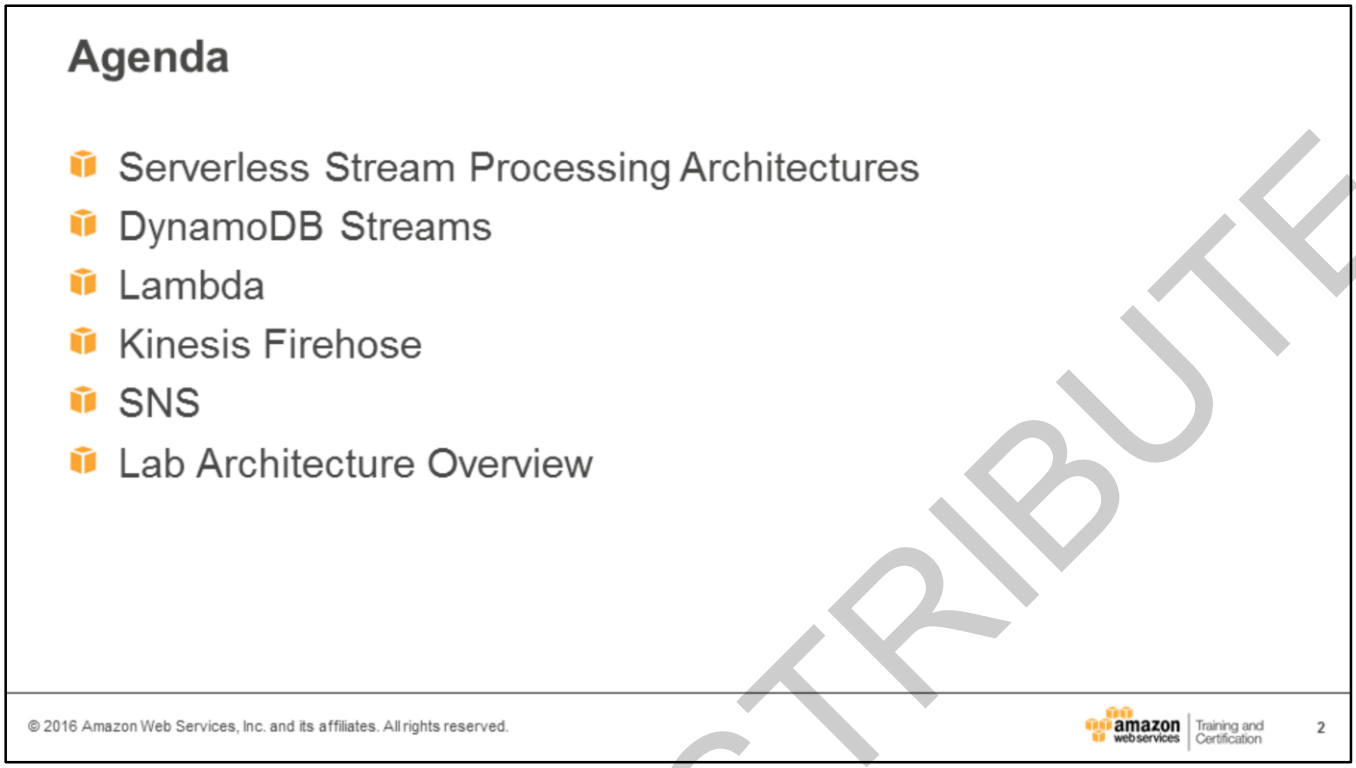
© 2016 Amazon Web Services, Inc. and its affiliates. All rights reserved.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

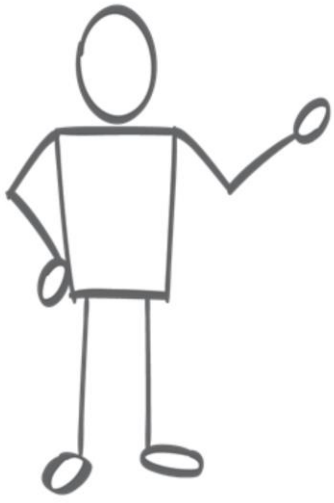


DO NOT DISTRIBUTE



This module will cover how to stream and process streams of data without having to manage servers.

DO NOT DISTRIBUTE



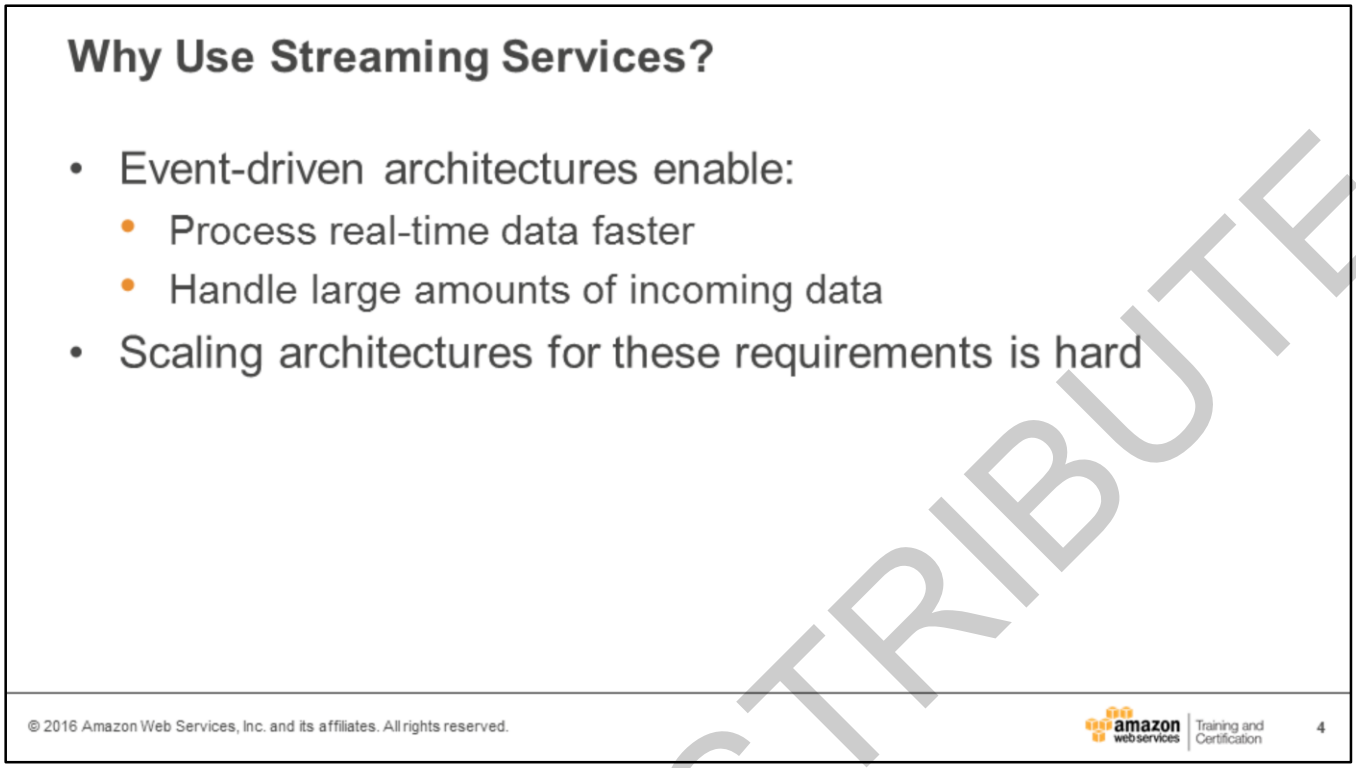
Serverless Stream Processing Architectures

© 2016 Amazon Web Services, Inc. and its affiliates. All rights reserved.

amazon web services | Training and Certification

3

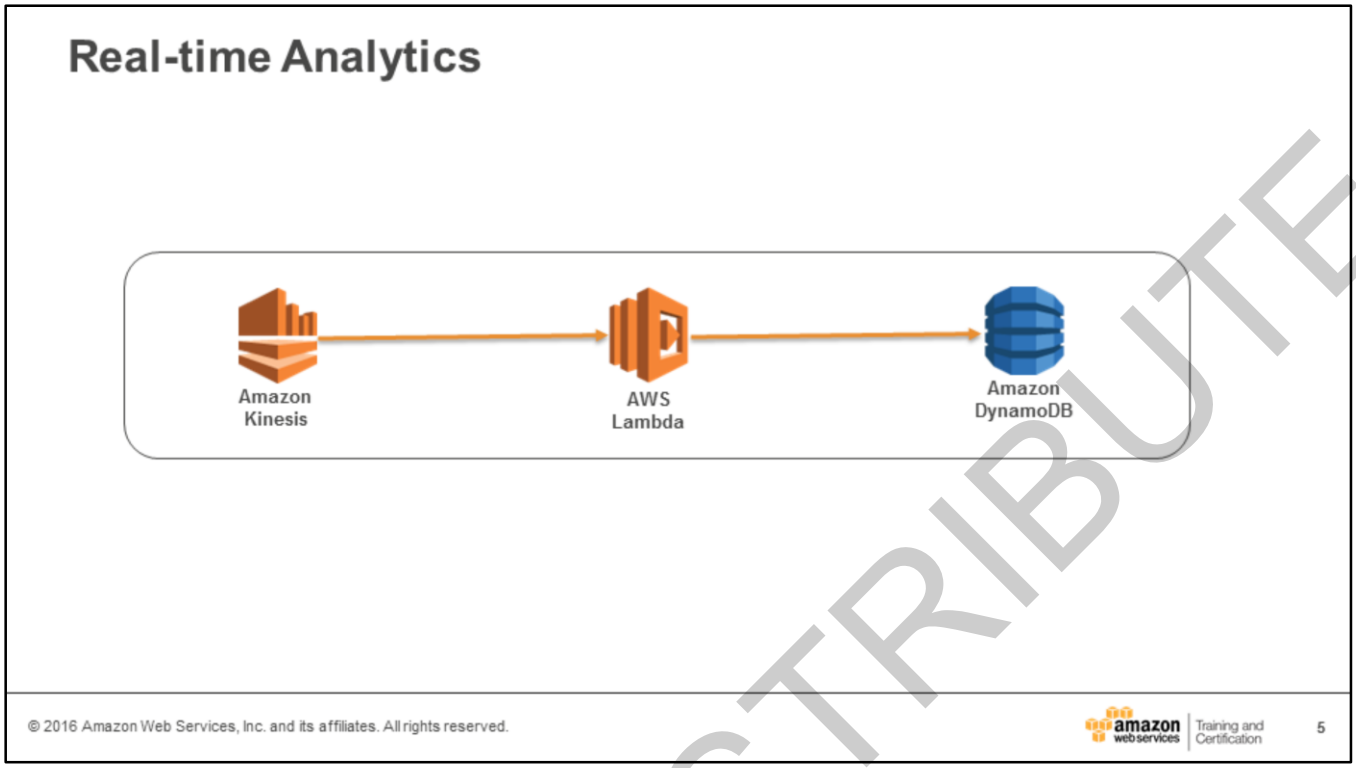
DO NOT DISTRIBUTE



There are many benefits to using streaming services within your architecture, but the main benefit is that they enable you to move to an event-driven architecture. Moving to an event-driven architecture allows you to process the changes in your data more quickly. It also enables you to ingest large amounts of changing data without you having to worry about scaling the infrastructure to do so.

DO NOT DISTRIBUTE

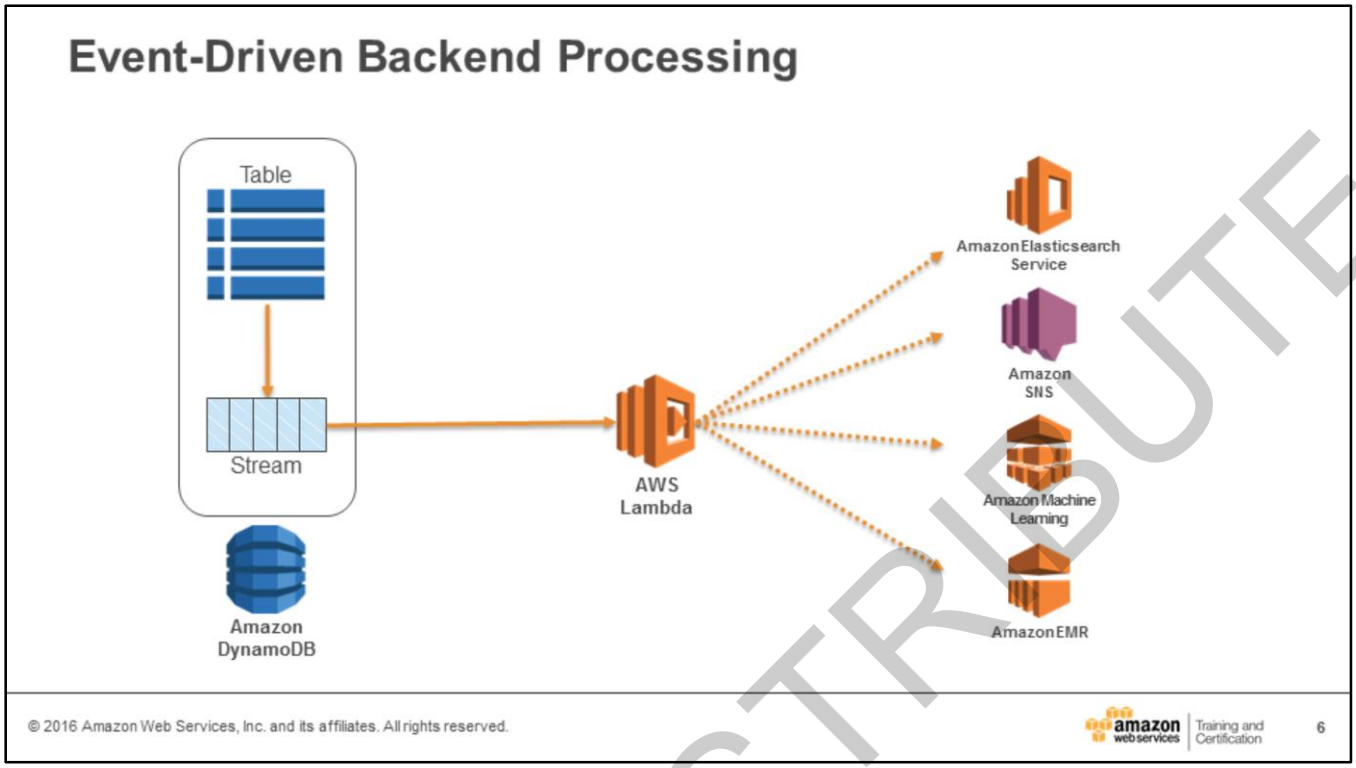
© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



In this example architecture, you can use Amazon Kinesis as your ingestion point, which is a service that accepts real-time streaming data. You can process data in the Kinesis Stream using AWS Lambda. AWS Lambda provides serverless compute resources. You simply upload your code functions and the function is executed by an event. In this case, the event is data being put into the Kinesis Stream. Your Lambda function can do whatever type processing you'd like. Here, we are showing the Lambda function saving the data to DynamoDB, our managed NoSQL database. You can then build a real-time visualization dashboard to display the data in the DynamoDB table.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



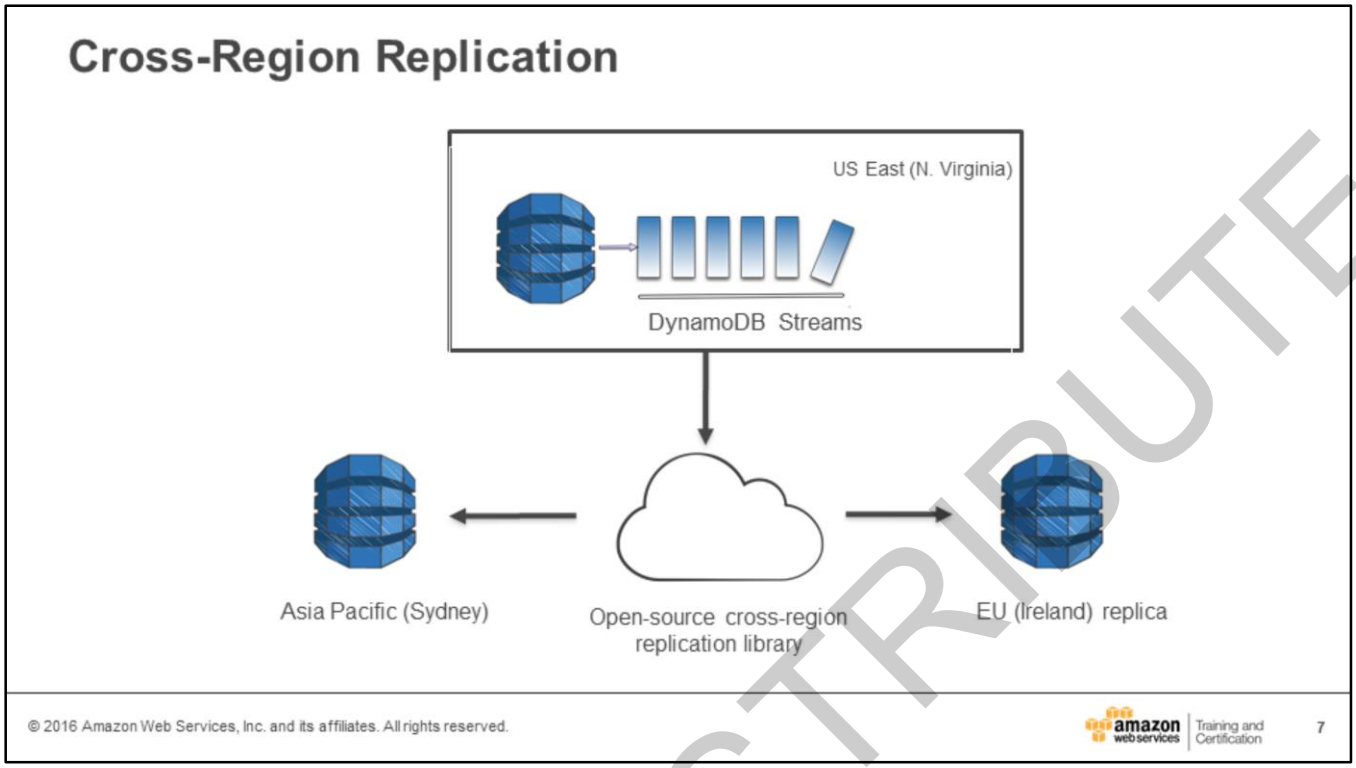
Another type of streaming architecture is processing changes to your backend system.

DynamoDB, our managed NoSQL database service, has the ability to support a DynamoDB stream. We will get into the features of these streams later, but essentially, you can configure it so that when updates are made to your DynamoDB table's data, those updates are put **in** a DynamoDB Stream.

You can then process the data in that stream using a service such as Lambda, our serverless compute service and do any number of actions from there such as populating a search index, sending push notifications, building a machine learning model, or ingesting the data into a Hadoop framework.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

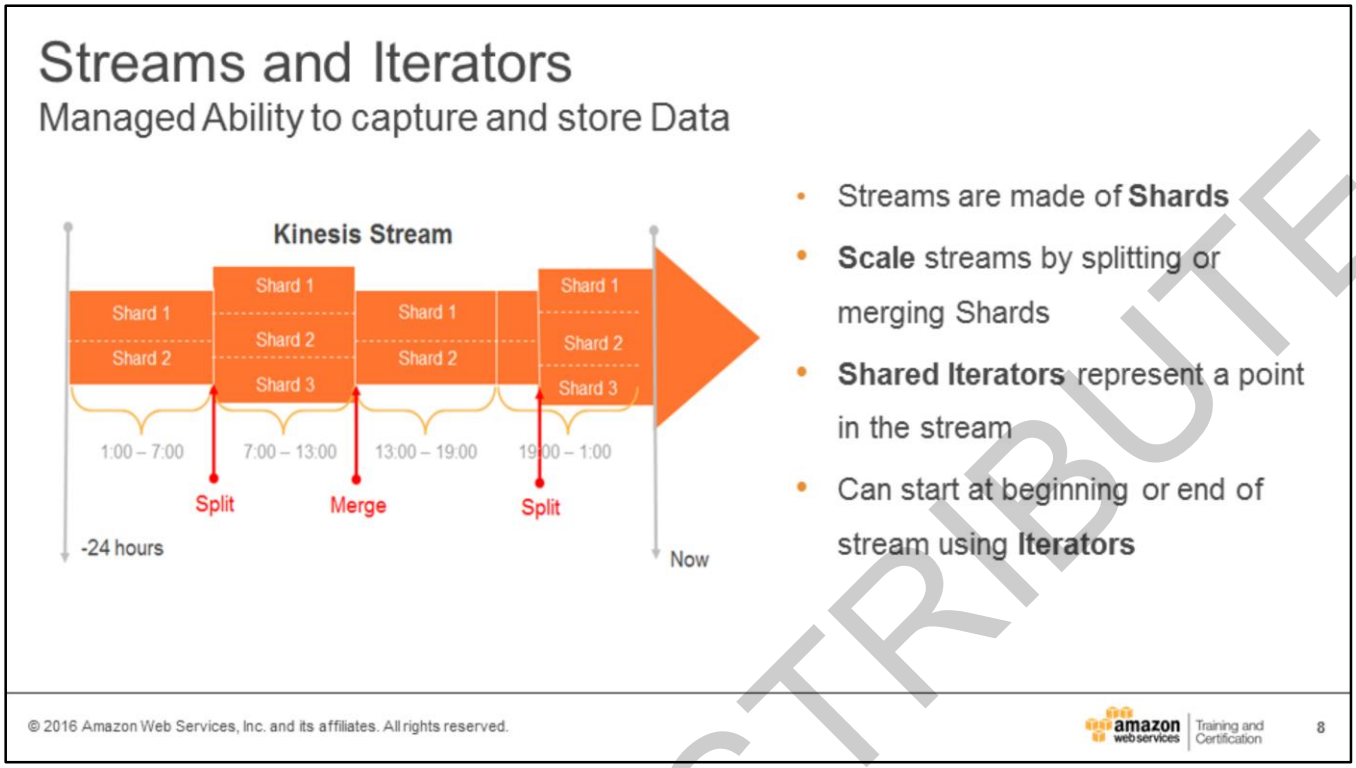


DynamoDB Streams can also be used to do cross-region replication of the data in your DynamoDB table. We have published a library to help you with this implementation here: https://github.com/awslabs/dynamodb-cross-region-library

By using this library you can replicate your DynamoDB table data in other regions enabling many sophisticated architecture designs.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



Streams, such as Kinesis and DynamoDB Streams, are made up of *shards*.

A *shard* is a uniquely identified group of data records in a stream. A stream is composed of one or more shards, each of which provides a fixed unit of capacity. If your data rate increases, add more shards to increase the size of your stream. Similarly, you can remove shards if the data rate decreases.

You can retrieve records from the stream on a per-shard basis. For each shard and for each batch of records that you retrieve from that shard, you need to obtain a *shard iterator*. The shard iterator is used to specify the shard from which records are to be retrieved. The type associated with the shard iterator determines the point in the shard that you are retrieving. In other words, you can retrieve data from the beginning of the stream or the end of the stream. You can specify which point in the shard you want to retrieve records from by specifying the type in the shard iterator.

There are different **Shard-Iterator** types you can specify. Each determines a different point at which you pull data off the shard. The following are valid options for Amazon Kinesis shard iterator types:

AT_SEQUENCE_NUMBER - Start reading from the position denoted by a specific sequence number, provided in the value StartingSequenceNumber.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

AFTER_SEQUENCE_NUMBER - Start reading right after the position denoted by a specific sequence number, provided in the value StartingSequenceNumber.

AT_TIMESTAMP - Start reading from the position denoted by a specific timestamp, provided in the value Timestamp.

TRIM_HORIZON - Start reading at the last untrimmed record in the shard in the system, which is the oldest data record in the shard.

LATEST - Start reading just after the most recent record in the shard, so that you always read the most recent data in the shard.

To read and process a stream, your application will need to connect to a DynamoDB Streams endpoint and issue API requests.

A stream consists of *stream records*. Each stream record represents a single data modification in the DynamoDB table to which the stream belongs. Each stream record is assigned a sequence number, reflecting the order in which the record was published to the stream.

Stream records are organized into *shards*. Each shard acts as a container for multiple stream records and contains information required for accessing and iterating through these records. The stream records within a shard are removed automatically after 24 hours.

Shards are ephemeral: they are created and deleted automatically, as needed. Any shard can split into multiple new shards and it occurs automatically. Note that it is also possible for a parent shard to have just one child shard. A shard might split in response to high levels of write activity on its parent table, so that applications can process records from multiple shards in parallel.

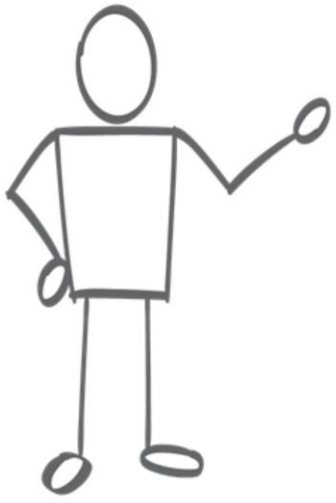
If you disable a stream, shards that are open will be closed.

Shards have lineage (parent and children) and an application must always process a parent shard before it processes a child shard. This will ensure that the stream records are also processed in the correct order.

When you use the serverless streaming services though, most of these concepts are abstracted for you, so you don’t have to worry about how the number of shards or which shard you are reading from.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



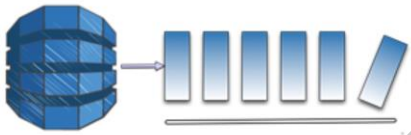
DynamoDB Streams

© 2016 Amazon Web Services, Inc. and its affiliates. All rights reserved.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

DynamoDB Streams



- Stream of updates to a table
- Asynchronous
- Exactly once
- Strictly ordered
  - Per item
- Highly durable
  - Scale with table
- 24-hour lifetime
- Subsecond latency

© 2016 Amazon Web Services, Inc. and its affiliates. All rights reserved.

amazon web services | Training and Certification 10

Many applications can benefit from the ability to capture changes to items stored in a DynamoDB table, at the point in time when such changes occur.

A *DynamoDB stream* is an ordered flow of information about changes to items in an Amazon DynamoDB table. When you enable a stream on a table, DynamoDB captures information about every modification to data items in the table.

Whenever an application creates, updates, or deletes items in the table, DynamoDB Streams writes a stream record with the primary key attributes of the items that were modified. A *stream record* contains information about a data modification to a single item in a DynamoDB table. You can configure the stream so that the stream records capture additional information, such as the "before" and "after" images of modified items.

DynamoDB Streams guarantees the following:

- Each stream record appears exactly once in the stream.

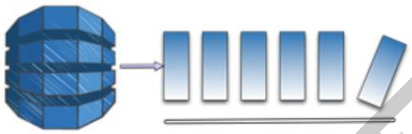
- For each item that is modified in a DynamoDB table, the stream records appear in the same sequence as the actual modifications to the item.

DynamoDB Streams writes stream records in near real time, so that you can build applications that consume these streams and take action based on the contents.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

View types



UpdateItem (Name = John, Destination = Pluto)

View Type	Destination
Old image—before update	Name = John, Destination = Mars
New image—after update	Name = John, Destination = Pluto
Old and new images	Name = John, Destination = Mars Name = John, Destination = Pluto
Keys only	Name = John

© 2016 Amazon Web Services, Inc. and its affiliates. All rights reserved.

amazon web services | Training and Certification 11

When you create a DynamoDB stream, specify the View Type that determines what data will be written to the stream whenever data in the table is modified.

The options are:

KEYS_ONLY—only the key attributes of the modified item.

NEW_IMAGE—the entire item, as it appears after it was modified.

OLD_IMAGE—the entire item, as it appeared before it was modified.

NEW_AND_OLD_IMAGES—both the new and the old images of the item.

In the example, let's assume you have a table in DynamoDB where the Primary Key is the Name field, with an attribute of Destination. You have an item in the table already that has a Key value = John and Destination value = Mars. The item is modified and the Destination value is now Pluto.

If your View Type is set to Old Image, you will have an object in the stream that has the data: Name = John, Destination = Mars.

If your View Type is set to New Image, you will have an object in the stream that has the data: Name = John, Destination = Pluto.

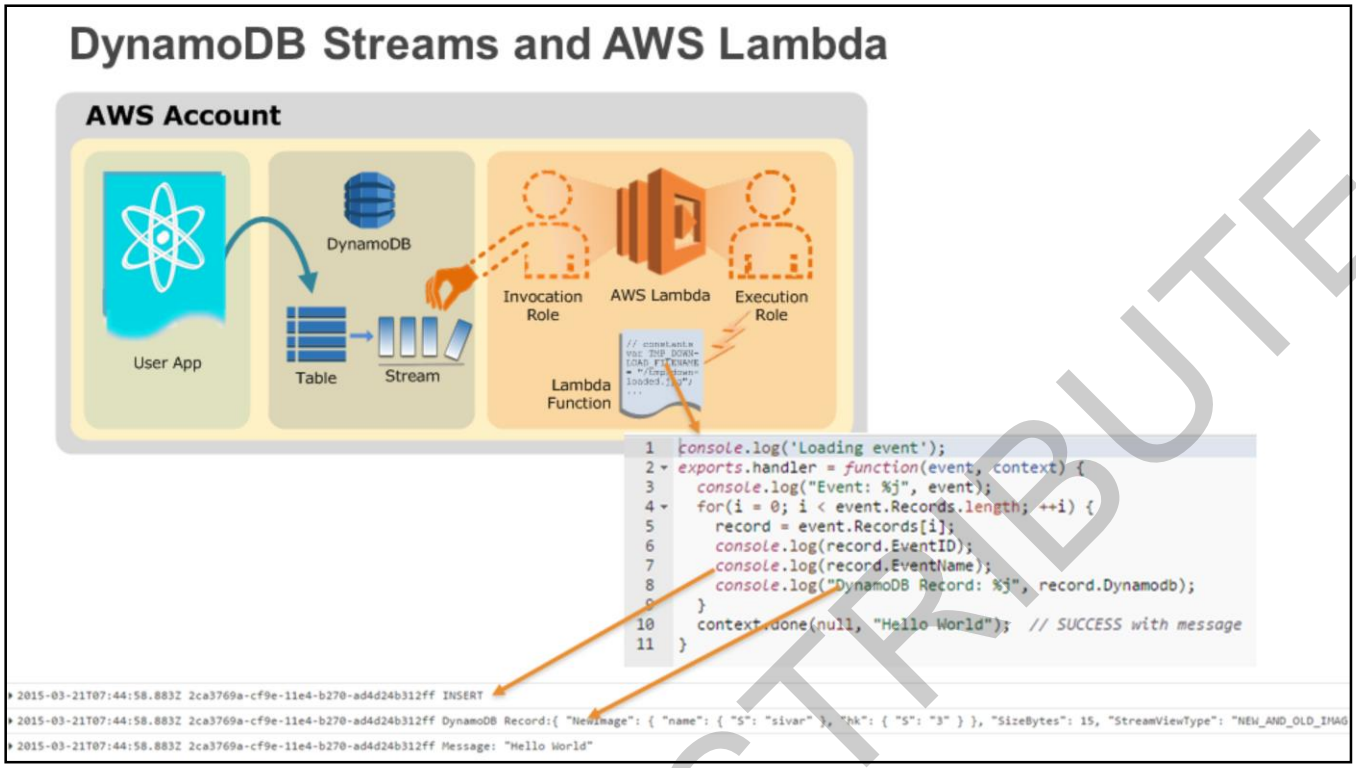
DO NOT DISTRIBUTE

If your View Type is set to Old and New Images, you will have an object in the stream that specifies the old and new values.

If your View Type is set to Keys only, you will have an object in the the stream that has the data: Name = John, which indicates which item changed.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



Amazon DynamoDB is integrated with AWS Lambda, so that you can create *triggers* - pieces of code that quickly and automatically respond to events in DynamoDB Streams. With triggers, you can build applications that react to data modifications in DynamoDB tables.

If you enable DynamoDB Streams on a table, you can associate the stream ARN with a Lambda function that you write. Whenever an item in the table is modified, a new stream record appears in the table's stream. AWS Lambda polls the stream and invokes your Lambda function when it detects new stream records.

When integrating DynamoDB Streams and Lambda, all the concepts of shards and iterators are abstracted from you. You will be able to pull your records from the stream without a lot of knowledge of those underlying stream concepts.

One of the streaming concepts you still need to be aware of is the type of Shard Iterator you want used for your Lambda function. You have the choice of TRIM HORIZON or LATEST. As you recall from earlier the definitions of those are:

TRIM_HORIZON - Start reading at the last untrimmed record in the shard in the system, which is the oldest data record in the shard.

LATEST - Start reading just after the most recent record in the shard, so that you always read the most recent data in the shard.

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

DO NOT DISTRIBUTE

This specifies whether you want your Lambda function to process the most recent or oldest data in the stream. After you determine this, the Lambda function can perform any actions that you specify, such as sending a notification or initiating a workflow. If your Lambda function needs to access any AWS resources, you need to grant the relevant permissions to access those resources. You also need to grant AWS Lambda permissions to poll your DynamoDB stream. You grant **all these** permissions to an IAM role (execution role) that AWS Lambda can assume to poll the stream and execute the Lambda function on your behalf. You create this role first and then enable it at the time you create the Lambda function.

Note the following about how the Amazon DynamoDB and AWS Lambda integration works:

Stream-based model – This is a model (see Event Source Mapping), where AWS Lambda polls the stream and, when it detects new records, invokes your Lambda function by passing the update event as parameter.

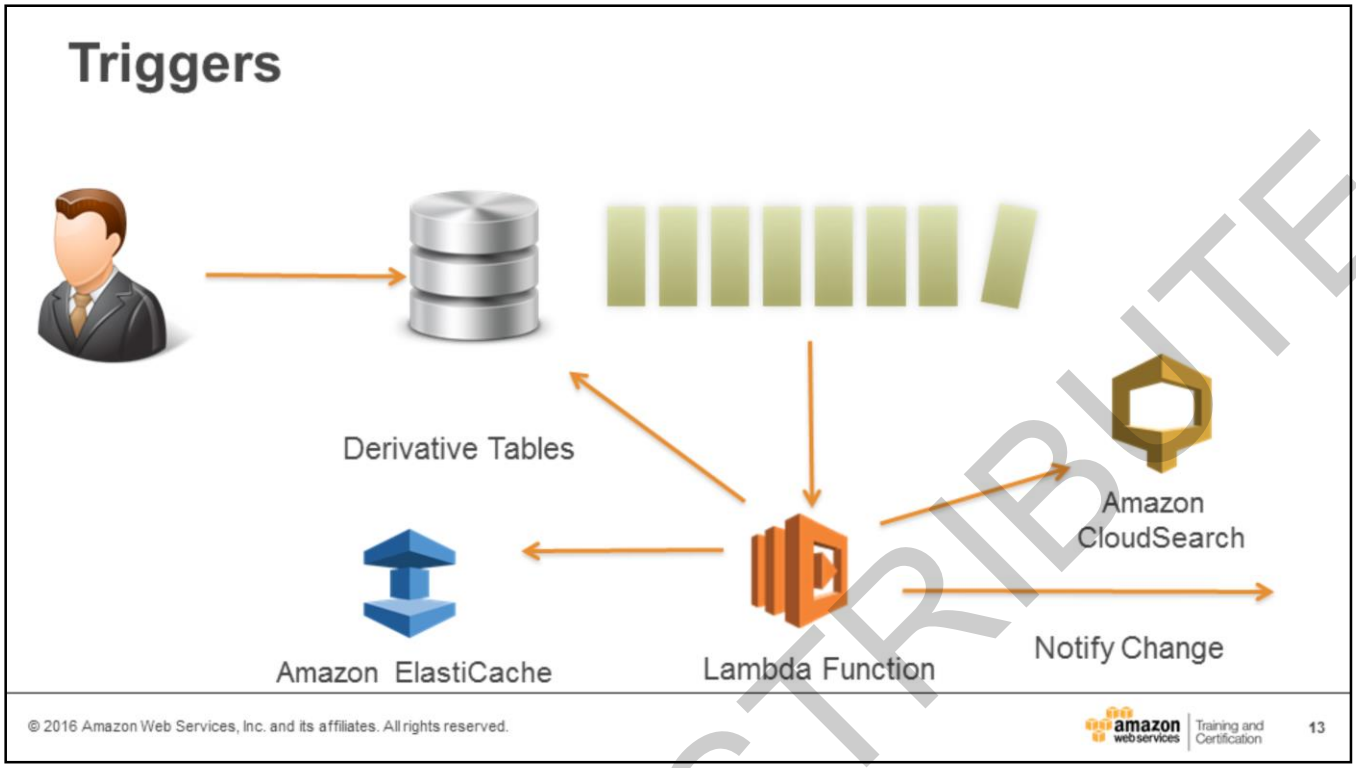
In a stream-based model, you maintain event source mapping in AWS Lambda. The event source mapping describes which stream maps to which Lambda function. AWS Lambda provides an API (CreateEventSourceMapping) for you to create the mapping. You can also the AWS Lambda console to create event source mappings.

Synchronous invocation – AWS Lambda invokes a Lambda function using the RequestResponse invocation type (synchronous invocation). For more information about invocation types, see Invocation Types.

Event structure – The event your Lambda function receives is the table update information AWS Lambda reads from your stream. When you configure event source mapping, the batch size you specify is the maximum number of records that you want your Lambda function to receive per invocation.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

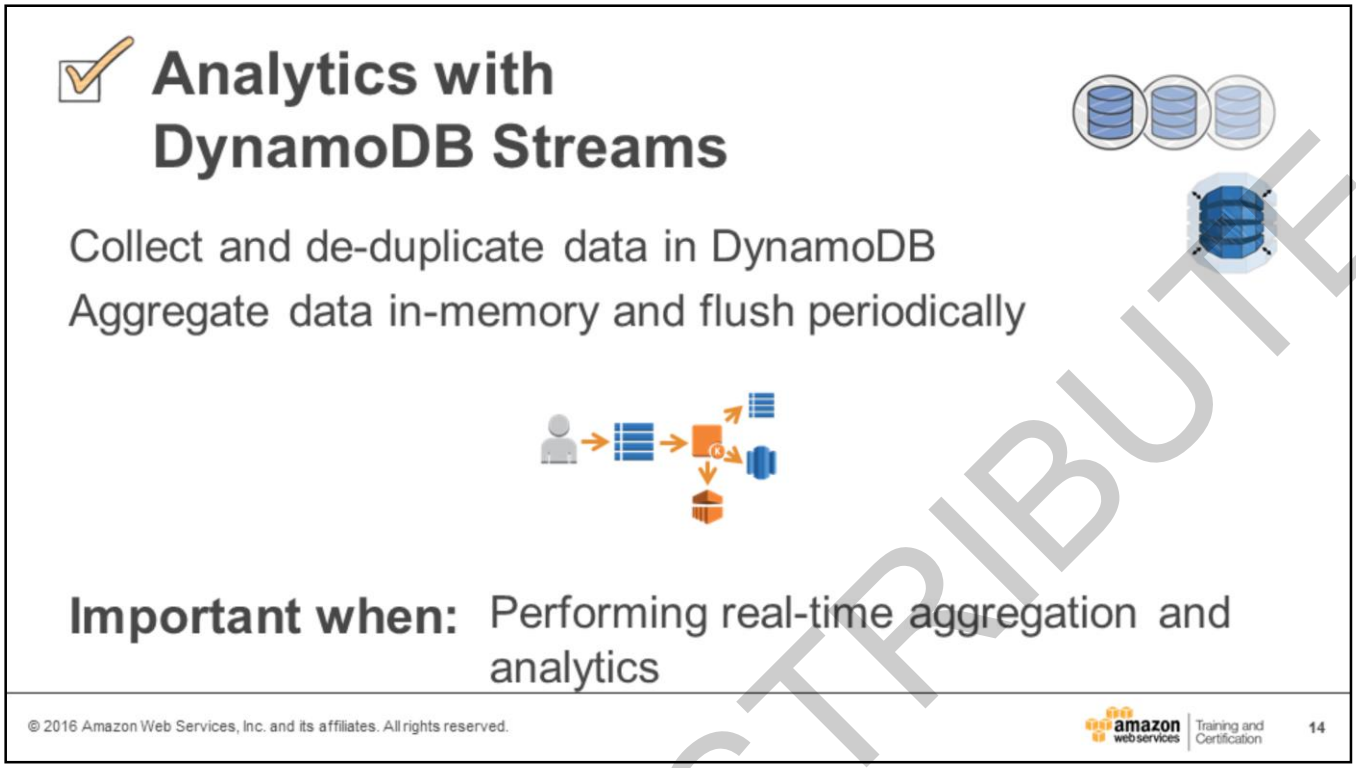


There are different use cases that DynamoDB Streams and Lambda can enable that do not require any servers for you to manage. You can have this act as a database trigger and populate a search index like CloudSearch or ElasticSearch, or populate an in-memory cache like ElastiCache.

You can also have this drive notifications out to your users based on state changes. You could do this via the Simple Email Service or Simple Notification service. All of these and much more are possibilities for your event-driven application enabled by these services.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

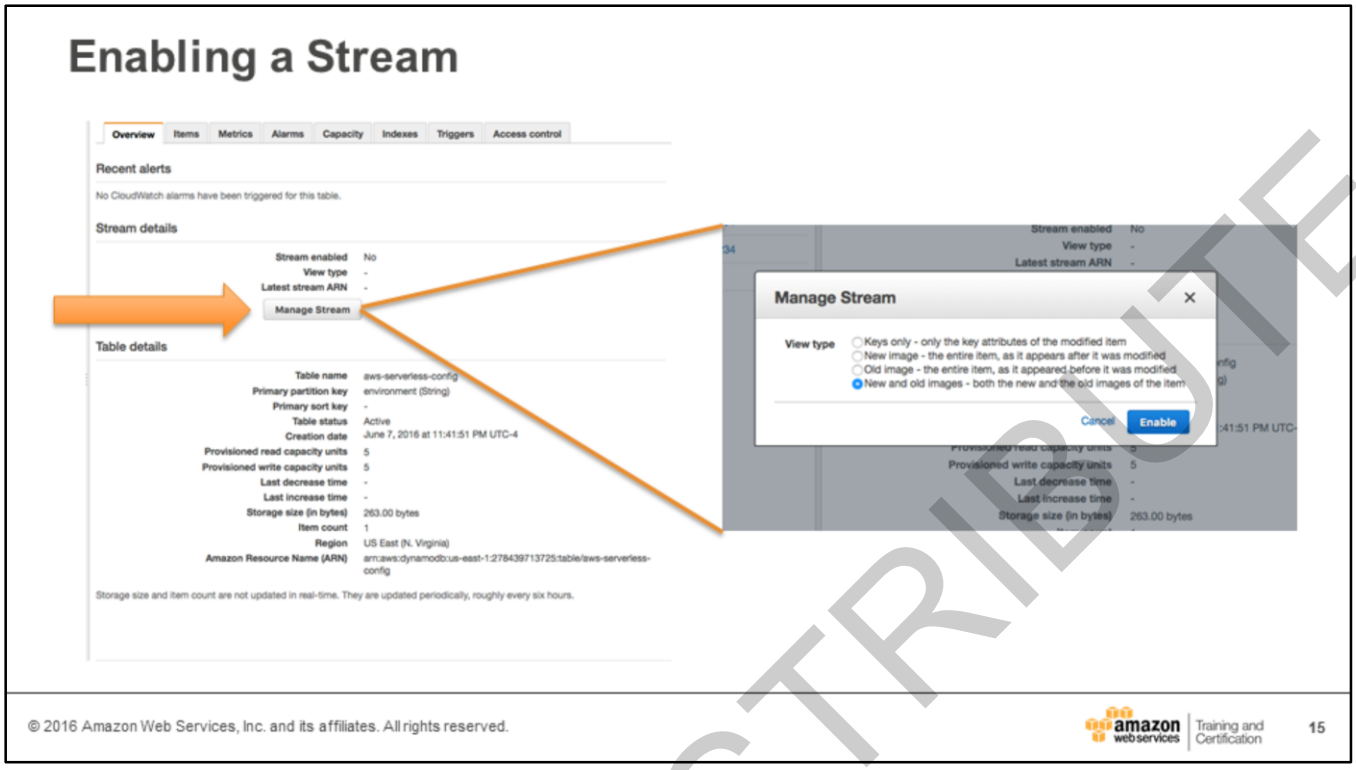


Not only can you use DynamoDB Streams to enable event-driven notifications and triggers within your application, but you can also use these updates to drive real-time analytics on your application. You can aggregate and de-duplicate the data and feed it into another DynamoDB Table that's driving a real-time analytics dashboard, you can feed it into a Redshift cluster for reporting, or feed the data into an Amazon Elastic Map Reduce cluster for additional analytics.

For example, if you have a popular mobile app modify data in a DynamoDB table at the rate of thousands of updates per second, you can have another application capture and store data about these updates, providing near real time usage metrics for the mobile app.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



You can enable a stream on a new table when you create it. You can also enable or disable a stream on an existing table, or change the settings of a stream. DynamoDB Streams operates asynchronously, so there is no performance impact on a table if you enable a stream.

The easiest way to manage DynamoDB Streams is by using the AWS Management Console.

1) Open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.

2) From the DynamoDB console dashboard, choose Tables.

3) On the Overview tab, choose Manage Stream.

4) In the Manage Stream window, choose the information that will be written to the stream whenever data in the table is modified:

Keys only—only the key attributes of the modified item.

New image—the entire item, as it appears after it was modified.

Old image—the entire item, as it appeared before it was modified.

New and old images—both the new and the old images of the item.

When the settings are as you want them, choose Enable.

5) (Optional) To disable an existing stream, choose Manage Stream and then choose Disable.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

You can also use the CreateTable or UpdateTable APIs to enable or modify a stream. The StreamSpecification parameter determines how the stream is configured:

- StreamEnabled—specifies whether a stream is enabled (true) or disabled (false) for the table.

- StreamViewType—specifies the information that will be written to the stream whenever data in the table is modified:

KEYS_ONLY—only the key attributes of the modified item.

NEW_IMAGE—the entire item, as it appears after it was modified.

OLD_IMAGE—the entire item, as it appeared before it was modified.

NEW_AND_OLD_IMAGES—both the new and the old images of the item.

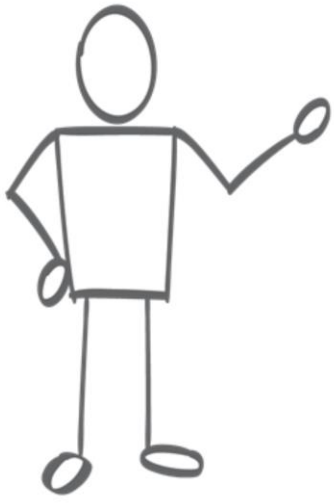
You can enable or disable a stream at any time. Note that you will receive a *ResourceInUseException* if you attempt to enable a stream on a table that already has a stream and you will receive a *ValidationException* if you attempt to disable a stream on a table that does not have a stream.

When you set StreamEnabled to true, DynamoDB creates a new stream with a unique stream descriptor assigned to it. If you disable and then re-enable a stream on the table, a new stream will be created with a different stream descriptor.

Every stream is uniquely identified by an Amazon Resource Name (ARN).

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

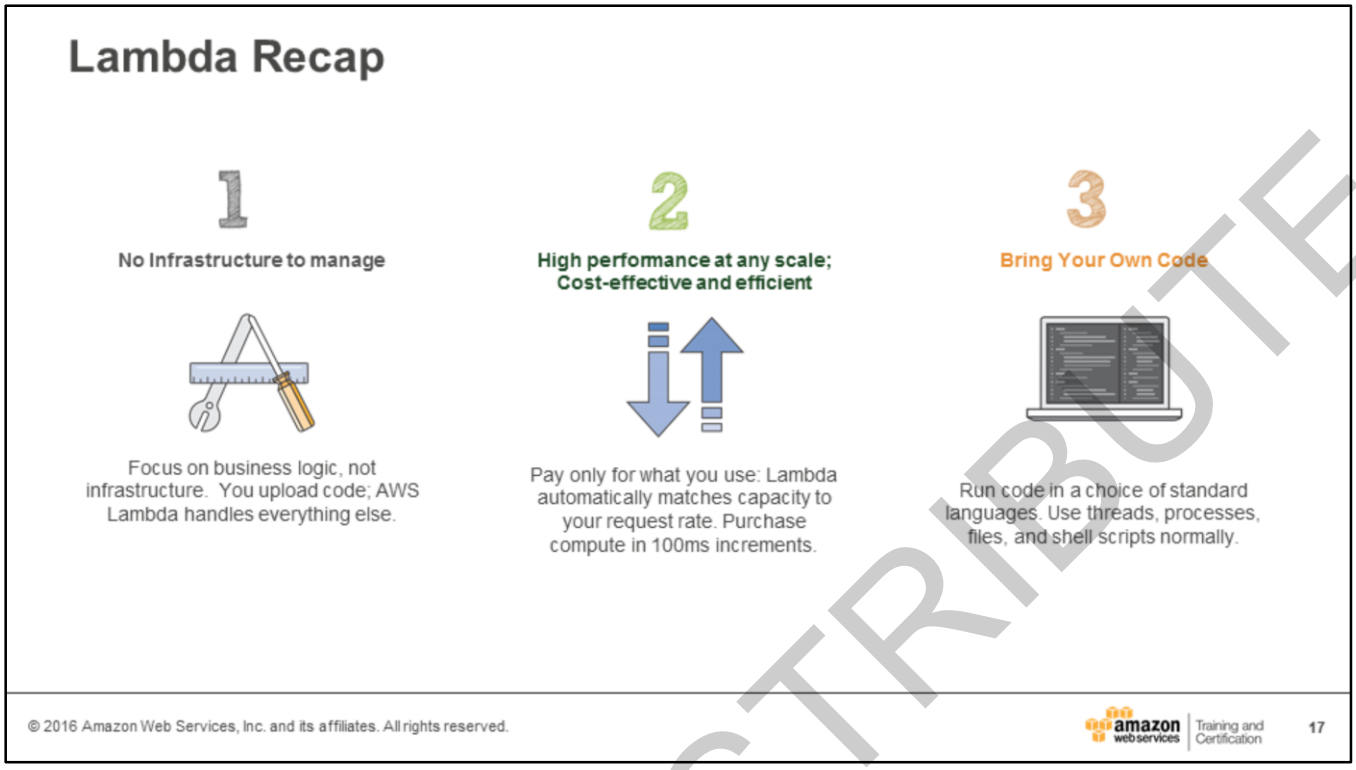


Lambda

© 2016 Amazon Web Services, Inc. and its affiliates. All rights reserved.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



- AWS Lambda is a compute service where you can upload your code to AWS Lambda and the service can run the code on your behalf using AWS infrastructure.

- AWS Lambda runs your code on a high-availability compute infrastructure and performs all of the administration of the compute resources, including server and operating system maintenance, capacity provisioning and automatic scaling, code monitoring, and logging.

- All you need to do is supply your code in one of the languages that AWS Lambda supports (currently Node.js, Java, and Python).

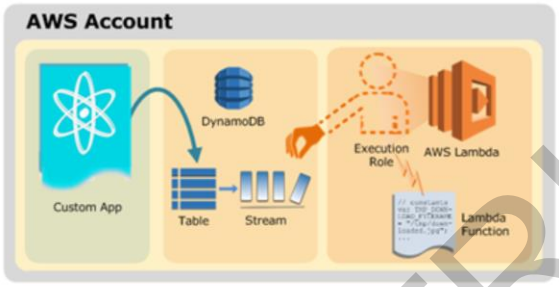
DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

Push vs. Pull Event Model

Pull event model:

AWS Lambda polls the event source and invokes your Lambda function when it detects an event.



© 2016 Amazon Web Services, Inc. and its affiliates. All rights reserved.

The Amazon Kinesis Streams and DynamoDB streams are the stream-based services that you can preconfigure with AWS Lambda. After you do the necessary event source mapping, AWS Lambda polls the streams and invokes your Lambda function (referred to as the *pull model*). In the pull model, note the following:

- The event source mappings are maintained within the AWS Lambda. AWS Lambda provides the relevant APIs to create and manage event source mappings.
- AWS Lambda needs your permission to poll the stream and read records. You grant these permissions via the execution role, using the permissions policy associated with role that you specify when you create your Lambda function. AWS Lambda does not need any permissions to invoke your Lambda function.

The diagram illustrates the following sequence:

1. The custom application writes records to a DynamoDB stream.
2. AWS Lambda continuously polls the stream and invokes the Lambda function when the service detects new records on the stream. AWS Lambda knows which stream to

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

DO NOT DISTRIBUTE

poll and which Lambda function to invoke based on the event source mapping you create in AWS Lambda.

3. AWS Lambda then executes the Lambda function.

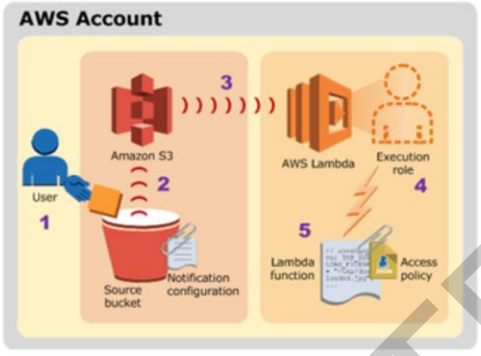
DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

Push vs. Pull Event Model (Continued)

Push event model

- The Event source directly invokes a Lambda function when it publishes an event.



© 2016 Amazon Web Services, Inc. and its affiliates. All rights reserved.

AWS services publish events and can also invoke your Lambda function (referred to as the *push model*). In the push model, note the following:

- Event source mappings are maintained within the event source. Relevant API support in the event sources enables you to create and manage event source mappings. For example, Amazon S3 provides the bucket notification configuration API. Using this API, you can configure an event source mapping that identifies the bucket events to publish and the Lambda function to invoke.

- Because the event sources invoke your Lambda function, you need to grant the event source the necessary permissions using a resource-based policy (referred to as the *Lambda function policy*).

The diagram illustrates the flow:

1. The user creates an object in a bucket.

2. Amazon S3 detects the object created event.

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

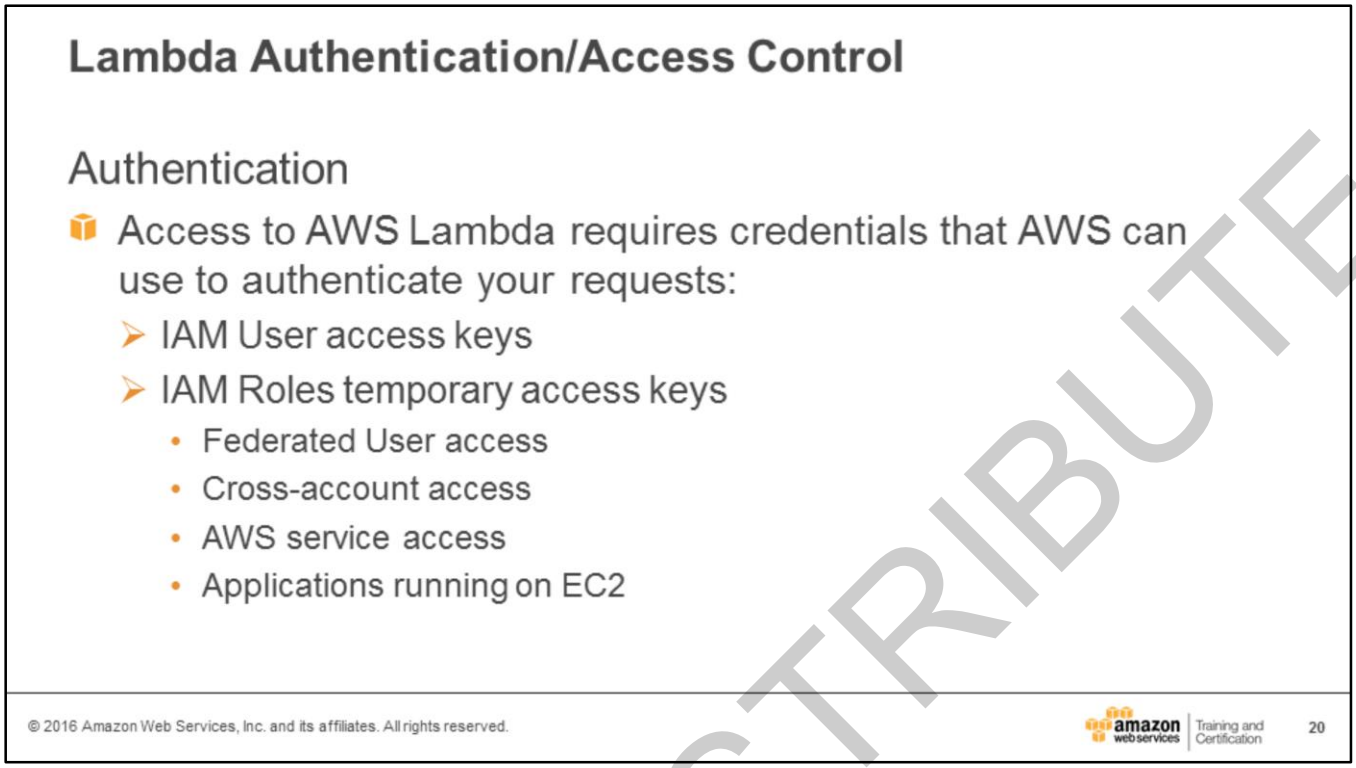
DO NOT DISTRIBUTE

3. Amazon S3 invokes your Lambda function according to the event source mapping described in the bucket notification configuration.

4. AWS Lambda verifies the permissions policy attached to the Lambda function to ensure that Amazon S3 has the necessary permissions, and then executes the Lambda function. Remember that your Lambda function receives the event as parameter.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



You can access AWS as any of the following types of identities:

- **IAM user** – An IAM user is simply an identity within your AWS account that has specific custom permissions (for example, permissions to create a function in Lambda).

- **IAM role** – An IAM role is another IAM identity you can create in your account that has specific permissions. It is similar to an *IAM user*, but it is not associated with a specific person. An IAM role enables you to obtain temporary access keys that can be used to access AWS services and resources.

IAM roles with temporary credentials are useful in the following situations:

- **Federated user access** – Instead of creating an IAM user, you can use preexisting user identities from AWS Directory Service, your enterprise user directory, or a web identity provider.

- **Cross-account access** – You can use an IAM role in your account to grant another

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

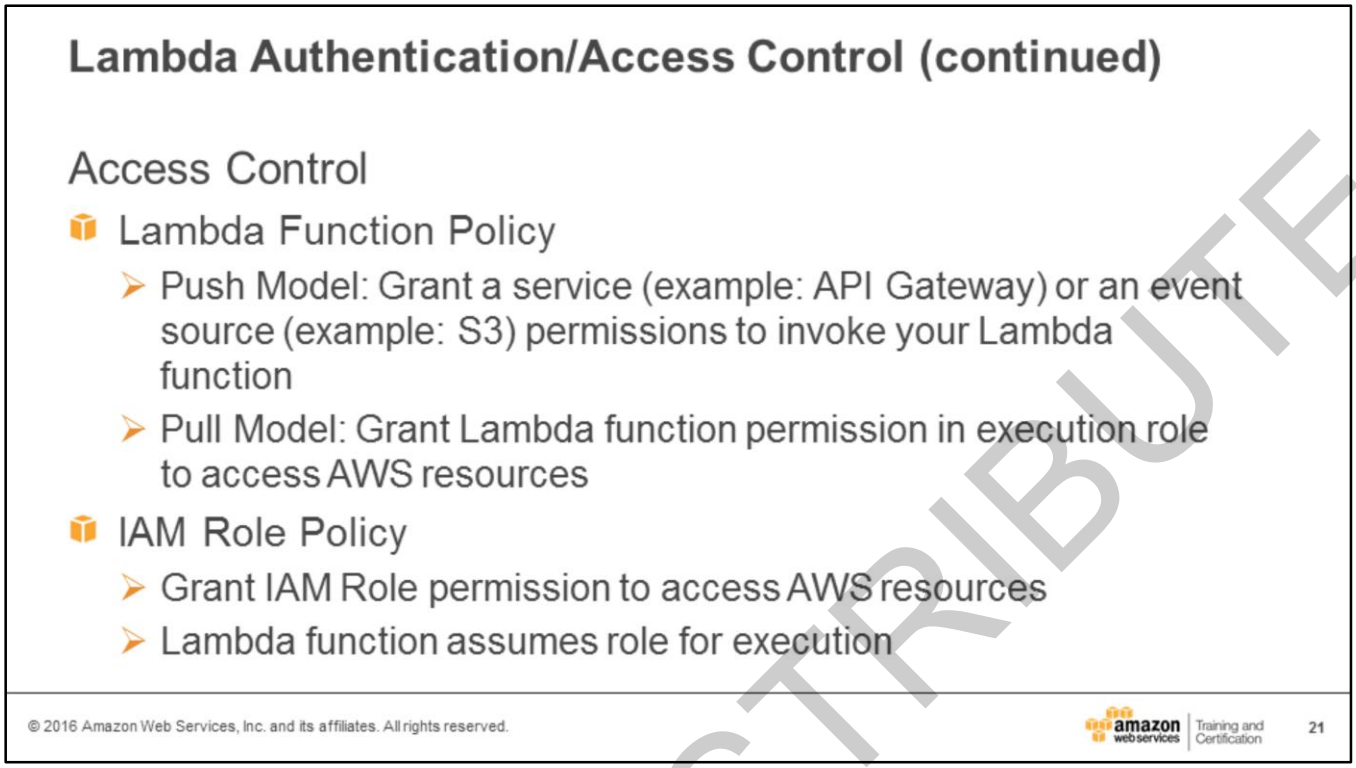
AWS account permissions to access your account's resources.

- **AWS service access** – You can use an IAM role in your account to grant an AWS service permissions to access your account's resources.

- **Applications running on EC2** – Sometimes applications running within EC2 need access to other AWS resources. Instead of storing access keys within the EC2 instance, you can use an IAM role to manage temporary credentials for these applications and get these credentials from the instance metadata.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



For the end-to-end AWS Lambda-based applications to work, you have to manage various permissions. For example:

- For stream-based event sources (Amazon Kinesis Streams and DynamoDB streams), AWS Lambda polls the streams on your behalf and reads new records on the stream, so you need to grant AWS Lambda permissions for the relevant stream actions.

- For event sources, except for the stream-based services (Amazon Kinesis Streams and DynamoDB streams), invoke your Lambda function, you must grant the event source permissions to invoke your AWS Lambda function.

Each Lambda function has an IAM role (execution role) associated with it**. You** specify the IAM role when you create your Lambda function. Permissions you grant to this role determine what AWS Lambda can do when it assumes the role. There are two types of permissions that you grant to the IAM role:

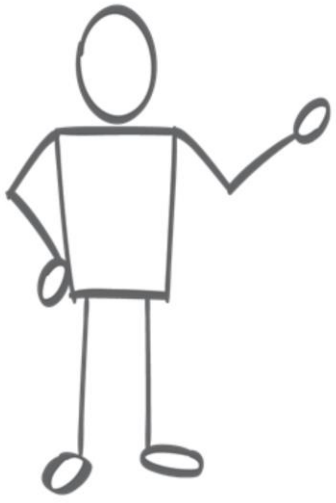
- If your Lambda function code accesses other AWS resources, such as to read an object from an S3 bucket or write logs to CloudWatch Logs, you need to grant permissions for relevant Amazon S3 and CloudWatch actions to the role.

DO NOT DISTRIBUTE

- If the event source is stream-based (Amazon Kinesis Streams and DynamoDB streams), AWS Lambda polls these streams on your behalf. AWS Lambda needs permissions to poll the stream and read new records on the stream so you need to grant the relevant permissions to this role.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

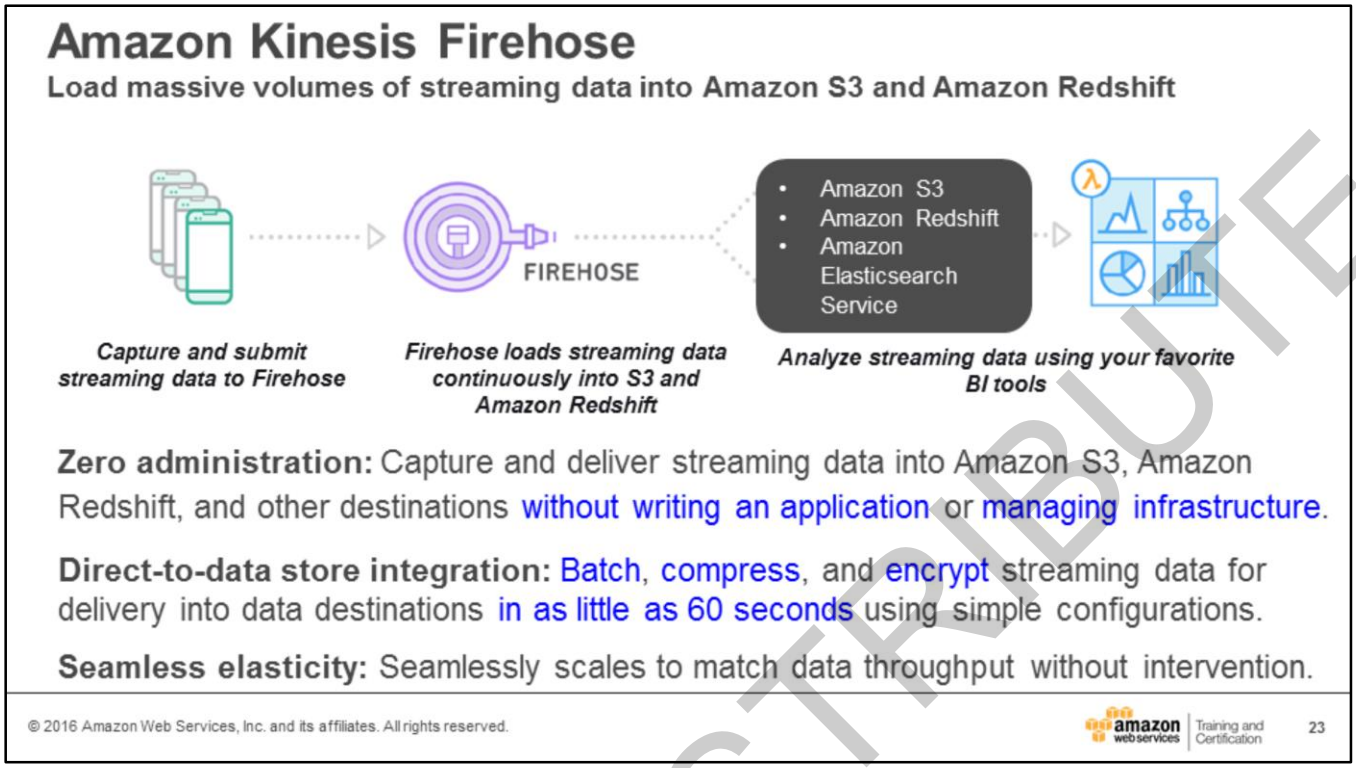


Kinesis Firehose

© 2016 Amazon Web Services, Inc. and its affiliates. All rights reserved.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

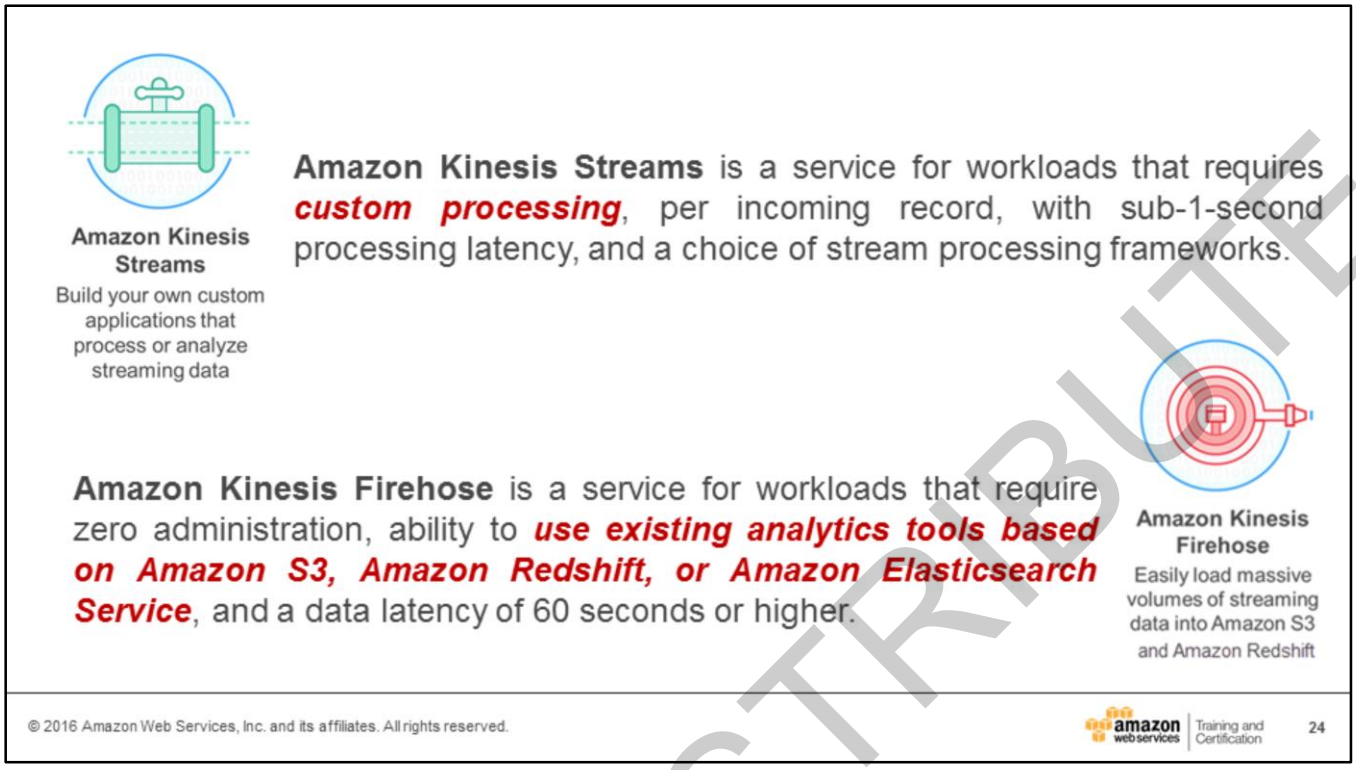


Amazon Kinesis Firehose is a fully managed service for delivering real-time streaming data to destinations such as Amazon Simple Storage Service (Amazon S3), Amazon Redshift, or Amazon Elasticsearch Service (Amazon ES). Firehose is part of the Amazon Kinesis streaming data platform, along with Amazon Kinesis Streams. With Firehose, you do not need to write any applications or manage any resources. You configure your data producers to send data to Firehose and it automatically delivers the data to the destination that you specified.

It is a fully managed service that automatically scales to match the throughput of your data and requires no ongoing administration. It can also batch, compress, and encrypt the data before loading it, minimizing the amount of storage used at the destination and increasing security.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

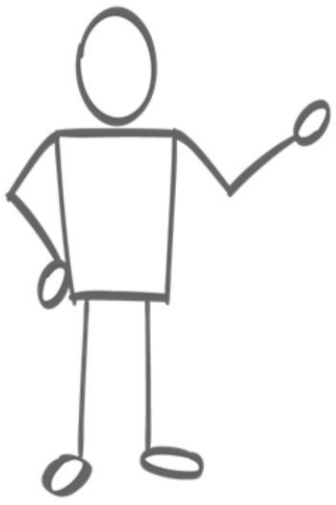


Amazon Kinesis Streams manages the infrastructure, storage, networking, and configuration needed to stream your data at the level of your data throughput. You do not have to worry about provisioning, deployment, ongoing-maintenance of hardware, software, or other services for your data streams. In addition, Amazon Kinesis Streams synchronously replicates data across three facilities in an AWS Region, providing high availability and data durability. With Kinesis Streams you are in charge of scaling the number of shards needed to support the throughput you require for your stream.

Amazon Kinesis Firehose manages all underlying infrastructure, storage, networking, and configuration needed to capture and load your data into Amazon S3, Amazon Redshift, or Amazon Elasticsearch Service. You do not have to worry about provisioning, deployment, ongoing maintenance of the hardware, software, or write any other application to manage this process. Firehose also scales elastically without requiring any intervention or associated developer overhead. Moreover, Amazon Kinesis Firehose synchronously replicates data across three facilities in an AWS Region, providing high availability and durability for the data as it is transported to the destinations.

DO NOT DISTRIBUTE

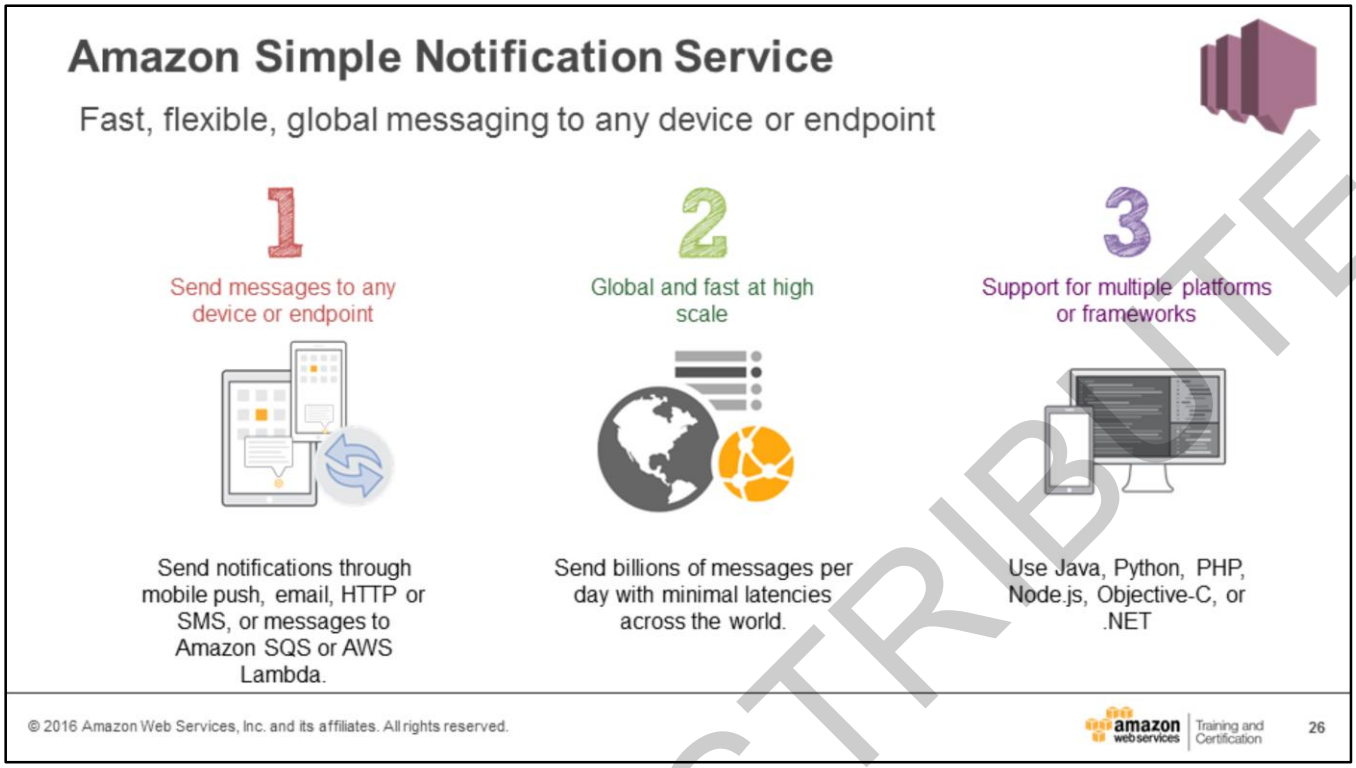
© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



© 2016 Amazon Web Services, Inc. and its affiliates. All rights reserved.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



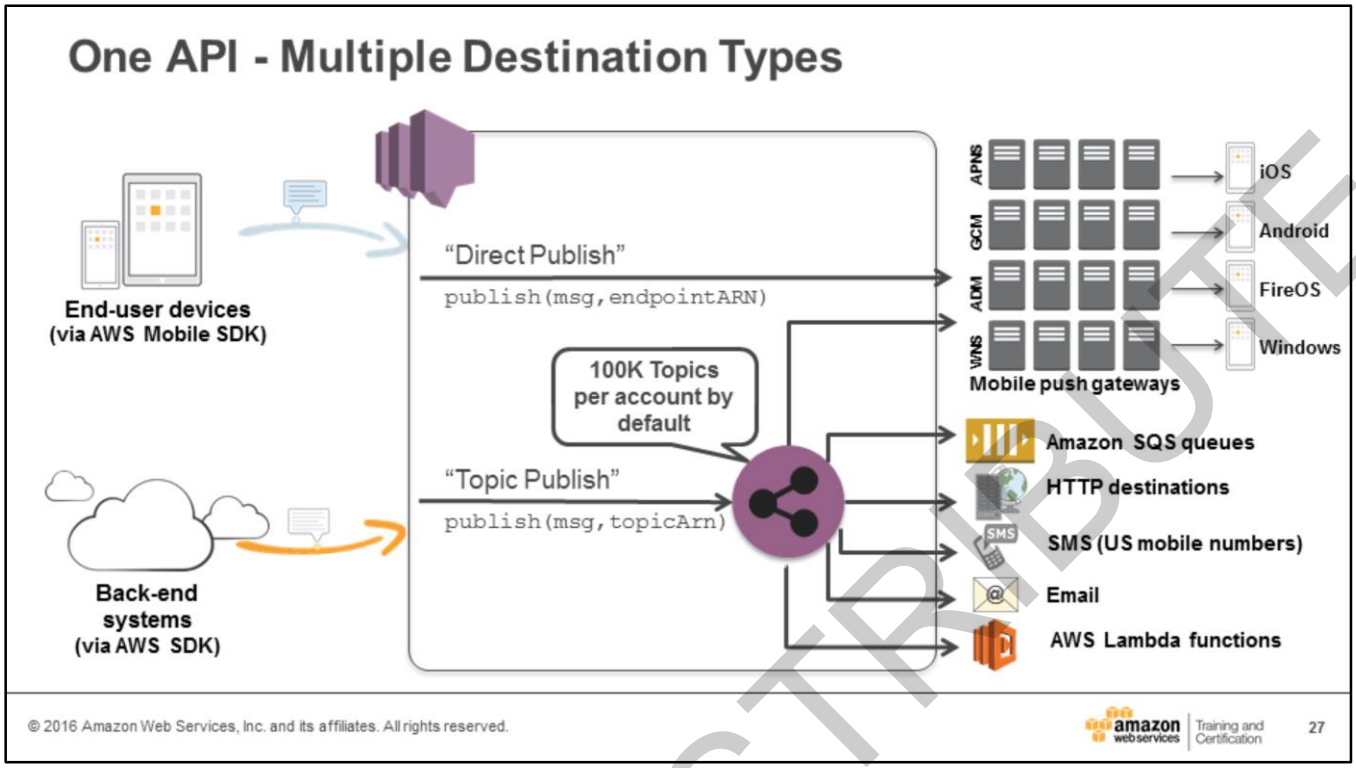
Amazon SNS allows applications and end-users on different devices to receive notifications via Mobile Push notification (Apple, Google and Kindle Fire Devices), HTTP/HTTPS, Email/Email-JSON, SMS or Amazon Simple Queue Service (SQS) queues, or AWS Lambda functions.

Amazon SNS is designed to meet the needs of the largest and most demanding applications, allowing applications to publish an unlimited number of messages at any time.

In most cases, developers can get started with Amazon SNS by using just three APIs: CreateTopic, Subscribe, and Publish. Additional APIs are available, which provide more advanced functionality.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

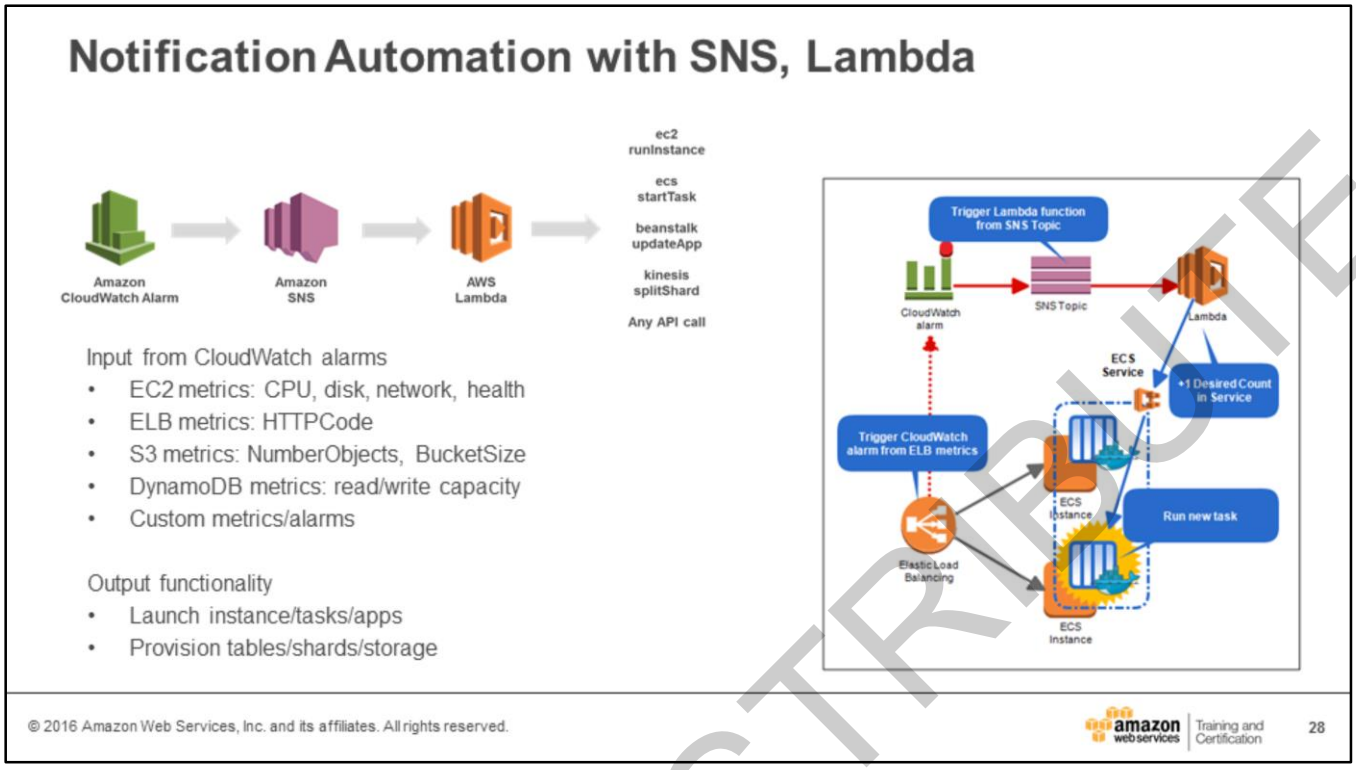


In order for customers to have broad flexibility of delivery mechanisms, Amazon SNS supports notifications over multiple transport protocols. Customers can select one the following transports as part of the subscription requests:

- "HTTP", "HTTPS" – Subscribers specify a URL as part of the subscription registration. Notifications will be delivered through an HTTP POST to the specified URL.
- "Email", "Email-JSON" – Messages are sent to registered addresses as email. Email-JSON sends notifications as a JSON object, while Email sends text-based email.
- "SQS" – Users can specify an SQS queue as the endpoint. Amazon SNS will enqueue a notification message to the specified queue (which subscribers can then process using SQS APIs such as ReceiveMessage, DeleteMessage, and so on.)
- "SMS" – Messages are sent to registered phone numbers as SMS text messages.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



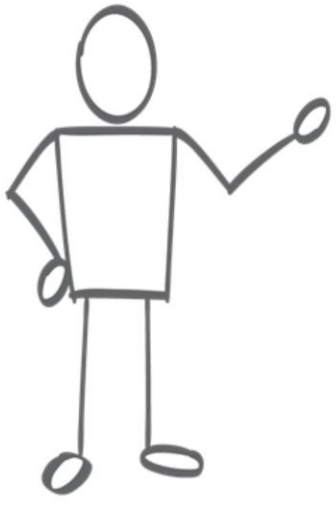
You can now invoke your AWS Lambda functions by publishing messages to Amazon SNS topics that have AWS Lambda functions subscribed to them. Amazon SNS supports message fan-out, **so** publishing a single message can invoke different AWS Lambda functions or invoke Lambda functions in addition to delivering notifications to supported Amazon SNS destinations such as mobile push, HTTP endpoints, SQS, email and SMS (US only).

By subscribing AWS Lambda functions to Amazon SNS topics, you can perform custom message handling. You can invoke an AWS Lambda function to provide custom message delivery handling by first publishing a message to an AWS Lambda function, have your Lambda function modify a message, for example, localize language and then filter and route those messages to other topics and endpoints.

A message delivery from Amazon SNS to an AWS Lambda function creates an instance of the AWS Lambda function and invokes it with your message as an input. You can also use delivery to an AWS Lambda function as a way to publish to other AWS services such as Amazon Kinesis or Amazon S3. You can subscribe an AWS Lambda function to the Amazon SNS topic and then have the Lambda function in turn write to another service.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

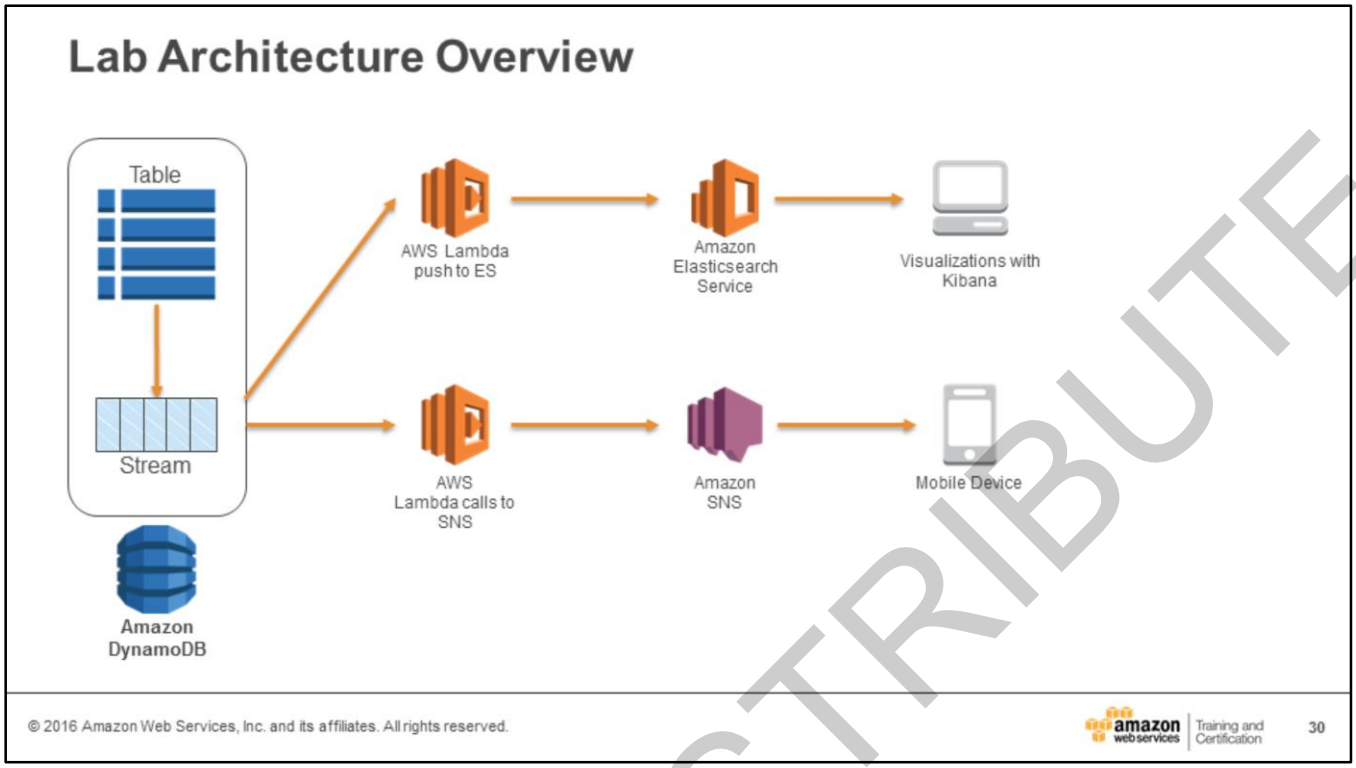


Lab 2

© 2016 Amazon Web Services, Inc. and its affiliates. All rights reserved.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

AWS Training and Certification

Self-Paced Labs

Try products, gain new skills, and get hands-on practice working with AWS technologies.

aws.amazon.com/training/self-paced-labs

Training

Skill up and gain confidence to design, develop, deploy, and manage your applications on AWS.

aws.amazon.com/training

Certification

Demonstrate your skills, knowledge, and expertise with the AWS platform.

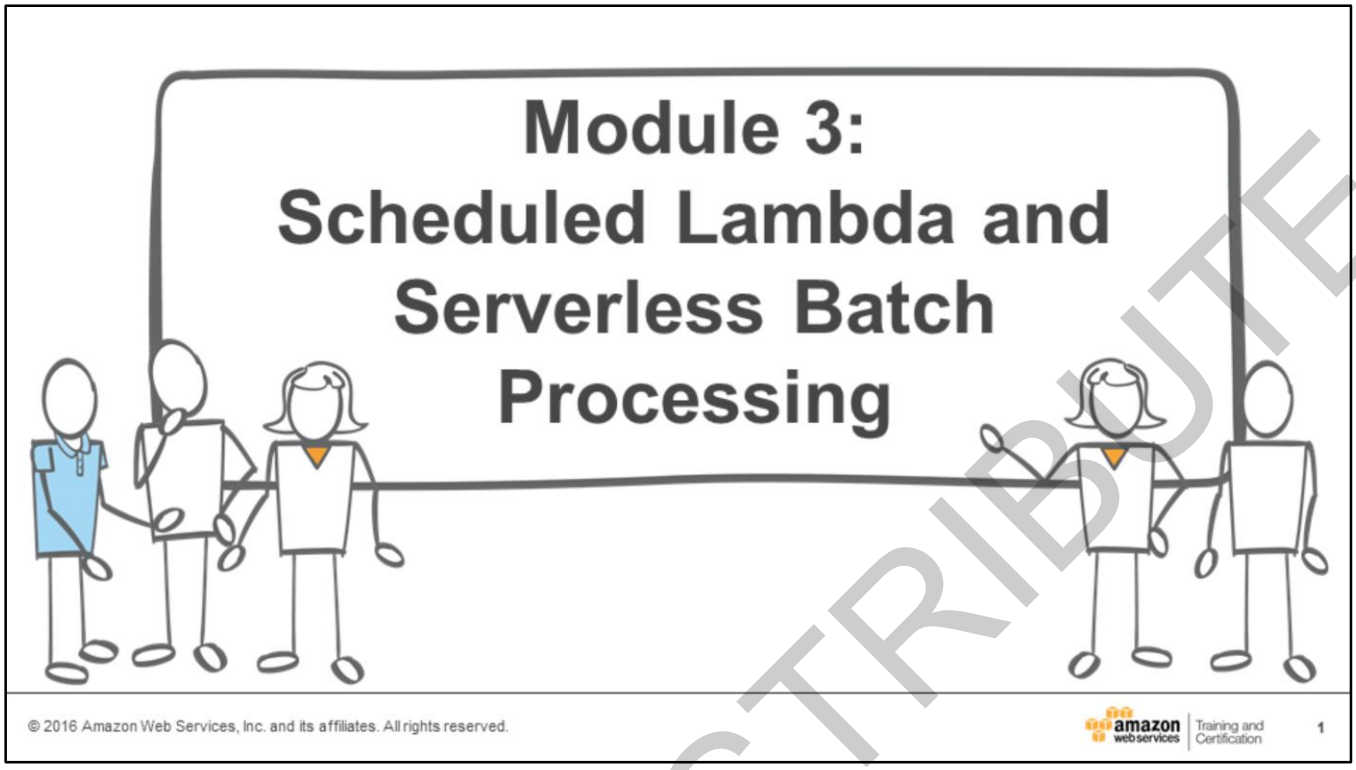
aws.amazon.com/certification

© 2016 Amazon Web Services, Inc. and its affiliates. All rights reserved.

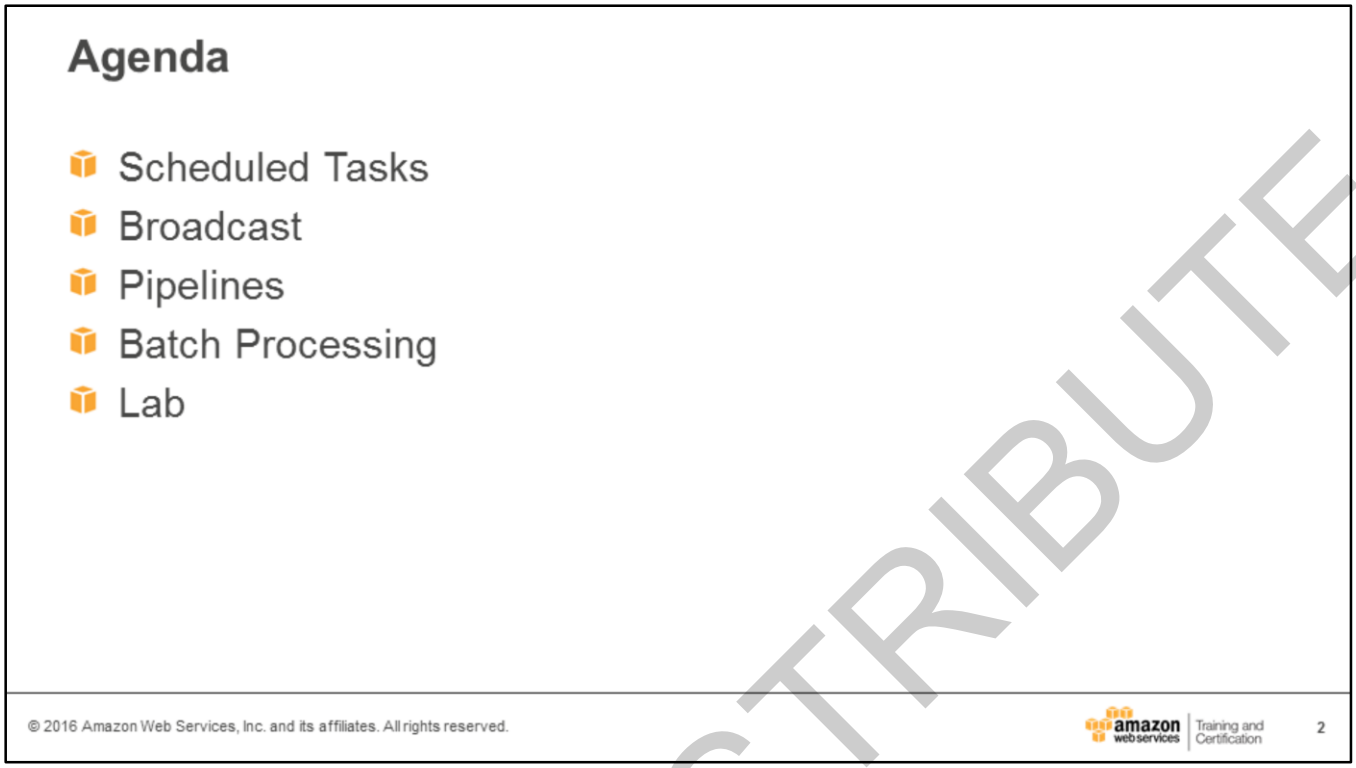
amazon web services | Training and Certification 31

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



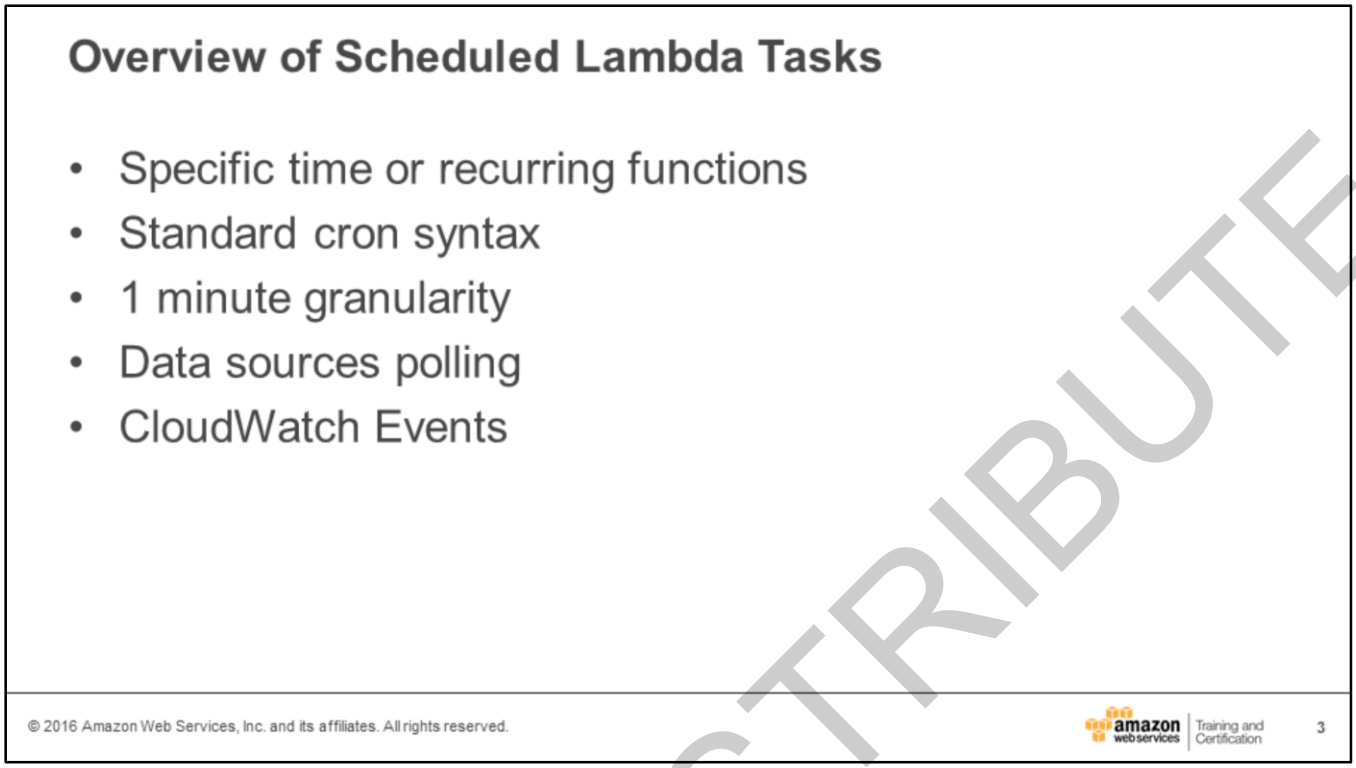
DO NOT DISTRIBUTE



- This module covers four patterns for building scheduled operations with Lambda and building coordinated pipelines with AWS Lambda.
- Because AWS Lambda is a stateless function execution, developers and operators use these patterns to help move data and state between Lambda functions during and between executions.
- Scheduled Tasks will focus on Lambda as a cron service.
- Broadcast Pattern will be the first step in decoupling Lambda executions in a one to many pattern.
- Pipelines will go into detail about connecting Lambda Functions together in one distributed transaction using AWS Lambda and other AWS Services.
- Batch Processing pulls these concepts together into a use case of running distributed workloads on AWS Lambda.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

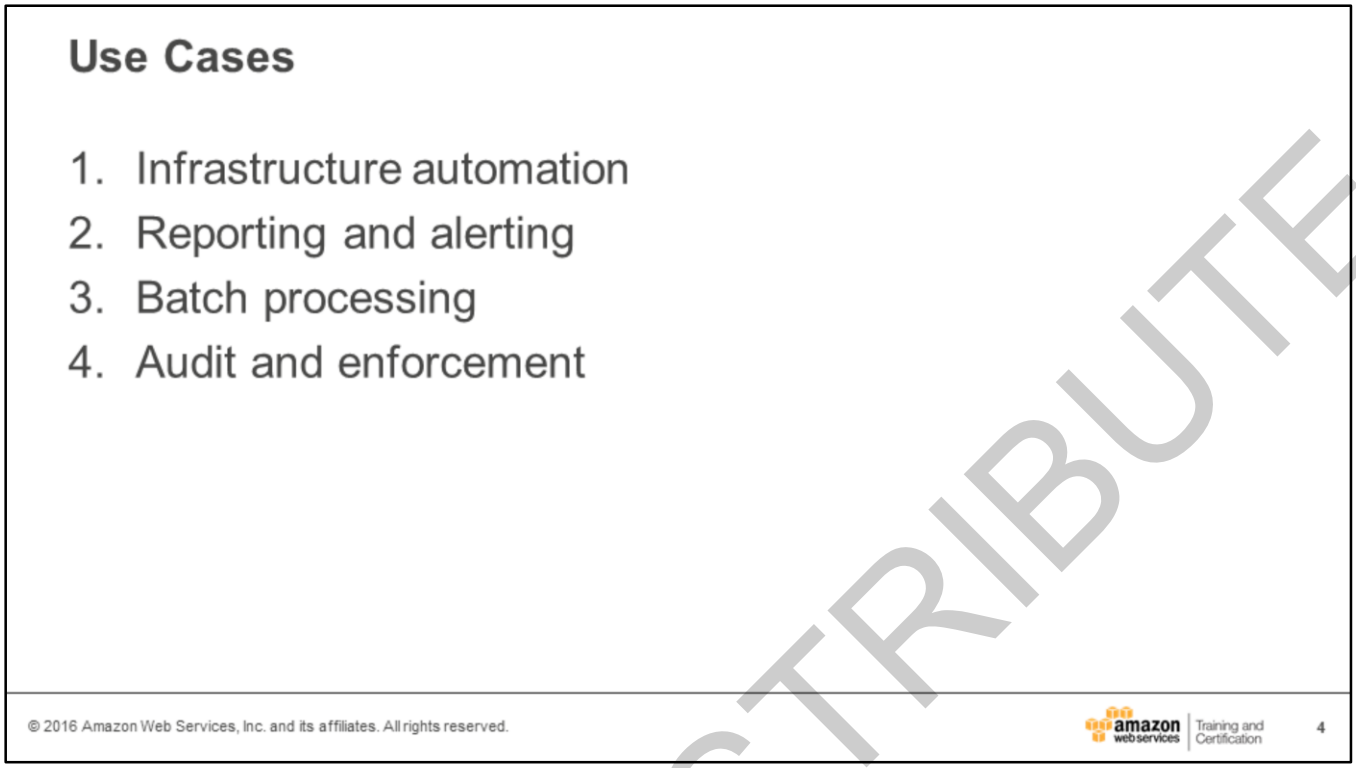


* AWS Lambda can be scheduled to run at a specific time (for example Monday at 3pm on December 1, 2016) or recurring (every Monday at 3pm).

- AWS Lambda accepts cron syntax for scheduling the occurrence and timing of a Lambda invocation.
- AWS Lambda can be scheduled as frequently as once a minute, which gives customers the ability to work on specific jobs that need to execute, for example, 10 past the hour or one minute until midnight, or as frequently as executing in one-minute increments.
- Because AWS Lambda uses IAM (Identity and Access Management) for assuming a role with specific AWS credentials, scheduled functions are a good use case for polling other AWS services or polling outside services.
- Scheduled Lambdas are triggered by CloudWatch Event Rules that act as a data source for Lambda Functions.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

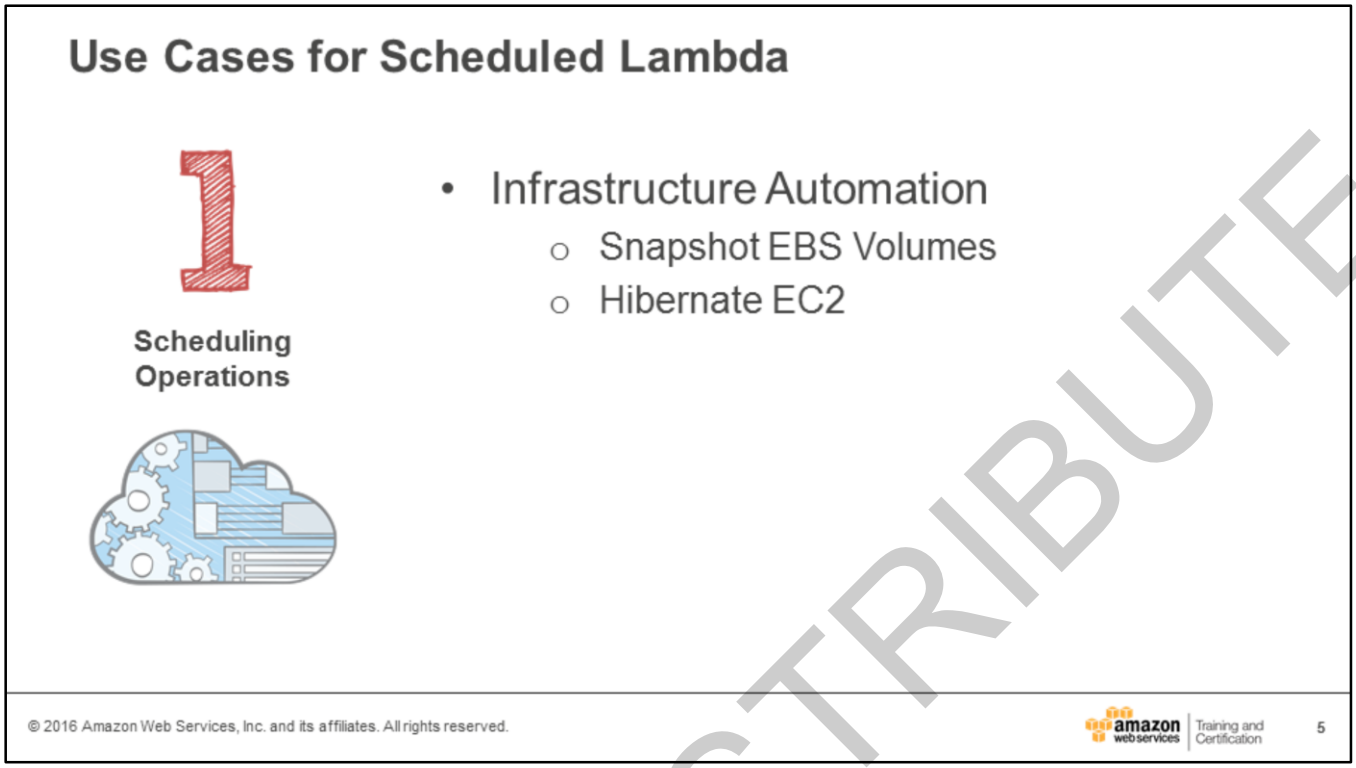


There are several use cases for scheduled functions. Think of common use cases you have today where you have an EC2 instance or a VM on premise whose sole job is to run cron tasks. These types of workloads are a natural fit for AWS Lambda.

Some specific use cases for Scheduled Tasks in Lambda are infrastructure automation, such as deleting or taking a snapshot of specific EBS volumes, reporting and alerting any alerts or reports that can be run with a one-minute interval, scheduling a Lambda in order to start a longer batch-running process like orchestrating the start and stop of other AWS resources, and even executing audit and enforcement checks using scheduled Lambda functions.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



To discuss infrastructure automation in more detail, here are two common use cases to start with.

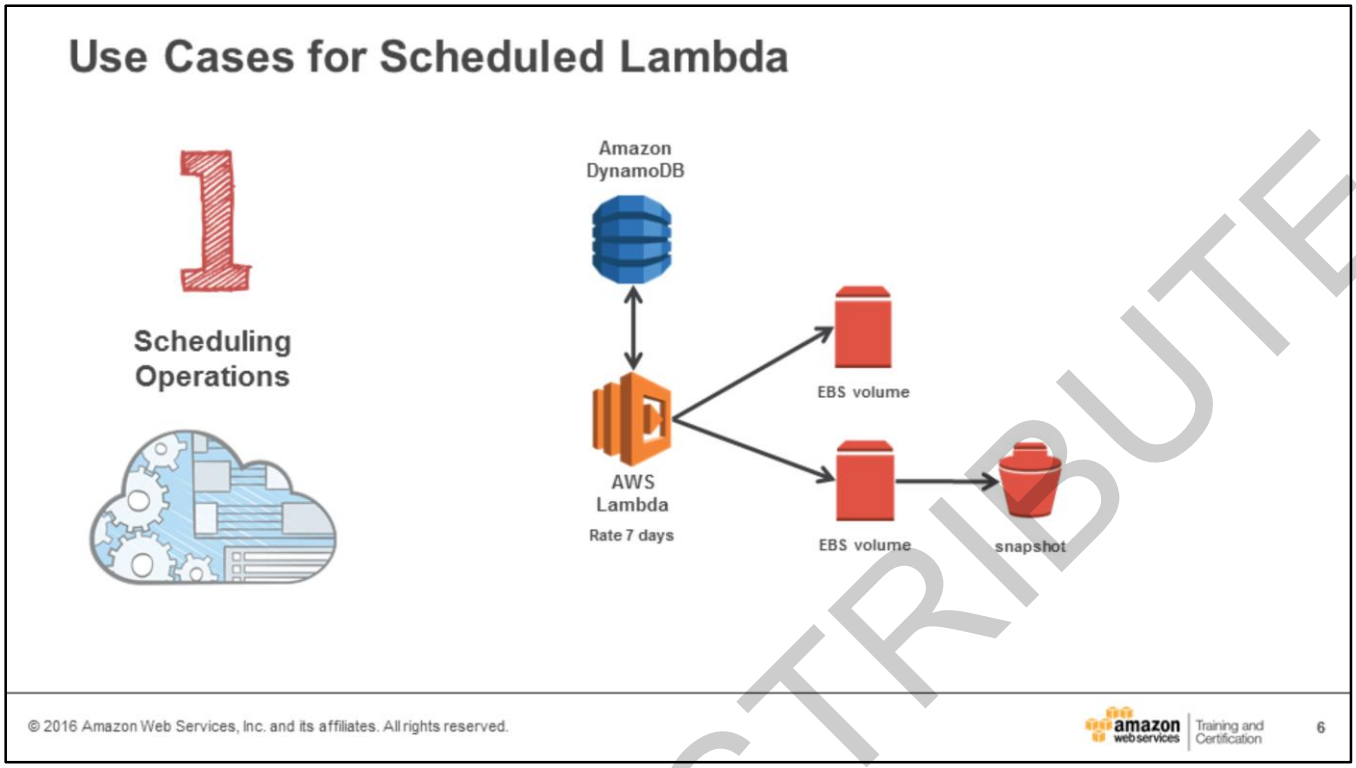
The first is snapshotting of EBS Volumes. With Scheduled Tasks in Lambda, you can use a Lambda function to not only search for specific EBS volumes using the AWS API, but you can also use it to then automate the process of snapshotting a specific EBS Volume, or take other operational tasks on an EBS Volume.

The second example use case for infrastructure automation is hibernating EC2 or stopping RDS instances based on a given schedule. For example, you may want to automatically hibernate an environment after a specific time of day or based on internal process metrics. By using a scheduled Lambda you can execute hibernating EC2 instances on a scheduled basis and even execute restarting those instances when required.

These are a subset of uses cases for infrastructure automation, but look for other opportunities when you are using the AWS SDK or CLI to continuously manage your infrastructure. AWS Lambda with scheduled tasks may be an opportunity to run your operations without having to run additional infrastructure.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

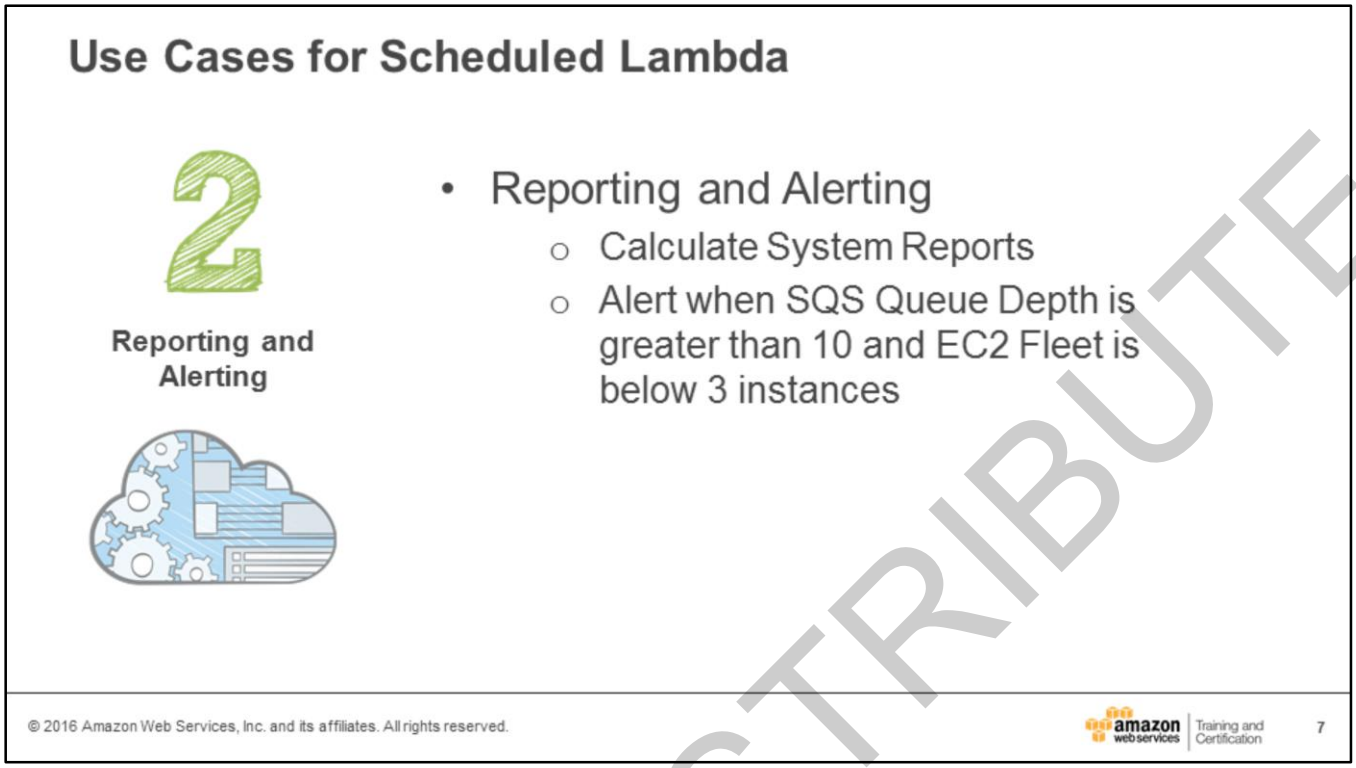


In this example, there is an AWS Lambda Function that is scheduled to execute every 7 days. It keeps track of a company's EBS Volume retention policies using an external service (in this case, DynamoDB), and then uses that to select specific EBS snapshots to execute. In this example, after it gets a list of potential EBS Volumes as candidates for snapshotting, the function then checks to see, based on internal policies, if both are required at this time. Based on business logic only the second volume is chosen for snapshotting.

In production, customers will have processes that specify which applications should be backed up and at what time. You can use one central Lambda function to coordinate this effort with a data store, or you can look to segment the types of snapshot and backup policies across several Lambda functions and data stores if you require more granularity at the application level.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

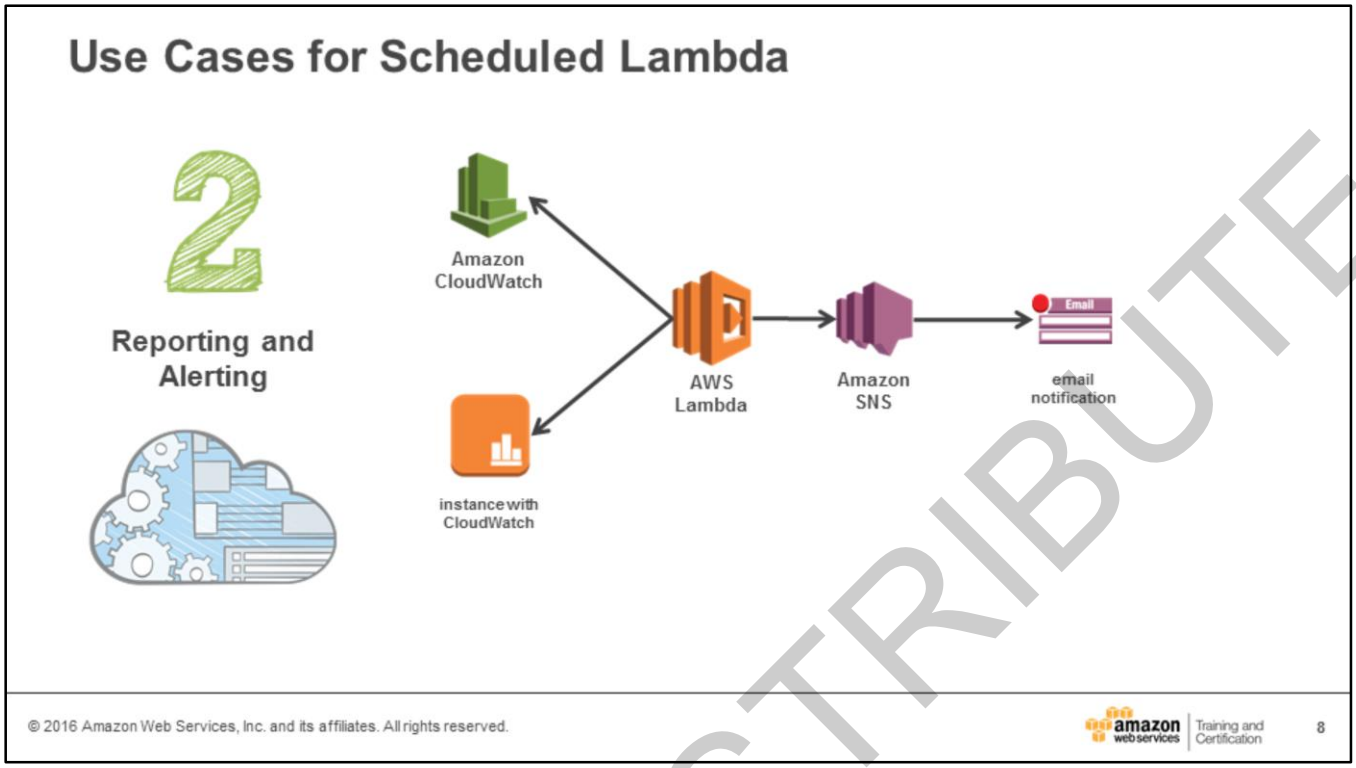


Scheduled Lambda Tasks can be utilized for creating or alerting on metrics that need a one-minute increment**,** or greater**,** of granularity. These types of metrics tend to be metrics that require coordination and correlation between several systems in an application. By using scheduled Lambda, you can on a regular interval query multiple AWS metrics or custom metrics and then take a specific action based on your business logic.

For example, you can automatically trigger an SQS notification whenever a queue grows longer than expected, but you might instead want to only trigger an alarm if the queue is over a certain depth and for some reason your EC2 Fleet does not scale properly based on count. With a scheduled function, you can correlate several metrics in order to take a given action, while also being able to do this in a regular interval.

DO NOT DISTRIBUTE

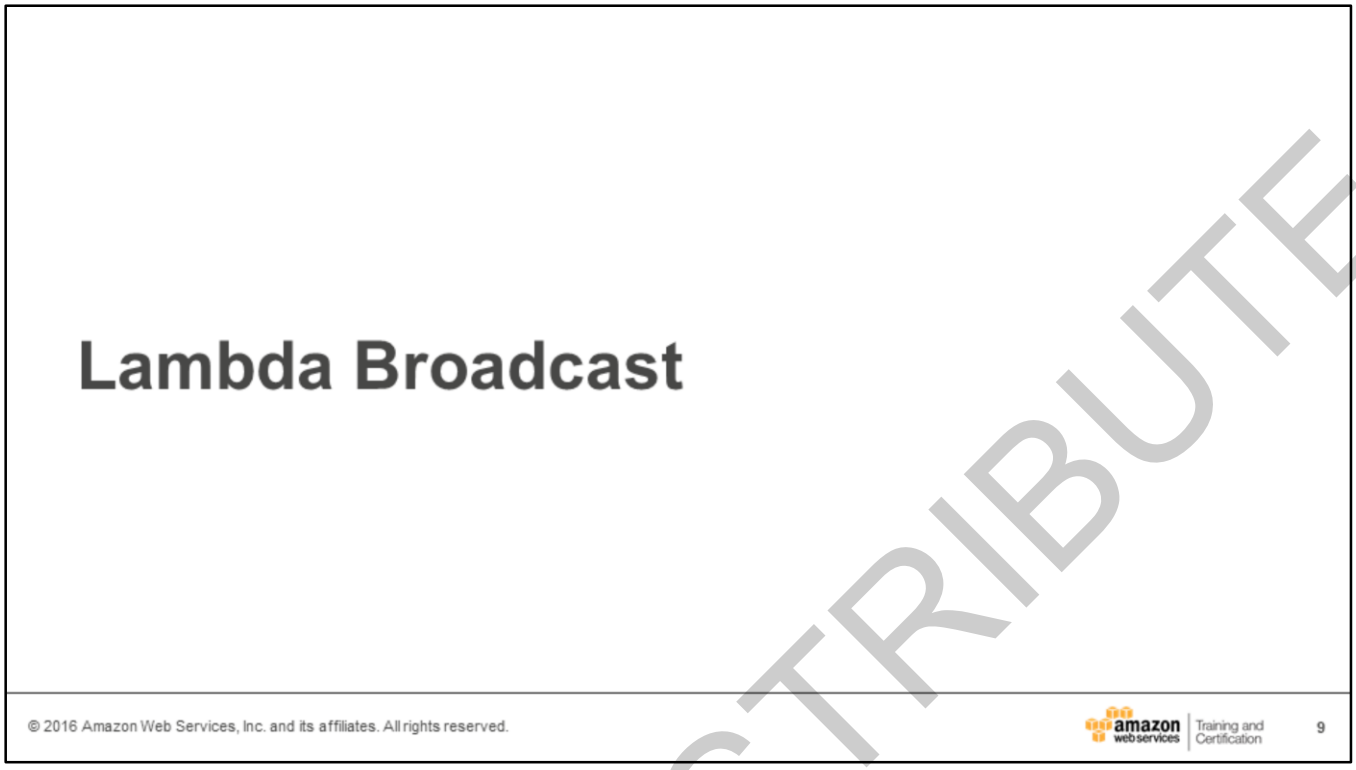
© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



For reporting and analytics as a use case, envision an environment where you have system wide cloudwatch metrics in AWS along with metrics being published from an EC2 instance. In order to correlate these metrics, prior to scheduled Lambdas, you'd be required to poll these metrics with a process running on EC2. With scheduled tasks in AWS Lambda, you can instead invoke a Lambda function that can run your business specific logic and then take the appropriate next steps. In this example, it could publish to an SNS topic that triggers an email notification to your engineers

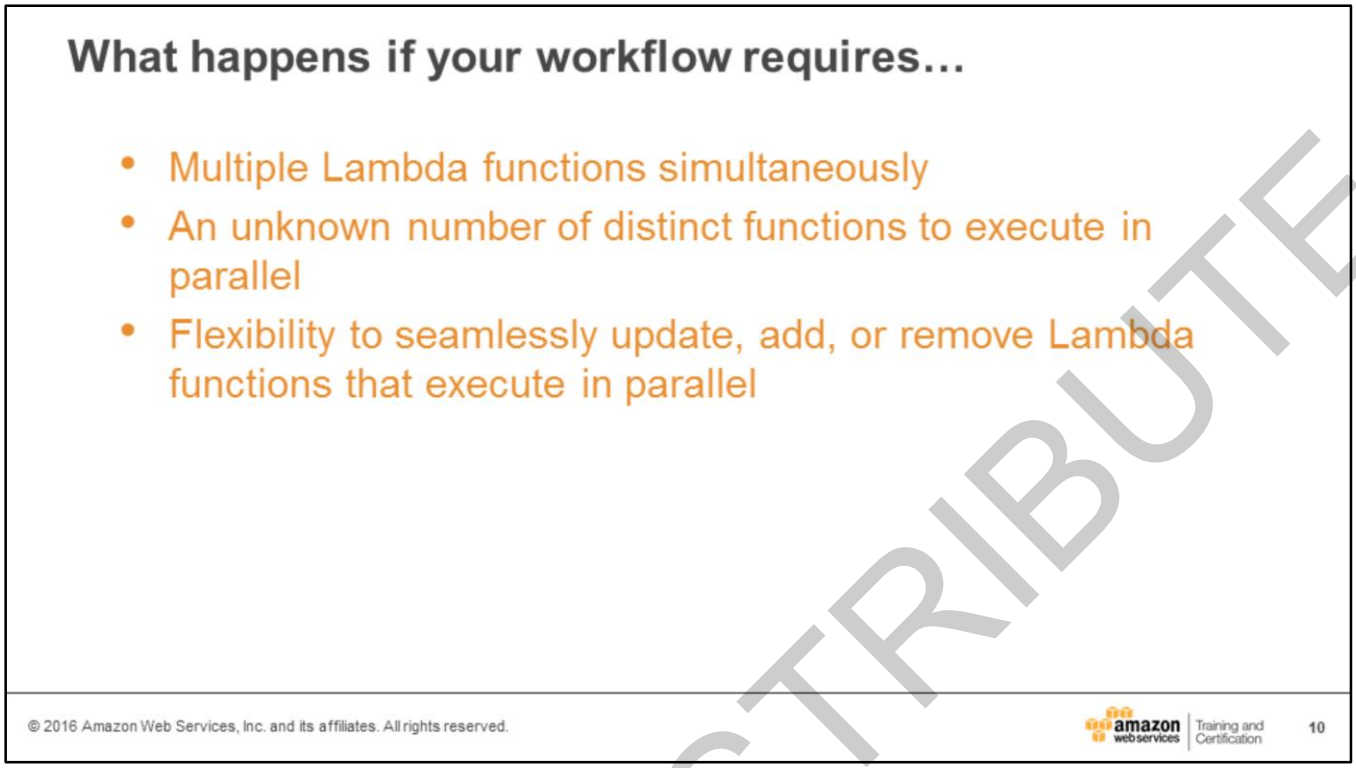
DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



Now that you've learned the concepts of scheduling tasks in Lambda, we will start to take those concepts of decoupling your applications and managing state to dive into other Lambda patterns that are common in serverless architectures.

DO NOT DISTRIBUTE

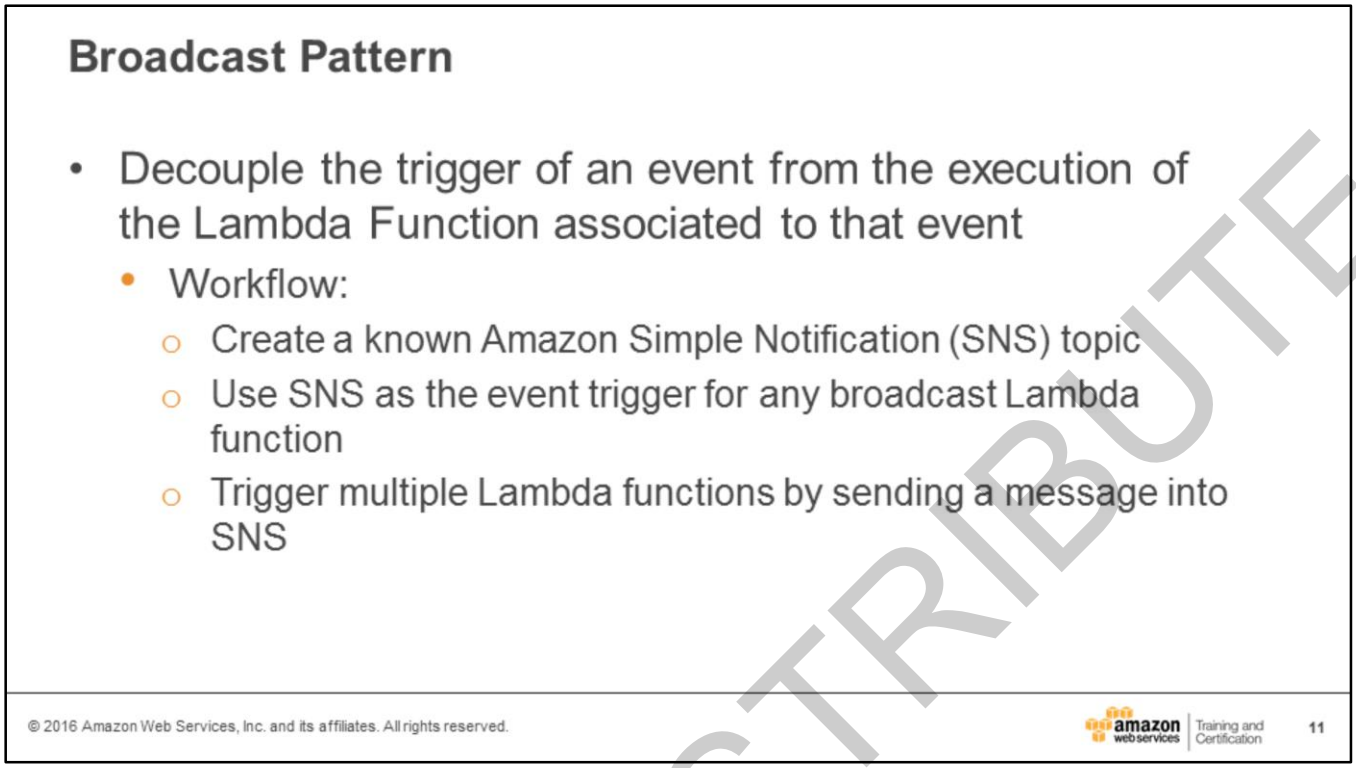


When considering how to execute multiple lambda functions in parallel or sequentially, you should ask yourself some questions such as: what are your workflow requirements? How will your functions execute? In what order? And, how will you handle state changes and storage in that workflow?

In the event that the workflow requirements align with executing functions simultaneously, the ability to add and remove functions seamlessly over time, and the ability to have a non-deterministic number of functions linked together, you should evaluate using the broadcast pattern.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

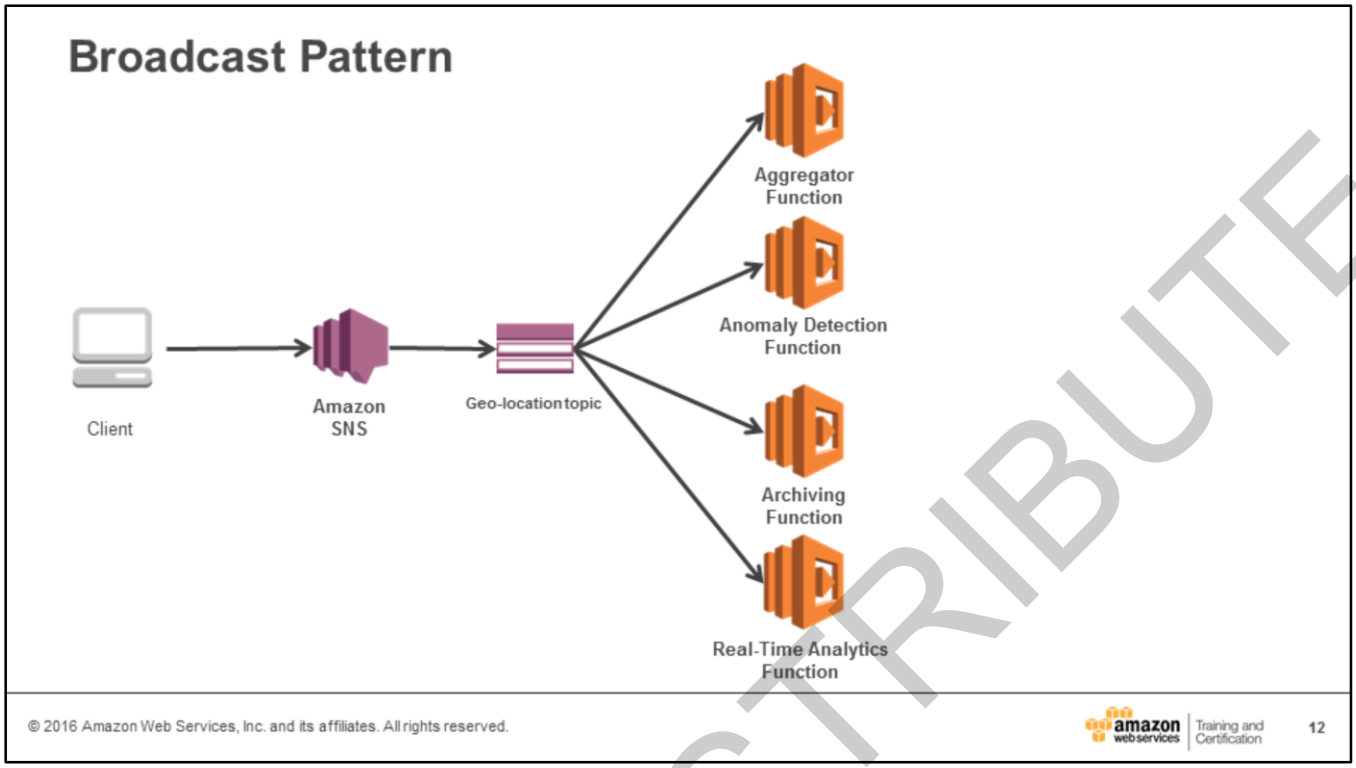


When your workflow has multiple Lambda Functions listening to one event, use the broadcast pattern. Broadcast patterns decouple the event from the multiple functions associated to it.

To implement this type of integration, there a few steps to follow. The first step is creating an SNS topic. SNS works in a publish and subscribe model, where one client can publish to an SNS topic, and SNS supports multiple subscribers that can receive updates on a topic. The next step is to use SNS as the event trigger for one or more Lambda Functions. The ability to have one or more Lambda Functions working independently allows you to add or remove Functions without impacting any Function associated to the same SNS topic. The last step is to send a message to the SNS topic in order to invoke all the associated Lambda Functions.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



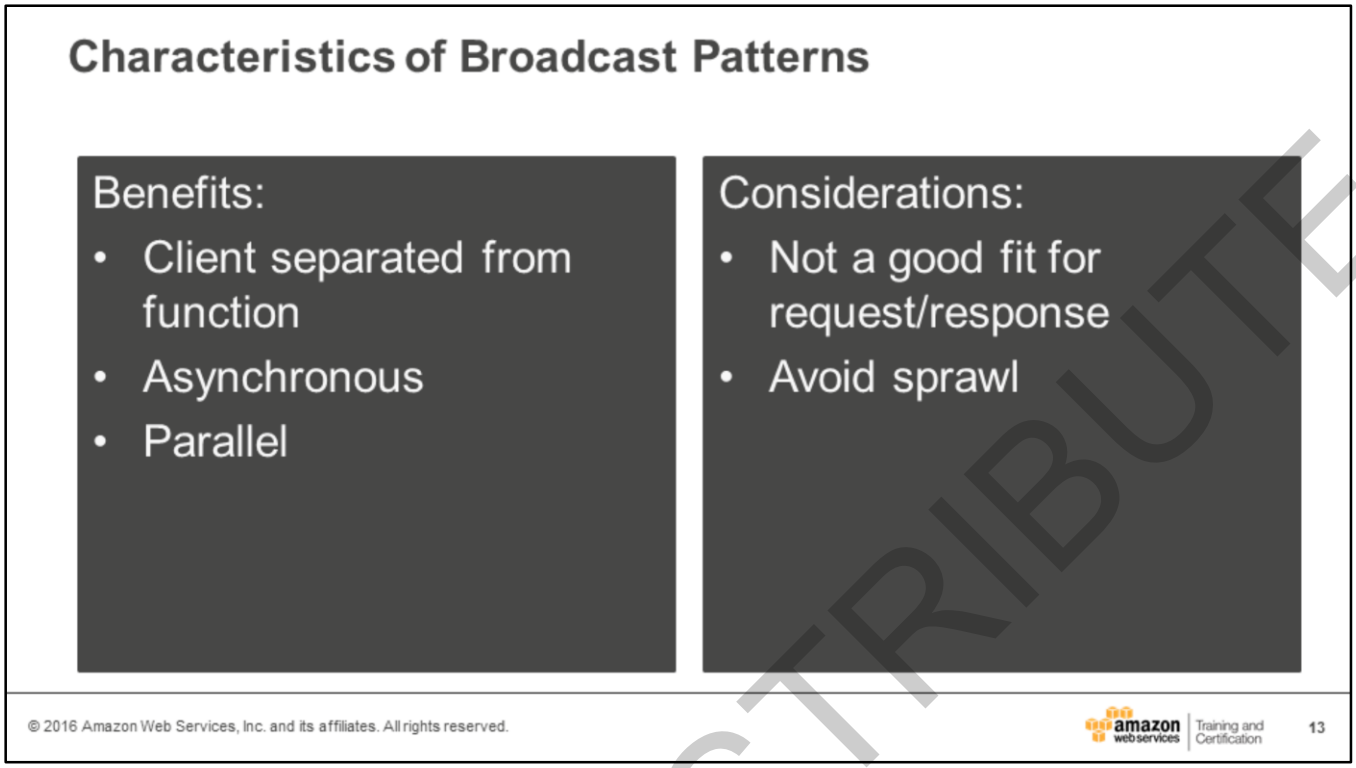
The broadcast pattern is a one-to-many association. The one side of the association is an SNS topic, and the many side is a set of Lambda functions. This one-to-many association creates a flexible integration between one sender and a set of interested operations.

In the example above, a client makes a request that triggers an SNS notification. The notification is pushed onto a specific topic based on the geo location of that browser. The power of the broadcast pattern shines downstream because multiple Lambda functions are all interested in the current location of a specific user. So in this case, three functions all execute on the same data point and take additional actions in parallel.

Because the end client does not execute Lambda directly, it's also straightforward to add a new function over time without impacting the behavior of other Lambda functions on the same topic.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



What are the benefits?

* Decouple the client calls from the lambda execution, giving you flexibility to change Lambda logic later on without impacting the client.

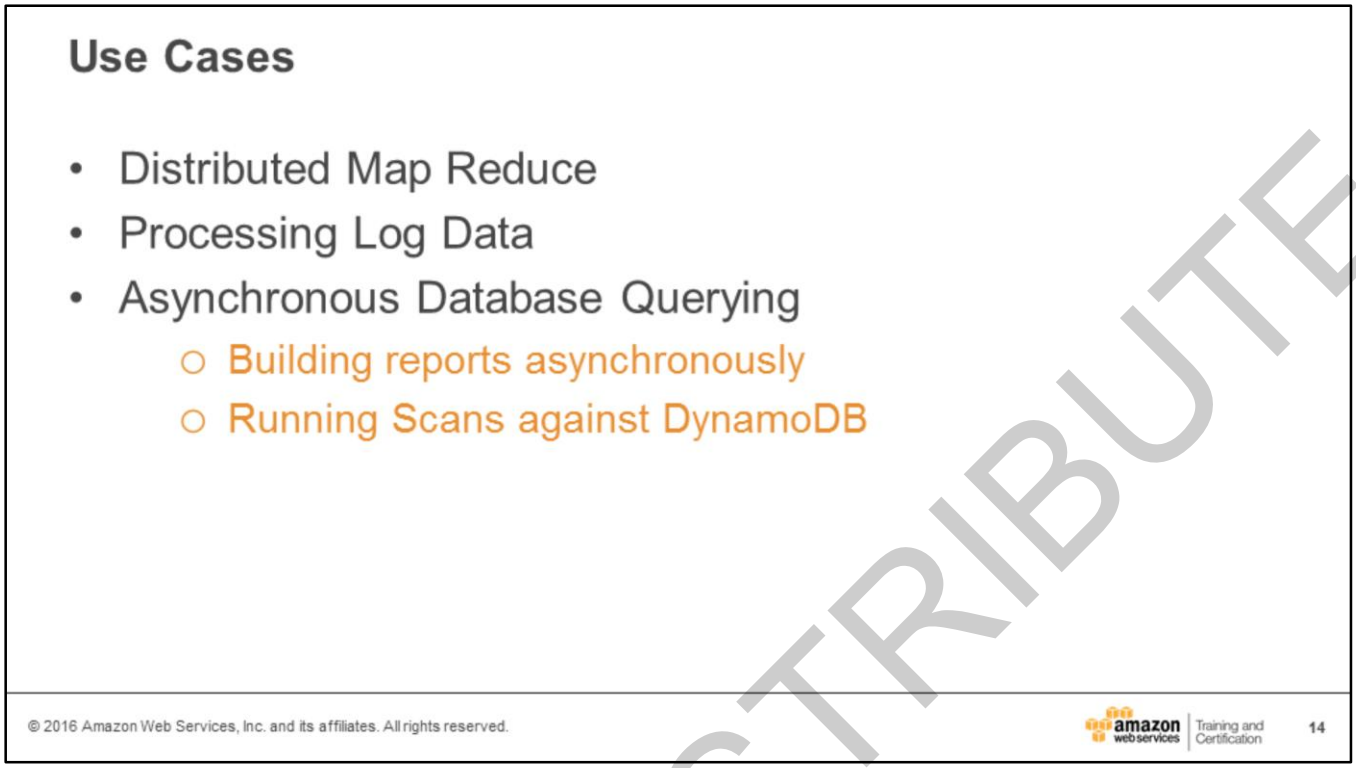
- This workflow is ideal for asynchronous communication because the Lambda functions cannot return a response to the client synchronously.
- Opportunity to have one event trigger multiple functions in parallel, lowering the total end-to-end time of a specific set of Lambda functions

What are things you should know ahead of time when deciding to use the broadcast pattern?

- The broadcast pattern does not follow a request and response workflow and cannot be synchronous because the client does not receive a response directly from Lambda.
- Although you can have a large number of subscribers to an SNS topic, avoid having very wide funnels of Lambda functions associated to one SNS topic. Instead, group similar parallel processes together to one SNS topic and create more SNS topics as a way to avoid sprawl of functions that are loosely related to one another but all listening on the same topic.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

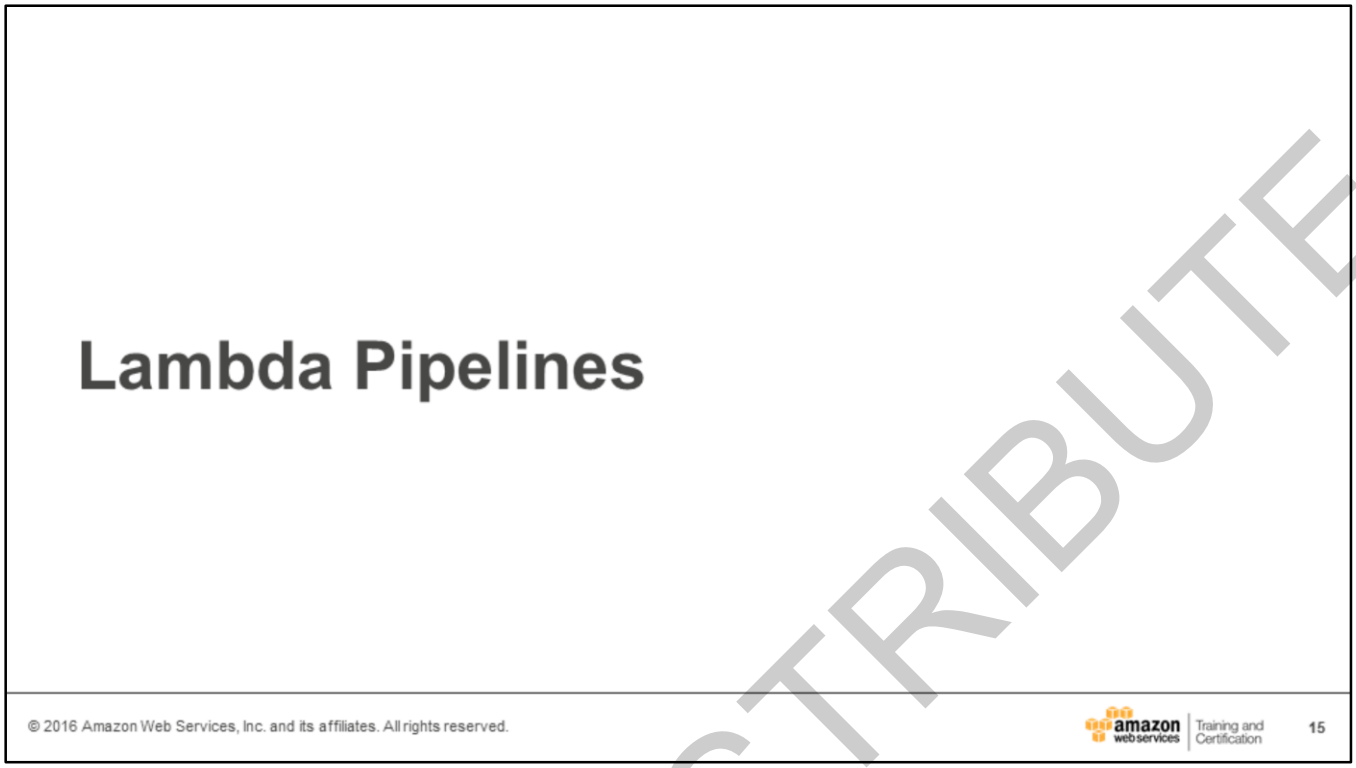


What are some common use cases for using broadcast patterns in Lambda? Here's a list of examples to get you started.

- Distributed Map Reduce – Use data stored in a system such as S3 to trigger a notification to SNS and then to a set of Lambda functions to perform mapping and reducing of that data in parallel.
- When performing parallel scans against a database such as DynamoDB, you can use a broadcast pattern and then in parallel Lambda functions can scan across the table and build different segments of a report.

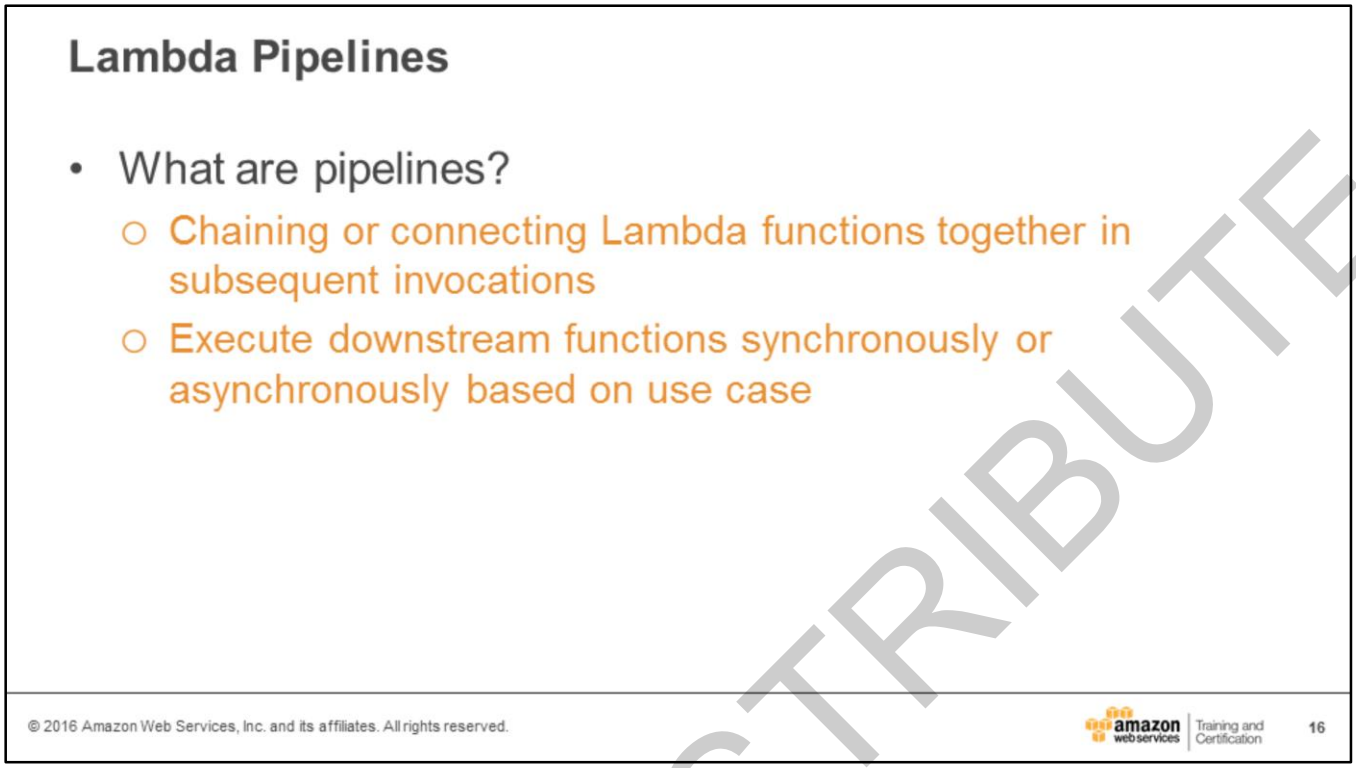
DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



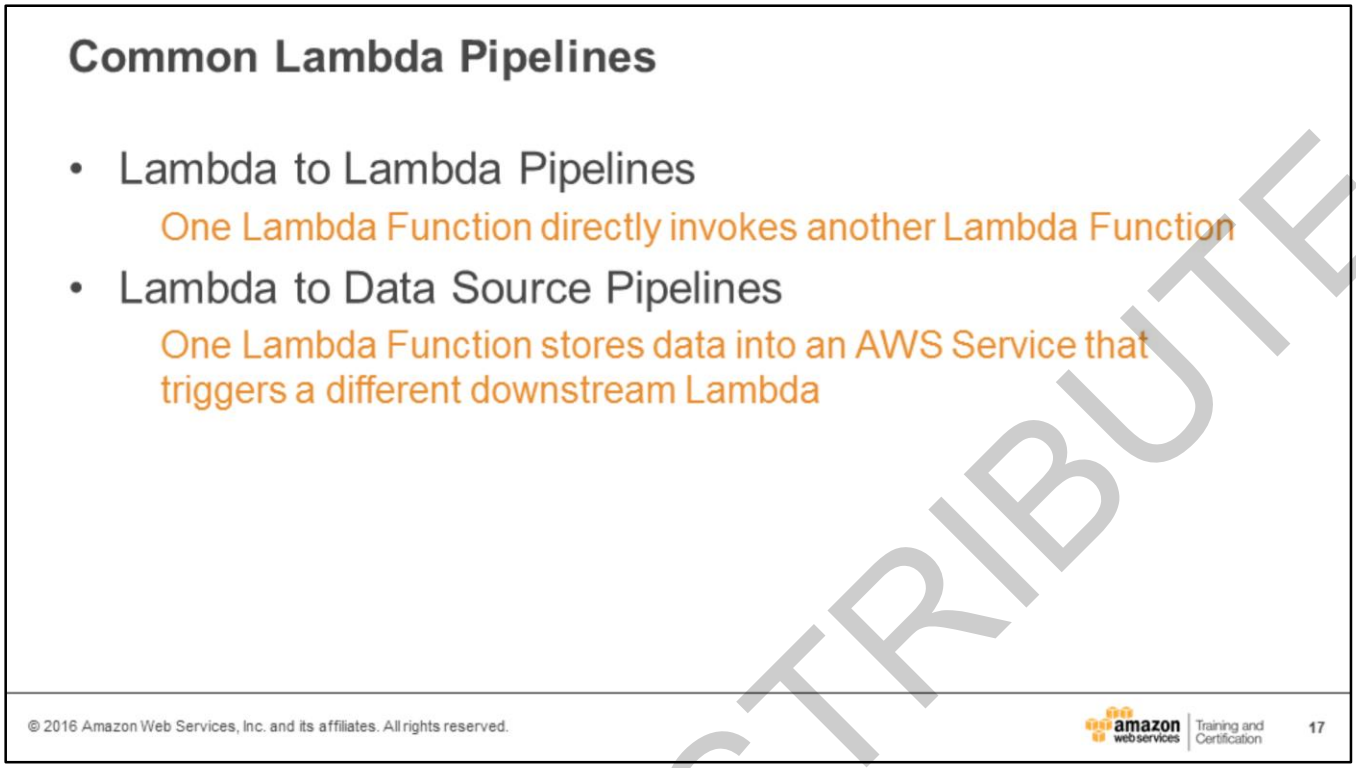
After you configure scheduled Lambda Tasks and broadcast patterns, you can look at how those types of operations can be part of a longer transaction between Lambda functions. In Lambda pipelines, we'll discuss how you can combine and link functions together.

DO NOT DISTRIBUTE



Pipelines are an architecture pattern where one Lambda function directly or indirectly triggers or calls another lambda function. Connecting Lambda functions directly creates the ability to bundle together multiple lambda functions as individual microservices, each responsible for specific domains in an overall architecture.

DO NOT DISTRIBUTE



There are two ways of combining and connecting Lambda functions together. The first way is for one Lambda function to directly invoke another Lambda function. In this scenario, the primary function can either synchronously invoke it's child function and wait for a response or asynchronously invoke the child function passing in a payload that can be executed downstream.

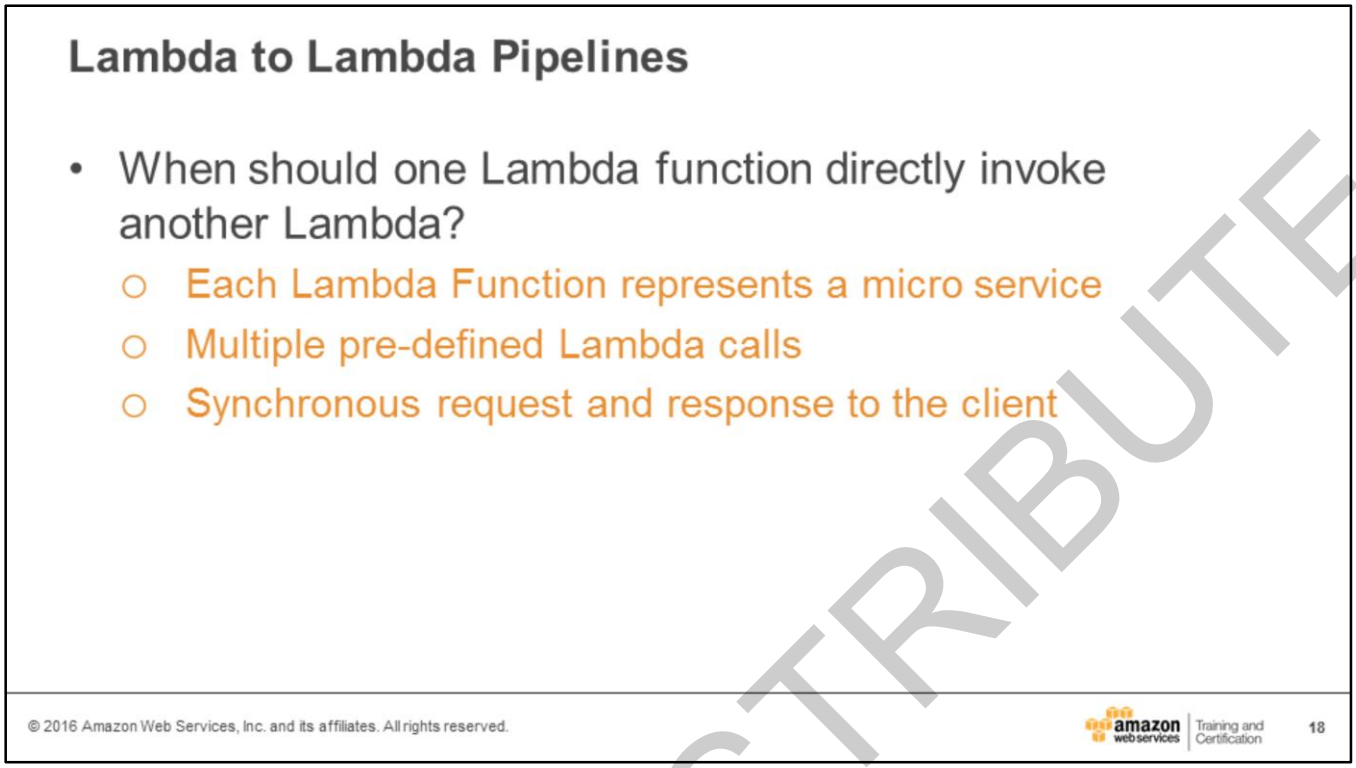
For example, when polling from an SQS queue in Lambda, instead of having the main function poll records off SQS and run the logic to process that record, you can create a pipeline where the main function polls records from SQS and then asynchronously invokes a different Lambda whose job is to process a record and delete it from the queue.

The other method for creating a lambda pipeline is using Lambda in addition to other AWS Services that are event triggers for Lambda. In this scenario, the individual data sources are the connective tissue between function invocations. By storing data in data sources between Lambda functions, you can create workflows that incorporate intermediary data changes.

For example, you may use a service such as S3 and store slightly modified data sets between Lambda invocations. These new data sets can trigger a second Lambda invocation that will further modify files and upload them to either another S3 bucket or a different AWS service.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

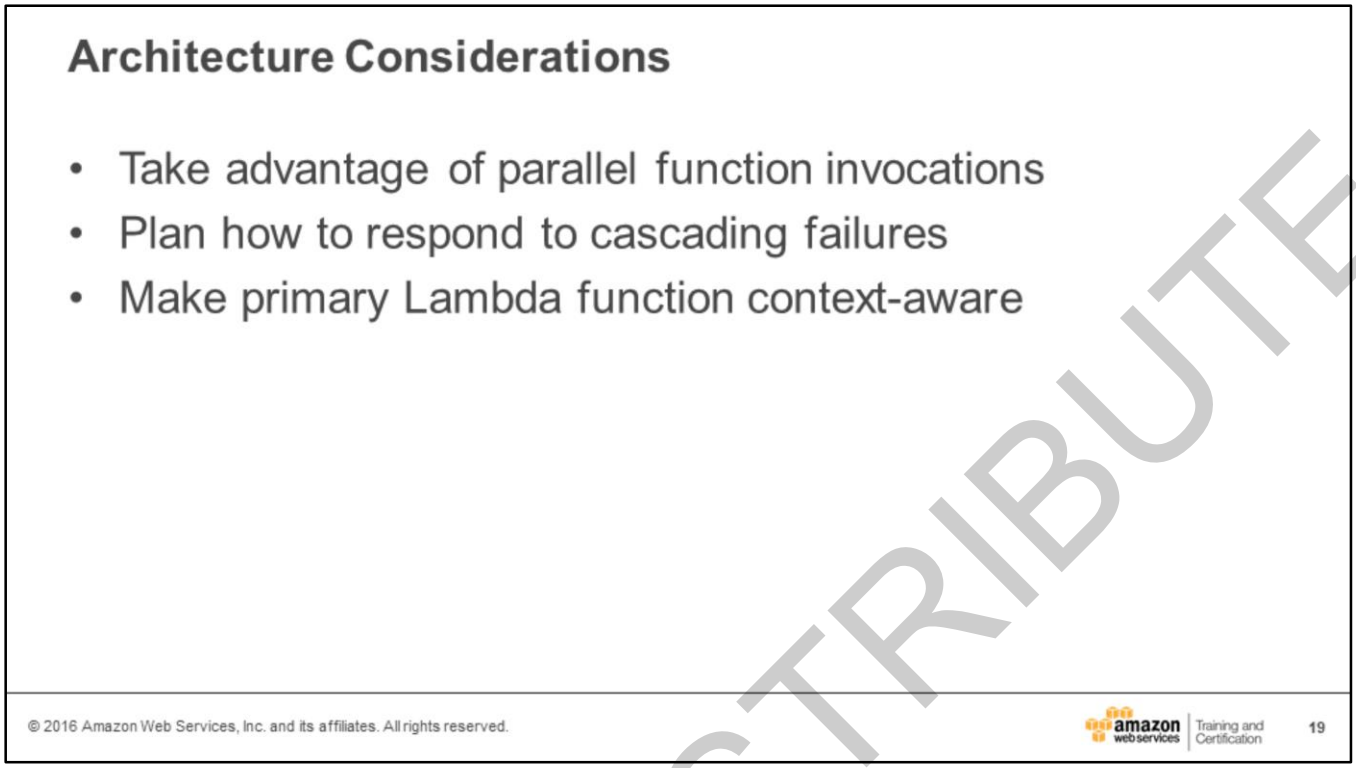


As you've seen so far, you'll have several options for connecting Lambda functions together as you build your architecture. When connecting multiple Lambda functions together, a common pattern is having one Lambda function explicitly invoke a second Lambda function. There are specific scenarios and architecture characteristics that make this an ideal choice for your application.

If the Lambda Functions are tightly wrapped as micro services, you can explicitly call a Lambda function that represents a concrete set of functionality. Similar to the broadcast pattern, there are times where you will have a one-to-many relationship, but you have a preset of downstreams functions that need to execute serially. When there's a known workflow of common functions, an upstream Lambda can orchestrate calling those Functions directly in order to execute them. The last scenario that is common for direct Lambda-to-Lambda invocation is when your application requires a synchronous response from a second function before it can continue. When using other patterns such as pipelines, the Lambda functions are invoked asynchronously. But, in cases where an upstream client or application is waiting for a response, you can use synchronous invocation to aggregate downstream data from one or more Lambda functions before responding to the client.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

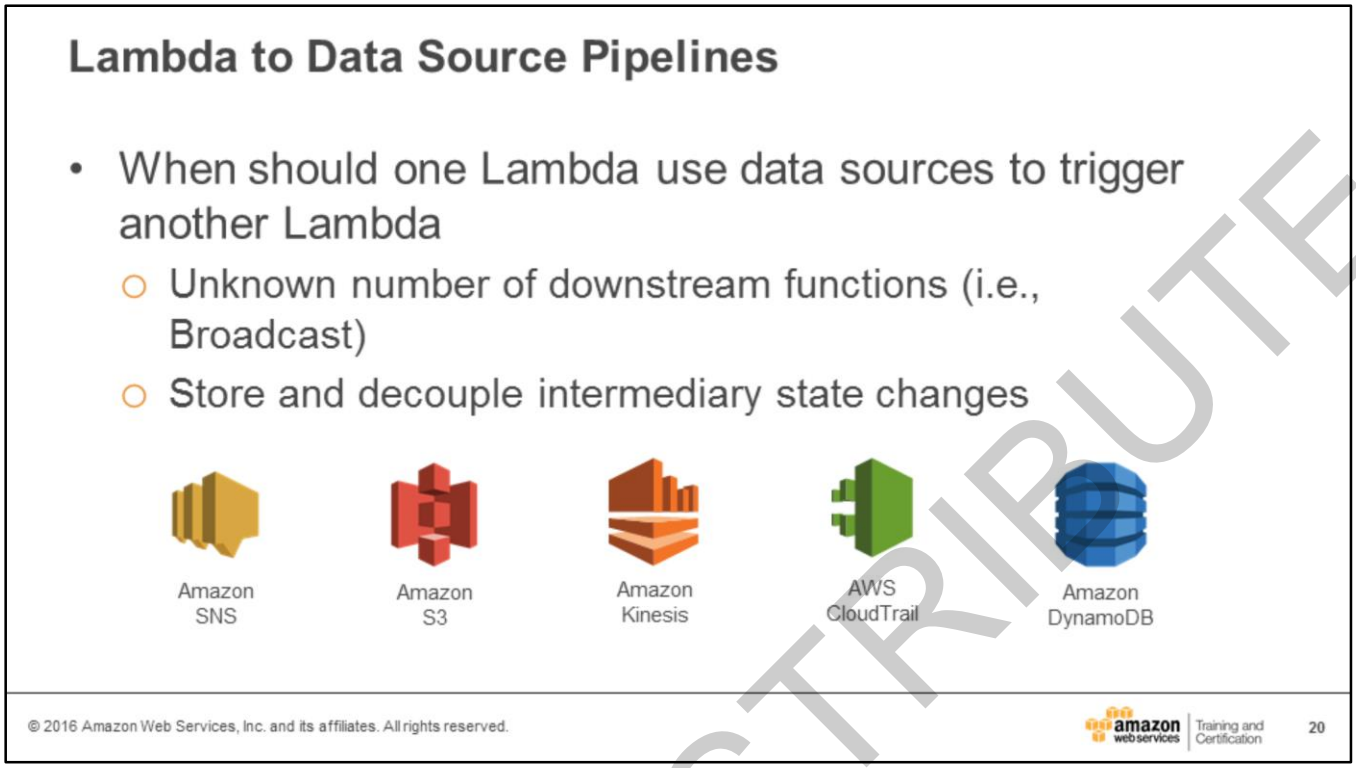


We've seen how using multiple Lambda functions as a unit can help increase the flexibility and create an ecosystem that fosters a separation of concern across different Lambda components. When building pipelines in production, there are several considerations to evaluate as they relate to performance, operations, and monitoring.

- The first recommendation is to look for opportunities to run functions in parallel over smaller segments of data. Unlike running a set number of EC2 instances, you can run multiple functions to decrease the total amount of time it takes to process a request. If you are using a Lambda-to-Lambda pipeline that needs to be synchronous, use reactive coding patterns such as Promises so your primary function can continue to perform different tasks while waiting for downstream functions to complete.
- It is a general AWS best practice to plan for failure, and this applies to working with Lambda. Determine what you want the expected behavior to be when a data source or Lambda function fails to execute properly. For example, if a DynamoDB table is throttled should your Lambda function retry the query? If so, for how long? Do you expect to handle Lambda functions so that they do not time out by returning a default value? Or, is it better in your workflow to have the function timeout and return an exception? These questions will help you to consider which data sources align most with your operational requirements and how to add the appropriate logic from end to end.
- When one function calls a second function, your primary is responsible for keeping track of context. This means it has an idea of which functions to call, whether functions should return in a given time period, and what types of responses to expect. The primary function is the aggregation and orchestrator for all Lambda requests downstream.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



- We discussed the ability for one Lambda function to immediately invoke other Lambda functions to create a direct chaining effect.
- We also discussed one type of data source for Lambda, SNS, which is a great way to create a broadcast pattern of Lambda Functions to one event source. But, there are other use cases and event sources that can be used to chain lambda together.
- This is not an all encompassing list of data sources, but to touch on a few: Amazon S3, Amazon Kinesis, AWS CloudTrail, and AmazonDB.
- When using a data source to then trigger a Lambda event, at a high level you gain the ability to decouple data from business logic in addition to being able to add more operational inspection by storing the individual transformations that happen throughout a pipeline.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

Use Cases for Lambda to Data Source Pipelines

Transformations

- Multiple Transformations
  - Media file formats
  - ETL

© 2016 Amazon Web Services, Inc. and its affiliates. All rights reserved.

amazon web services | Training and Certification 21

What are some use cases for using different event sources to trigger Lambda Functions?

- One use case would be to transform media files, but into multiple formats. For example, you use a combination of S3 and DynamoDB to not only store location of a file and information about it's format, but also to create several versions of a file. You might have an image file you want to transform into a thumbnail and a portrait still, so you'll execute multiple Lambda functions and store the output into different S3 buckets, each triggering a notification to Lambda and inserting into DynamoDB.
- Another use case is creating ETL jobs that normally require several forms of data manipulation. Files can be stored in S3, but then moved to other AWS data stores for intermediary formats such as in DynamoDB or in a structured database such as RDS.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

Use Cases for Lambda to Data Sources Pipelines

Synchronizing Data Stores

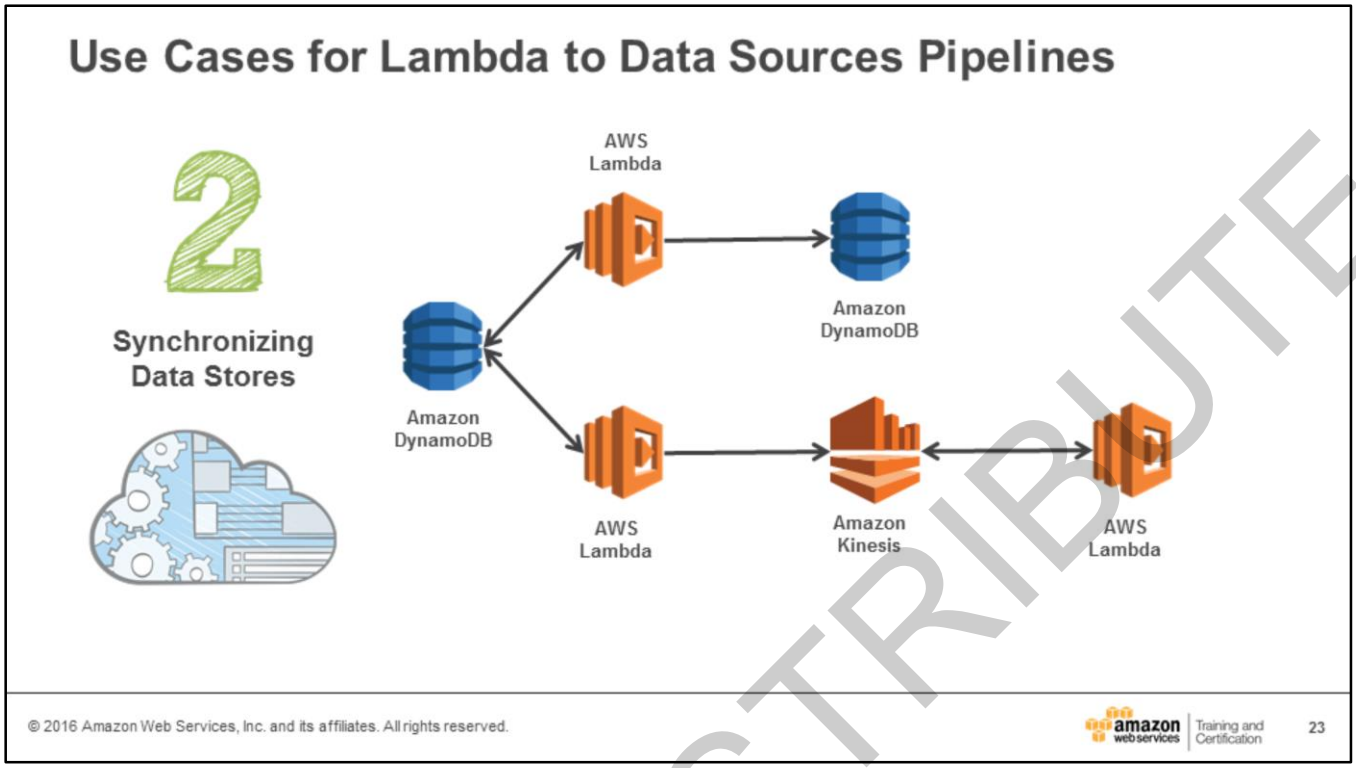
- Synchronizing Data
  - Store data in multiple data engines asynchronously
  - Create read replicas of data or real time backups

© 2016 Amazon Web Services, Inc. and its affiliates. All rights reserved.

- A common use case for using data sources as triggers for Lambda functions is in conjunction with streaming changes from one data source to another. A common use case would be using DynamoDB Streams to migrate data changes from one database to a second database either in the same format to act as a read replica or process the data in a different format to create pre-aggregated analytics for faster querying.
- The ability to connect Lambda functions through streams such as DynamoDB Streams or Kinesis create a mechanism for near real-time data replication between multiple applications.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



How can we start using Data Sources to connect our Lambda functions together?

- Let's start with a DynamoDB table that stores some data, for example, user data.
- By enabling DynamoDB Streams, you can attach multiple Lambda functions that will read all of the changes coming off of the stream, and use this stream to move data in our application.
- The first Lambda function will replicate the data from one DyanmoDB Table to a second table to support a read replica for external uses.
- The second Lambda Function will aggregate data and immediately store it into a Kinesis Stream for other downstream consumers to use.
- This Kinesis stream can be subscribed to by other AWS Lambda functions and moved to other data sources (such as RDS, S3 to Redshift, or an internal database running on EC2).

DO NOT DISTRIBUTE

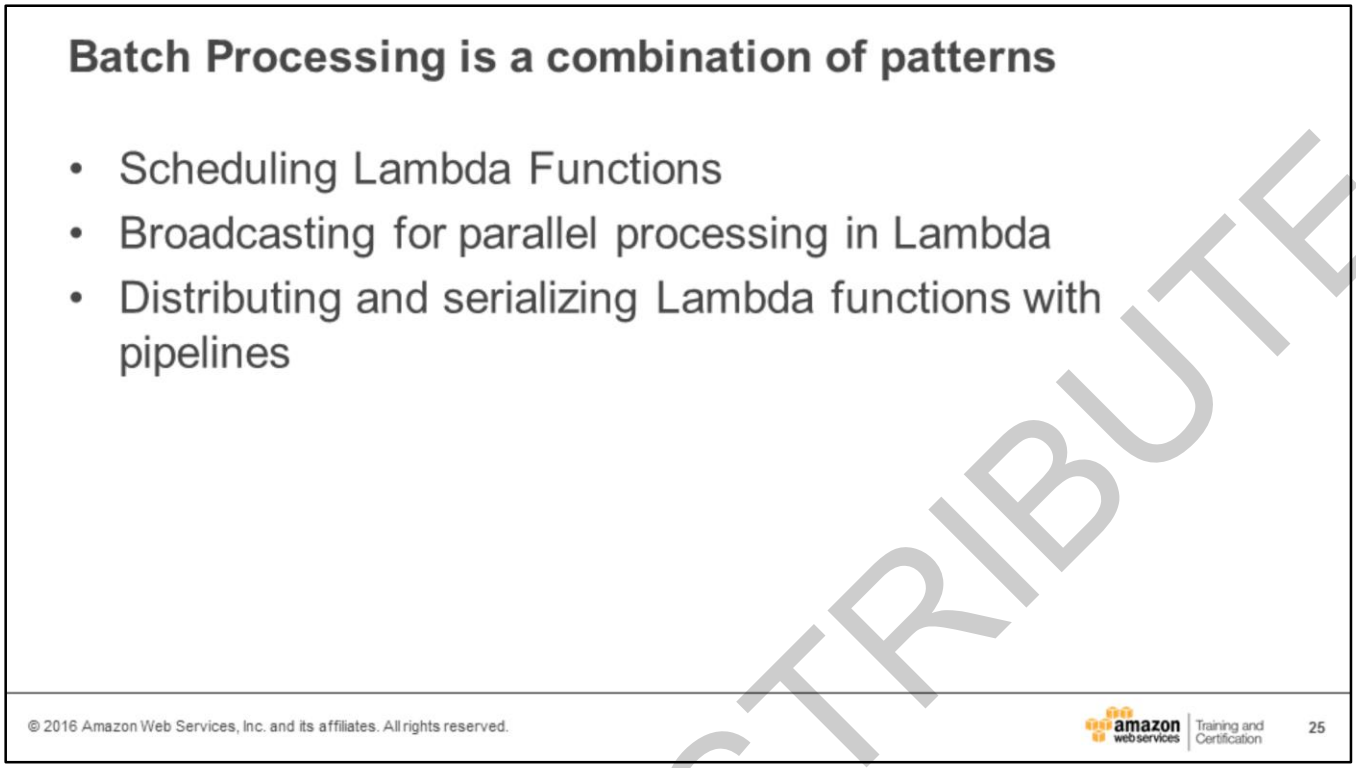
© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

Batch Processing with Lambda

© 2016 Amazon Web Services, Inc. and its affiliates. All rights reserved.

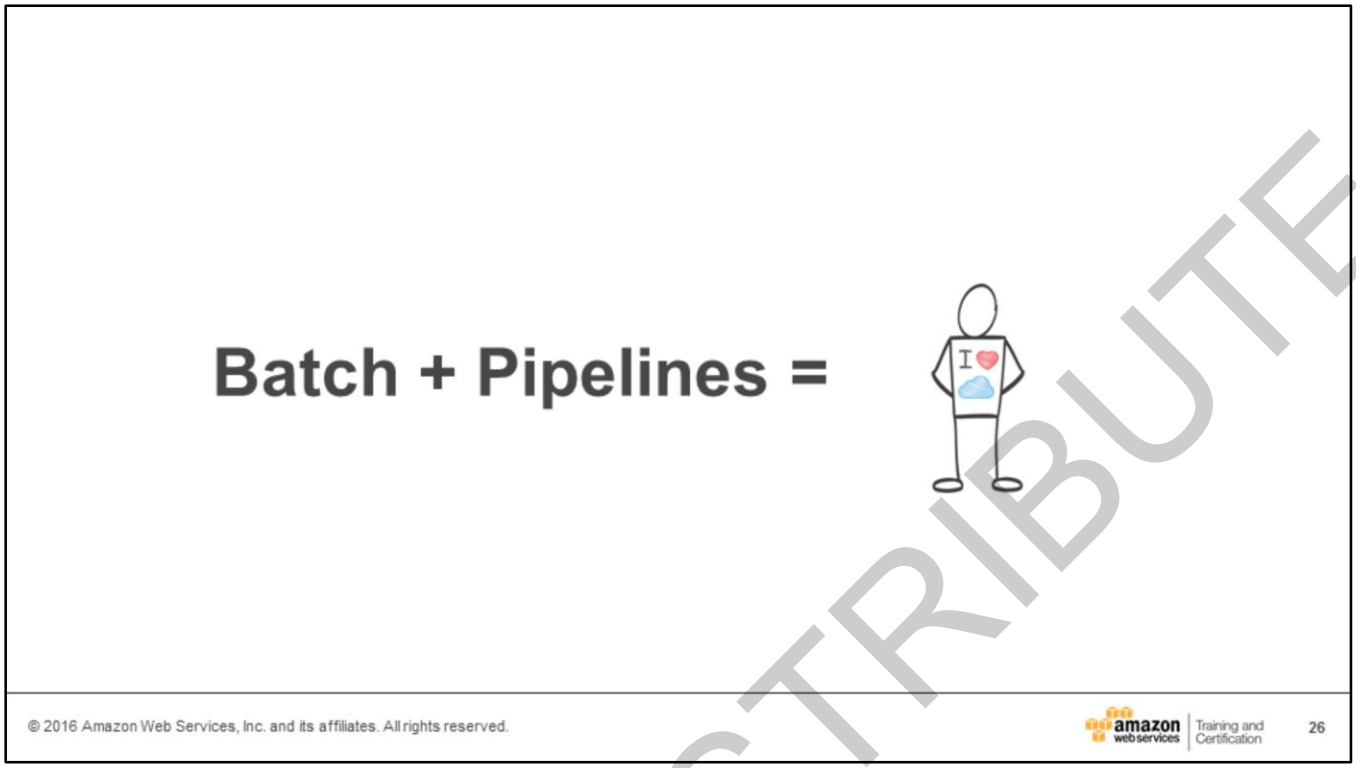
DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



So, you've configured scheduled Lambda Functions, then discussed a broadcast pattern to run Lambda functions in parallel, and now you've added onto your options by discussing how you can use other AWS Services as a way to build pipelines between your business logic running in Lambda. With all of these tools together, you can start batch processing data using Lambda.

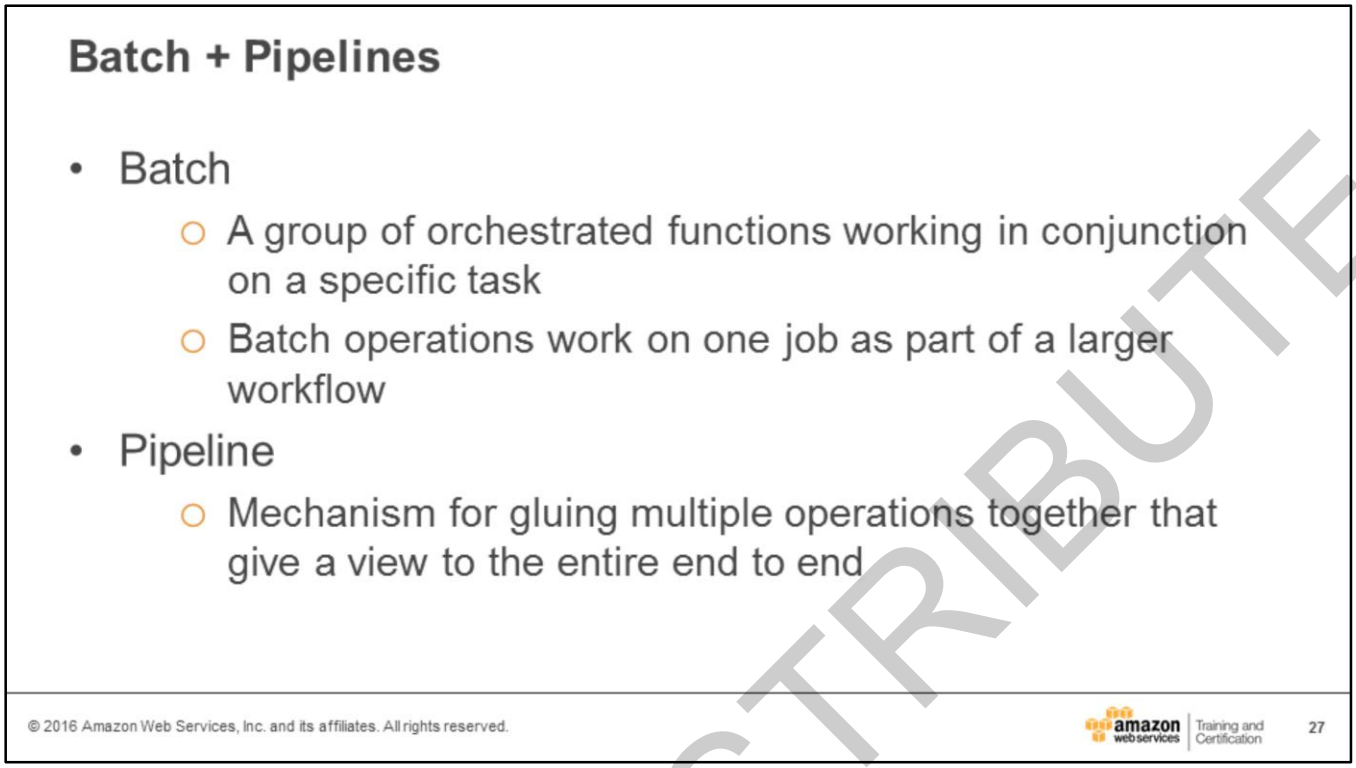
DO NOT DISTRIBUTE



By combining batch processing and grouping them into your larger pipeline workflow, you can build rich, distributed systems all running on AWS Lambda**.**

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

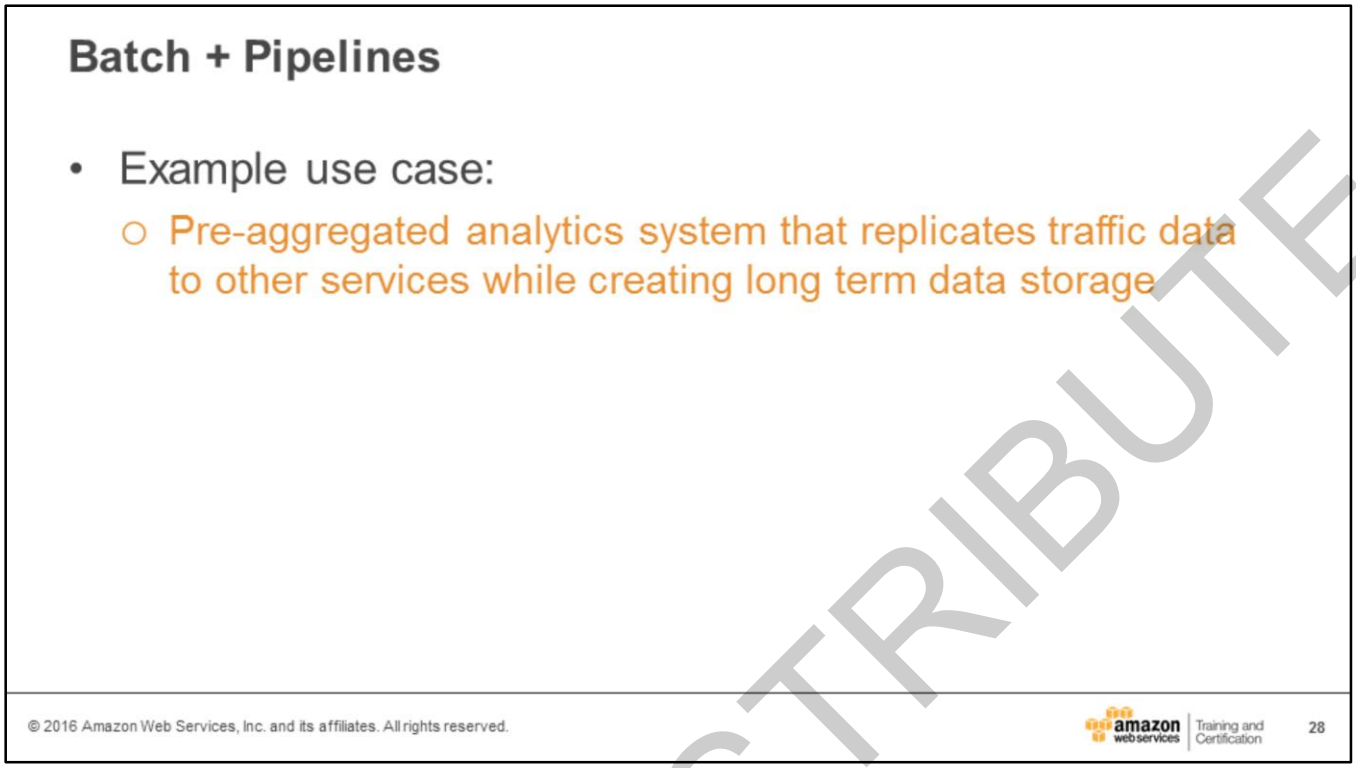


* Batch and pipelines are grouped into two segments. Batch functions can be seen as primarily parallel operations that scale out in order to complete a portion of work. Batch functions need to store intermediary state because they process large transactions or sum of data. Batch processes use other AWS services such as SNS to broadcast out data in parallel along with services for storage such as DynamoDB to track the progress of a long-running job.

* Utilize pipelines as a way to organize multiple individual discrete units of work. A pipeline can be one lambda function that is one step in a workflow, or it can be multiple lambda functions treated as an aggregate.

DO NOT DISTRIBUTE

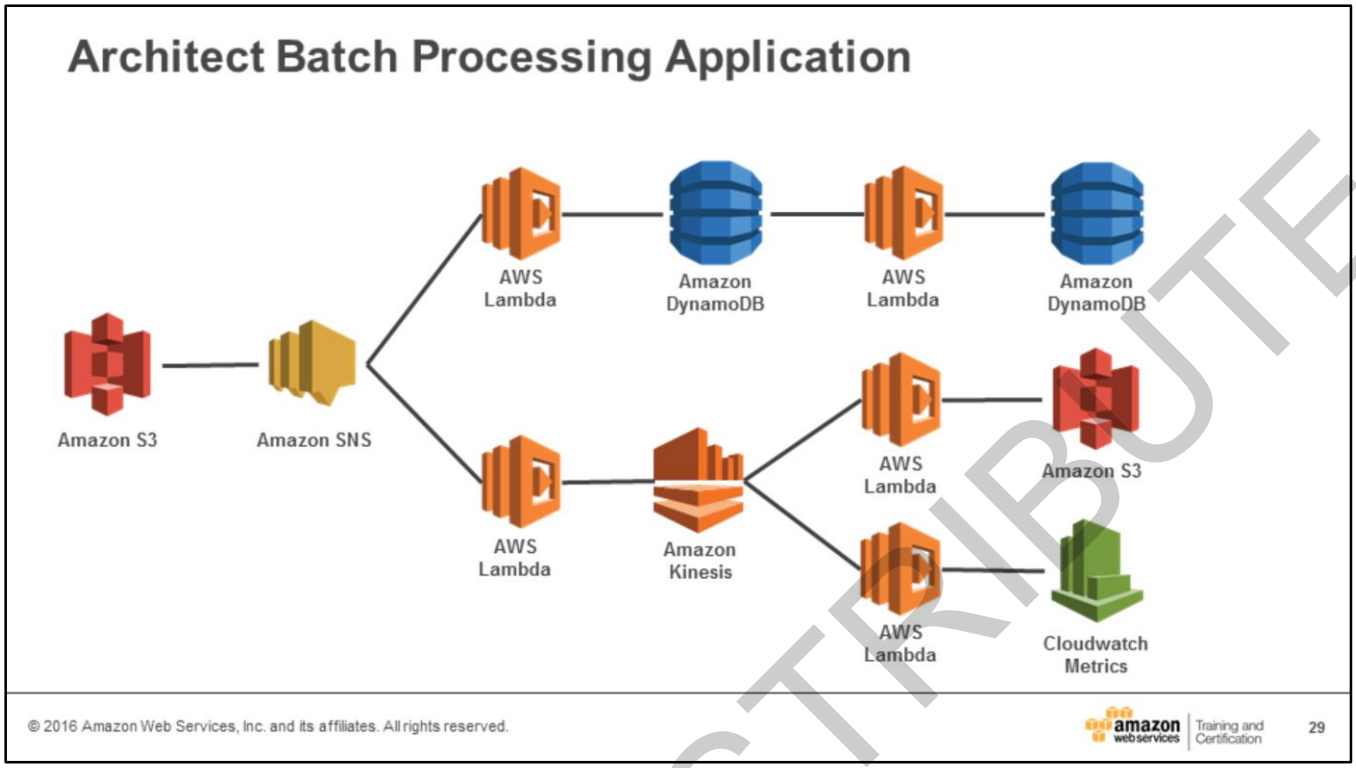
© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



Here's an an example to show all three patterns working together and illustrate how you can use leverage batch, pipelines, and broadcast patterns together in an application. For example, we'll use an analytics system that replicates traffic data to several other AWS services.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



In our analytics process, the first step is storing the unprocessed analytics on S3. After these values are stored on S3, use SNS to trigger a broadcast pattern. Use a broadcast pattern because you want multiple parallel functions to operate independently from each other when the unstructured data is uploaded into S3.

* The first Lambda function formats this data in a semi-structured schema in DynamoDB. By storing it in DynamoDB, you can reference a link to the unstructured file in S3 for a data lake. Additionally, this data can also store the primary fields for querying in DynamoDB.

* In the next step, enable streams of all the changes in DynamoDB so you can aggregate those changes into another DyamoDB table. The purpose of formatting this data into multiple databases is to create additional data formats for quicker data access. The first query pattern makes it easier to find the original unstructured data along with pulling out relevant analytics fields to query directly against DynamoDB. The second table holds short term analytics numbers (such as a time series of each specific type of traffic data – accidents, traffic stops, etc.).

- The second Lambda function associated with the broadcast pattern is responsible for creating another set of parallel functions. You could have used the original broadcast pattern, but in this case you are treating the stream as a separate set of operations that should happen as a unique function. Envision this as a separate team working on a different set of services as part of the traffic data workstream.
- The first Lambda function reads the metadata of the S3 bucket from the stream, retrieves the original file from S3 and now reformats the data so it can be stored in Kinesis firehose for batch process analytics into other services like RedShift.

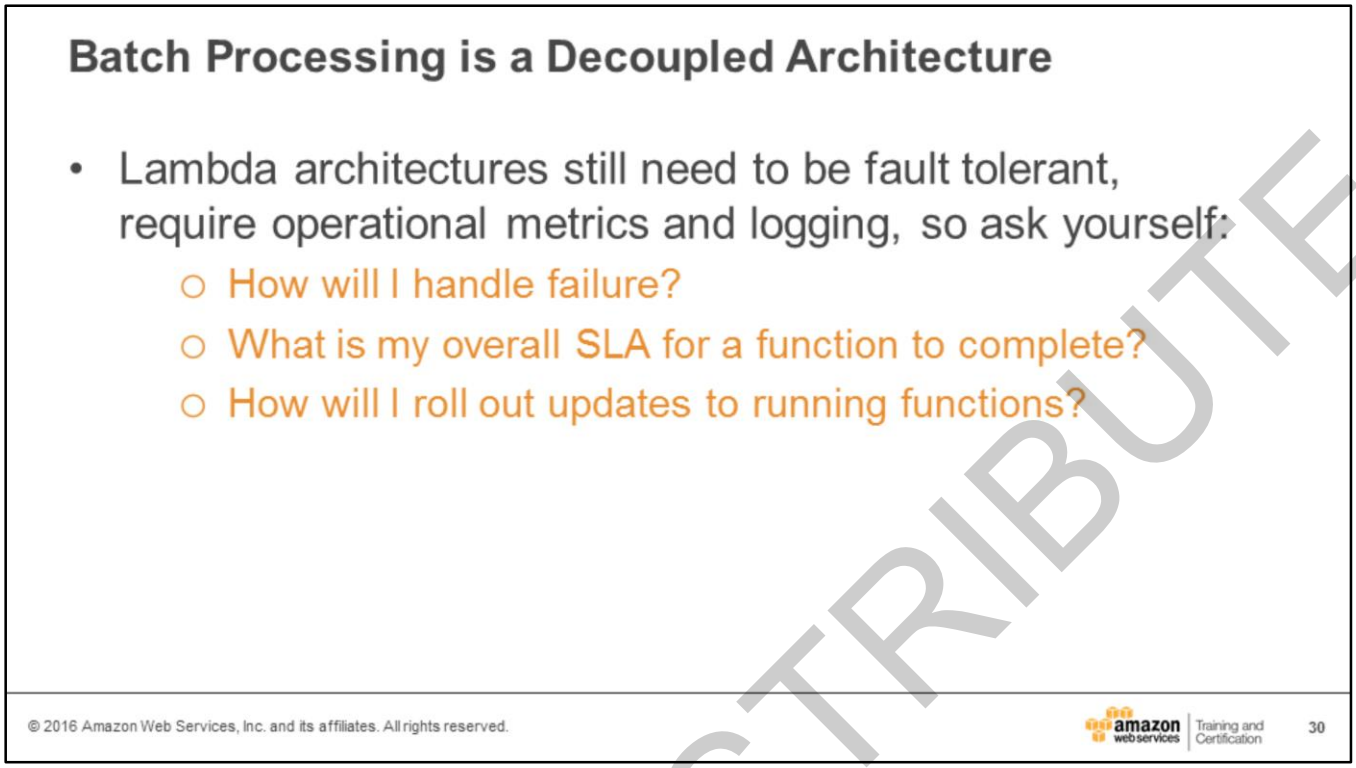
DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

- The second Lambda function batches the metrics of files from S3 and puts a custom metric to track the total number of traffic points added to the system.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



In a complex system where each function is stateless, consider how you will operate your applications and what metrics are important to track throughout a workflow.

DO NOT DISTRIBUTE

Batch Processing is a Decoupled Architecture

- How will I handle failure?
  - Dead-letter queue
  - Monitors and alarms
- What is my overall SLA for a function to complete?
  - Decide to run functions in parallel or serial
- How will I roll out updates to running functions?
  - Decide where functions can be decoupled

© 2016 Amazon Web Services, Inc. and its affiliates. All rights reserved.

DO NOT DISTRIBUTE

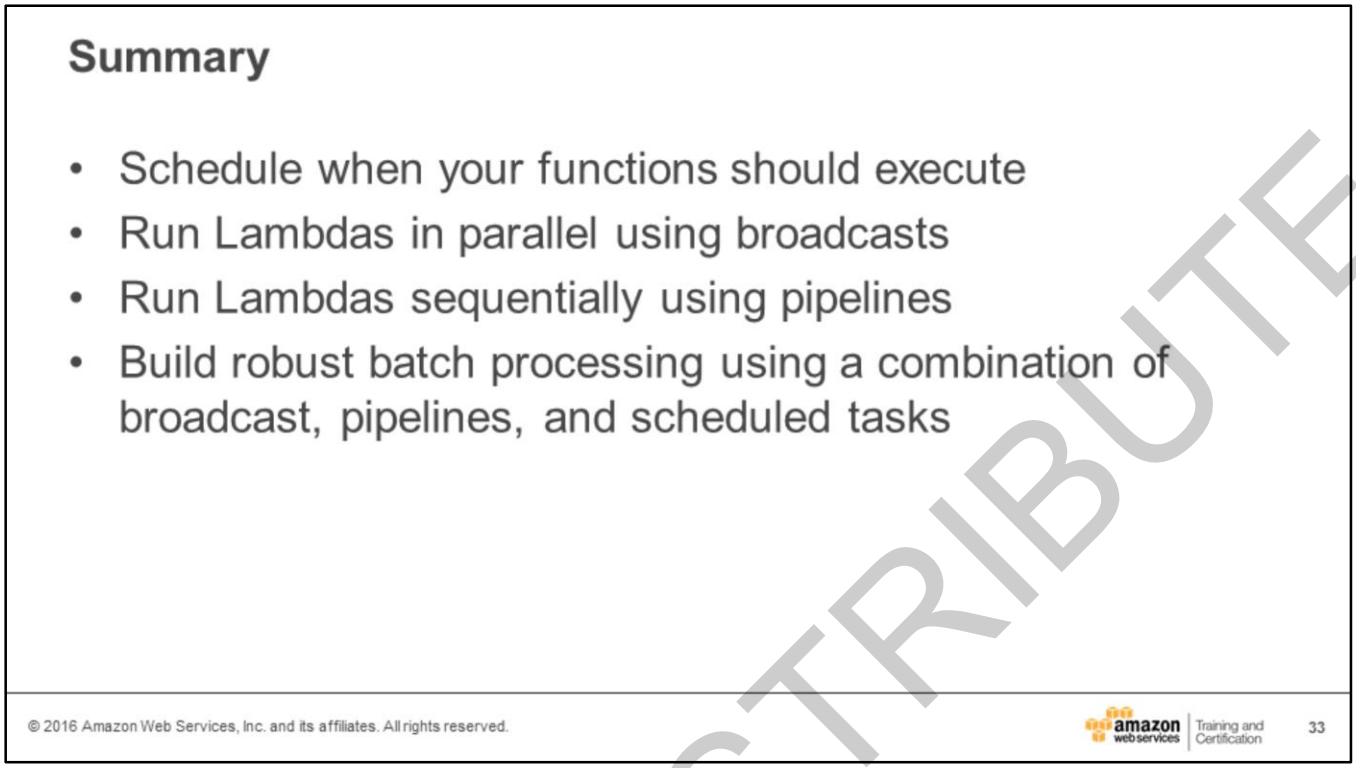
© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

Summary

© 2016 Amazon Web Services, Inc. and its affiliates. All rights reserved.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

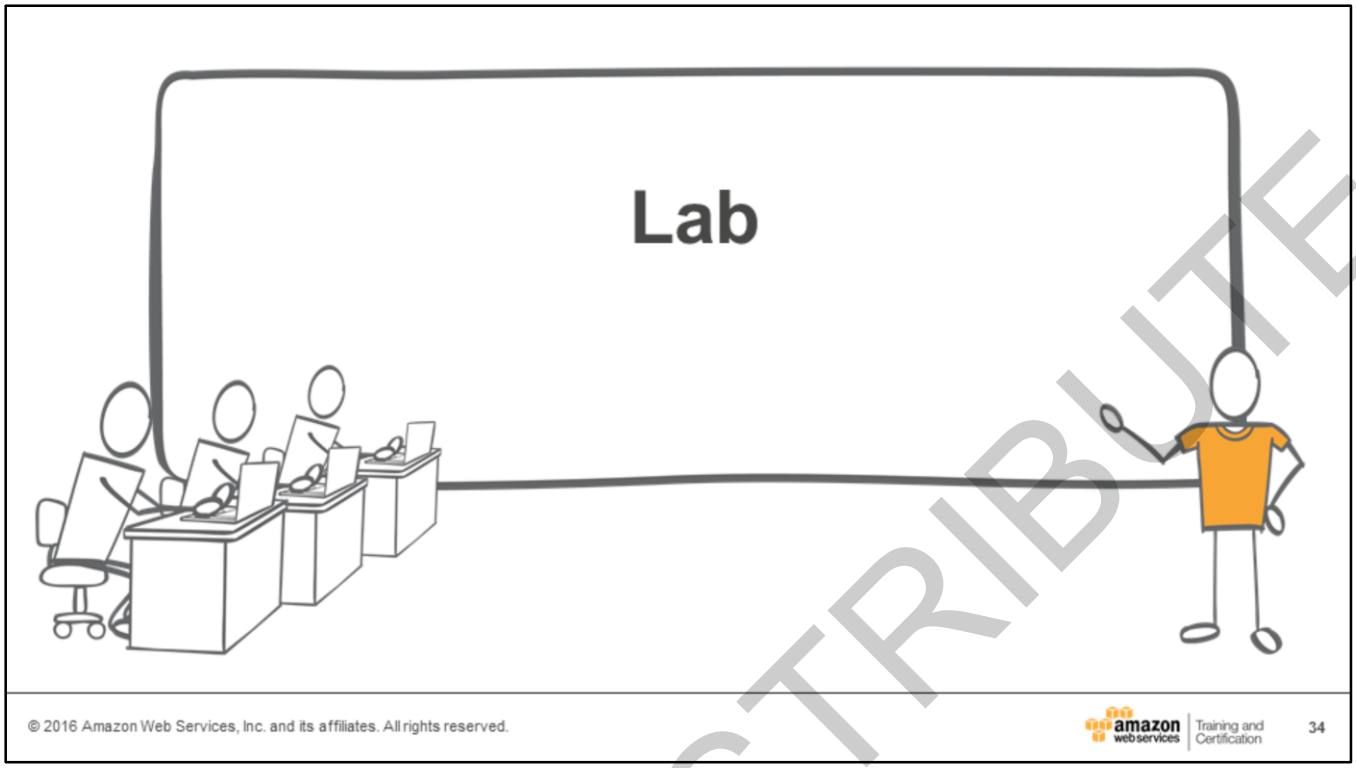


So, what have we covered in terms of the patterns and architectures that you can build in Lambda?

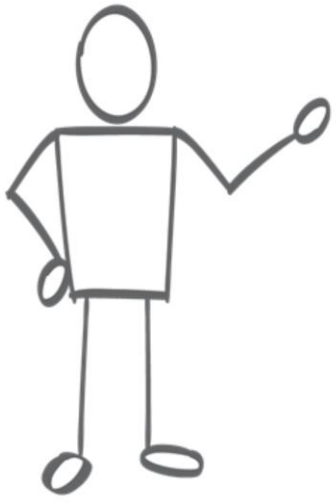
You learned that patterns such as broadcast and pipelines can help to group your units of work together in a modular process. And, by using many of these patterns, you can build rich microservices architectures running on AWS Lambda.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



DO NOT DISTRIBUTE



Module 4: VPC Integration

© 2016 Amazon Web Services, Inc. and its affiliates. All rights reserved.

amazon web services | Training and Certification

1

DO NOT DISTRIBUTE

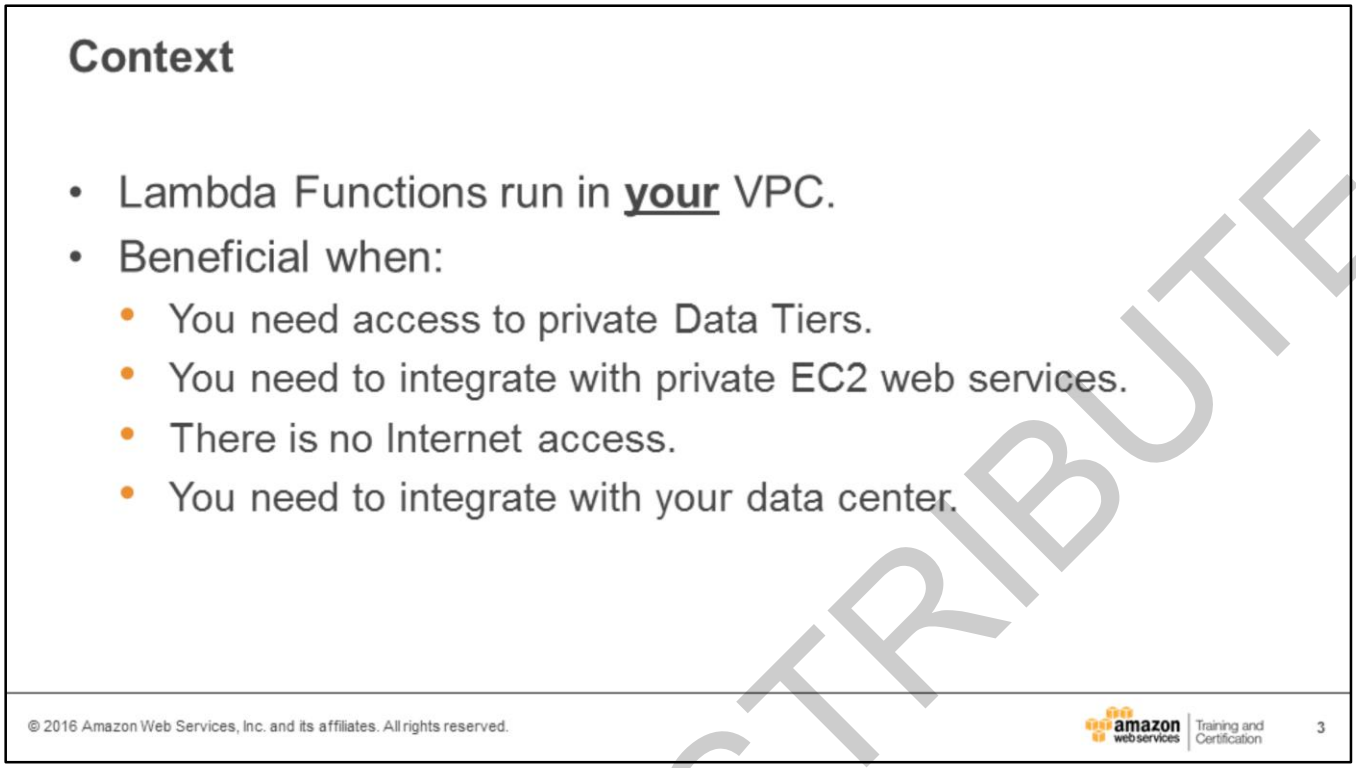
Agenda

- Context
- Lambda VPC Integration Use Cases
- How to Configure
- Account Artifacts
- Tradeoffs - VPC vs. Default
- Things to be Aware of
- Best Practices
- Lab

© 2016 Amazon Web Services, Inc. and its affiliates. All rights reserved.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



AWS Lambda runs your function code securely within a VPC by default. However, to enable your Lambda function to access resources inside your private VPC, you must provide additional VPC-specific configuration. By providing this additional configuration your function can access databases running in private subnets, access private web services running on Amazon EC2, and interface with on-premises resources via a VPN or Direct Connect connection. Also, by adding a VPC configuration to your Lambda function you can control its access to the Internet the same way you would for an EC2 instance without a public IP address.

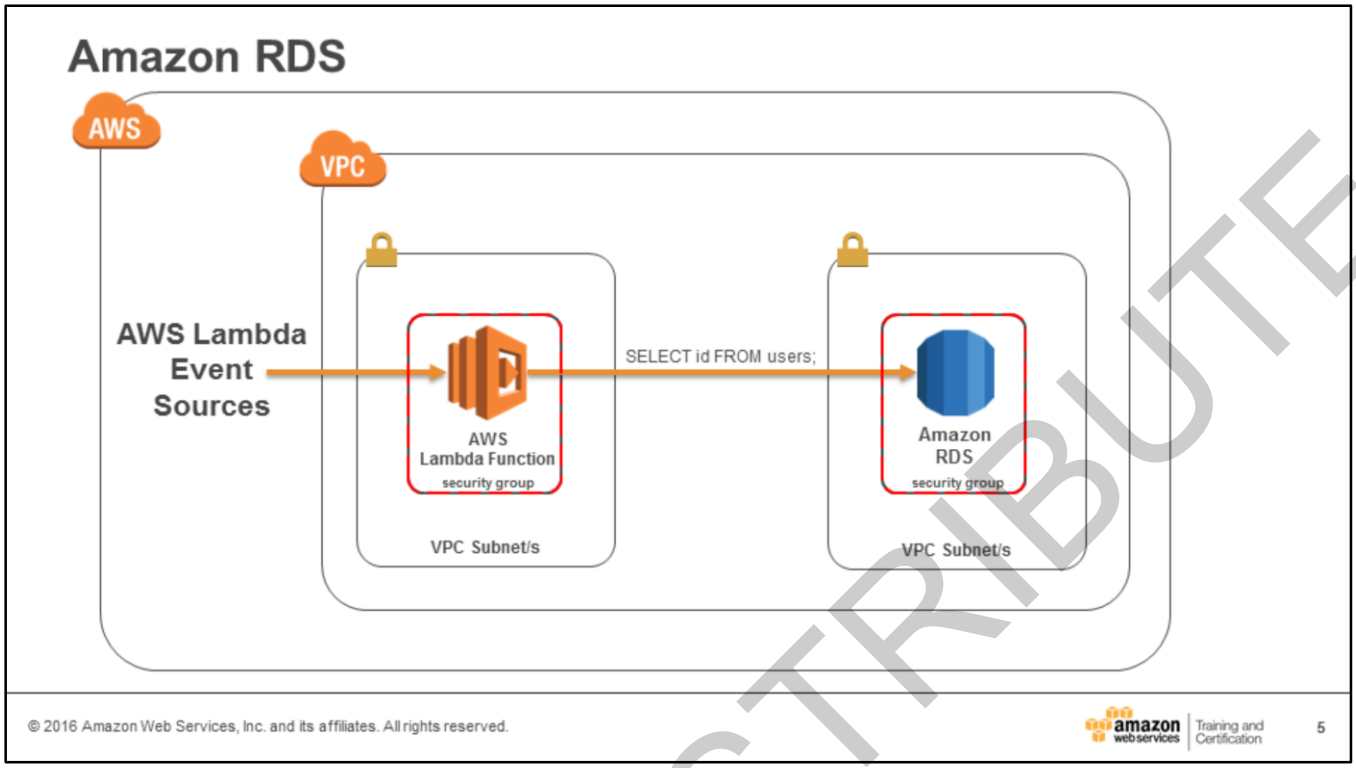
DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

Lambda VPC Integration Use Cases

© 2016 Amazon Web Services, Inc. and its affiliates. All rights reserved.

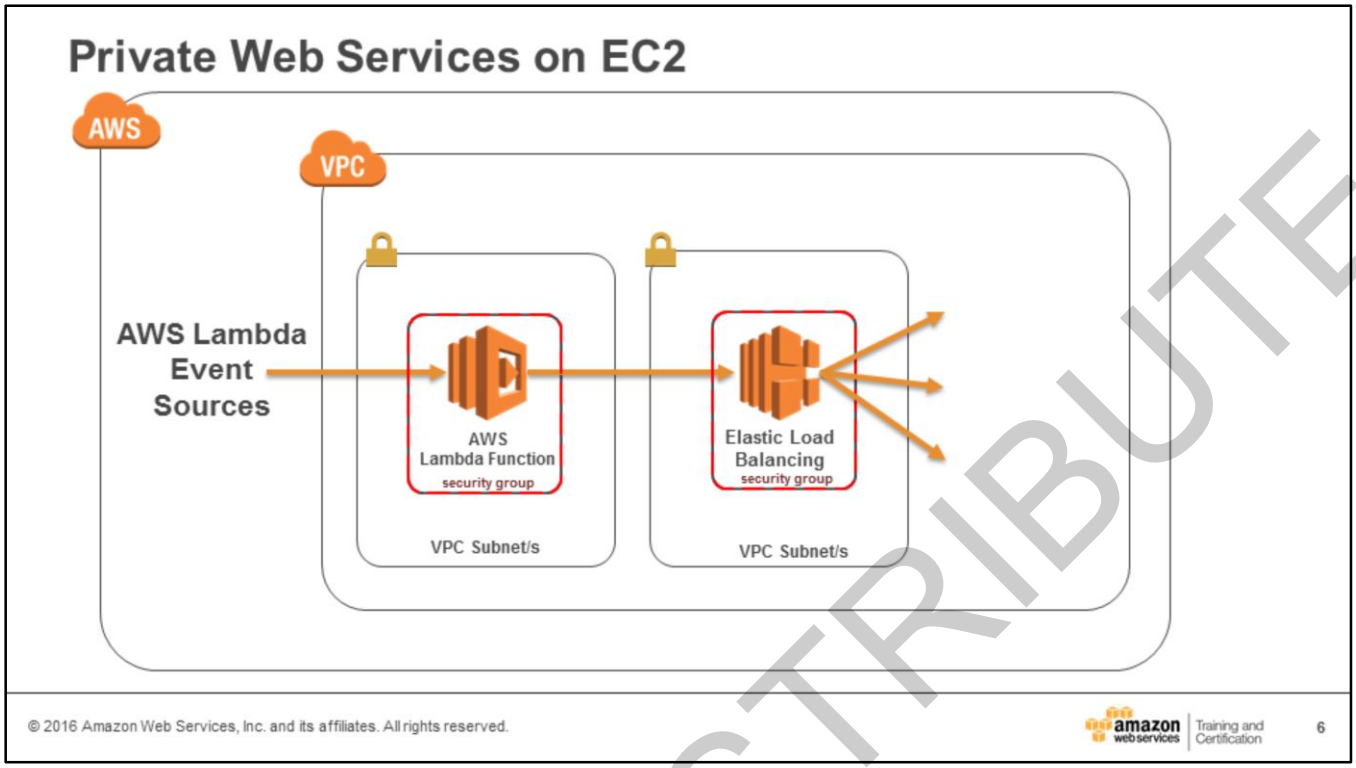
DO NOT DISTRIBUTE



Lambda functions can query RDS databases that run in private subnets. This is one of the most common use cases for configuring a Lambda function to access VPC resources. This configuration also works if you host your database on EC2.

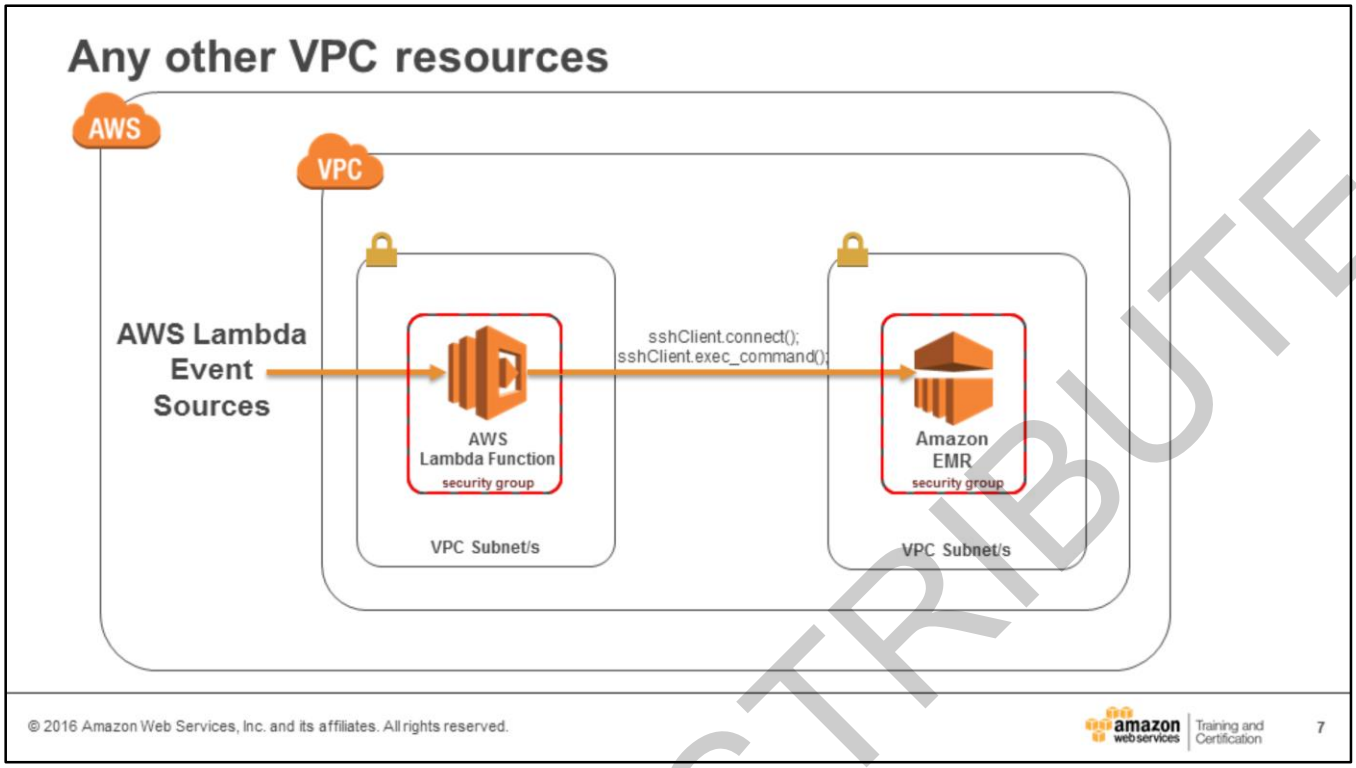
DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



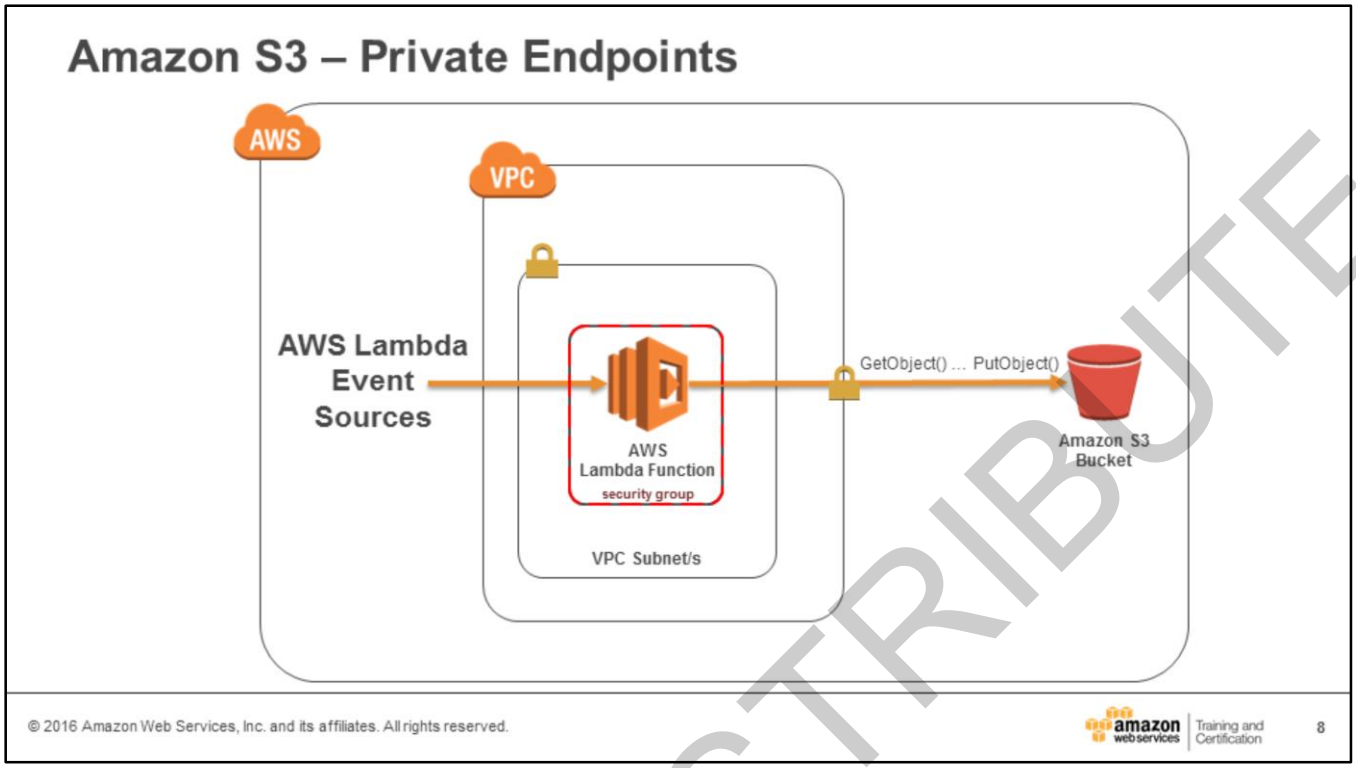
You can also access web services running on EC2 from your Lambda functions when they are configured for VPC access.

DO NOT DISTRIBUTE



Other AWS managed services that run in a VPC such as Amazon EMR, Amazon ElastiCache, and Amazon Redshift are also accessible.

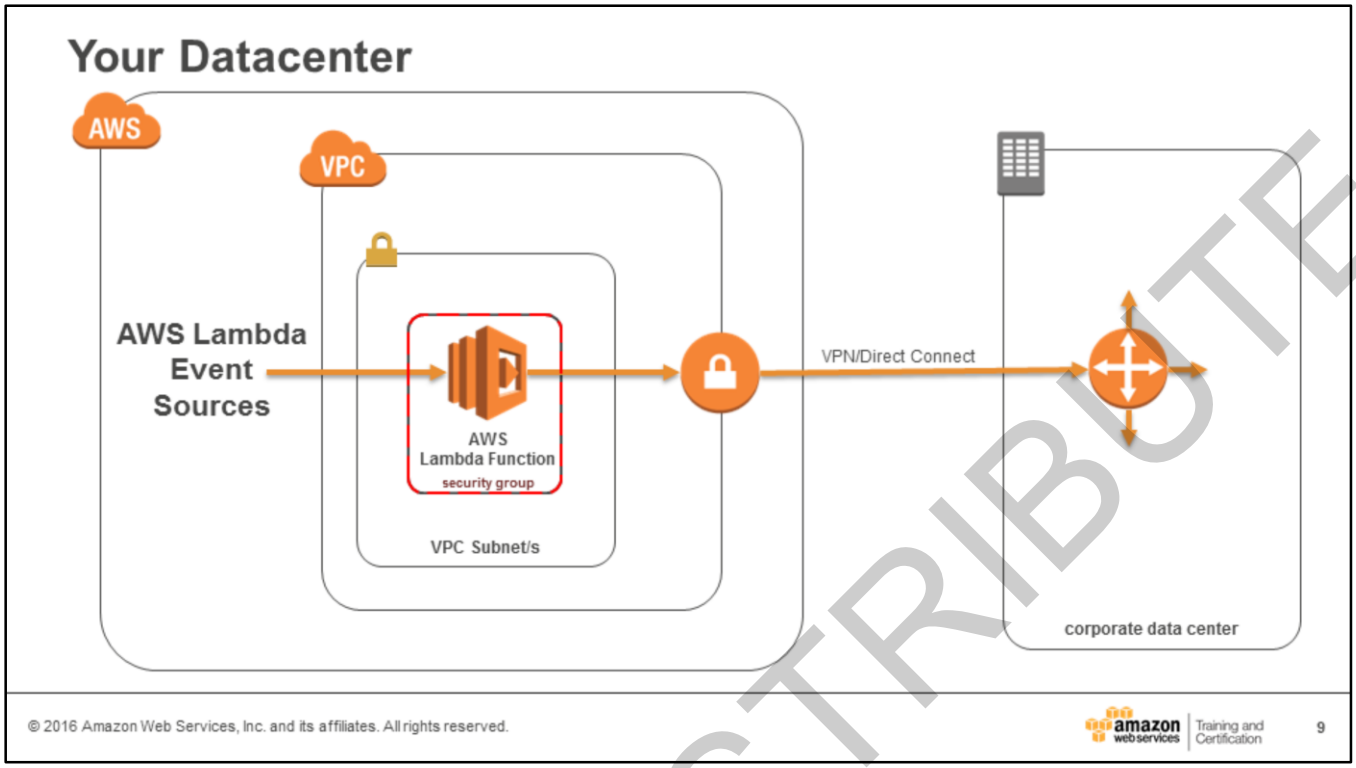
DO NOT DISTRIBUTE



Because your Lambda function uses an ENI that's associated with your VPC's subnet, it can access private endpoints the same way an EC2 instance can. This is useful if you have resources with policies that require the use of a VPC endpoint.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



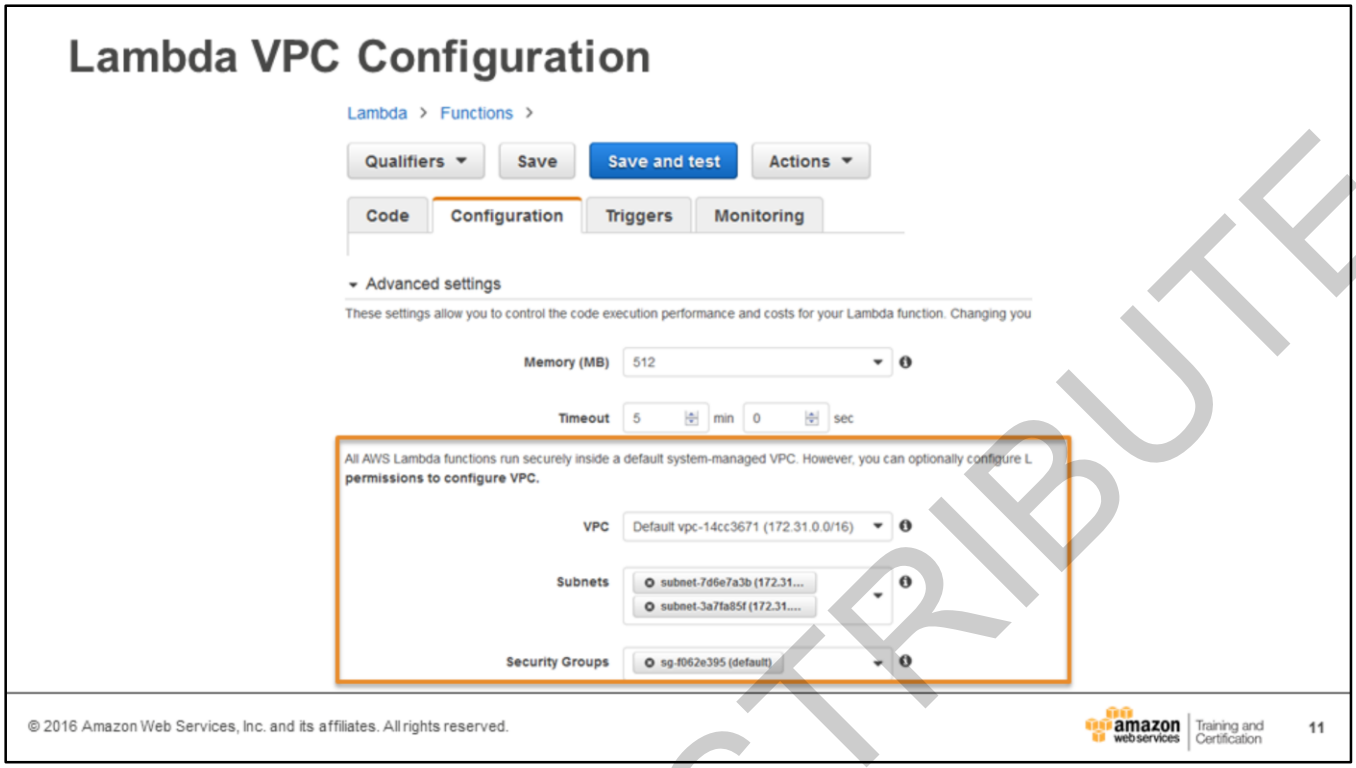
You can also use AWS Lambda to access resources running on-premises via a VPN or Direct Connect connection.

DO NOT DISTRIBUTE

Configuration and Operation

© 2016 Amazon Web Services, Inc. and its affiliates. All rights reserved.

DO NOT DISTRIBUTE

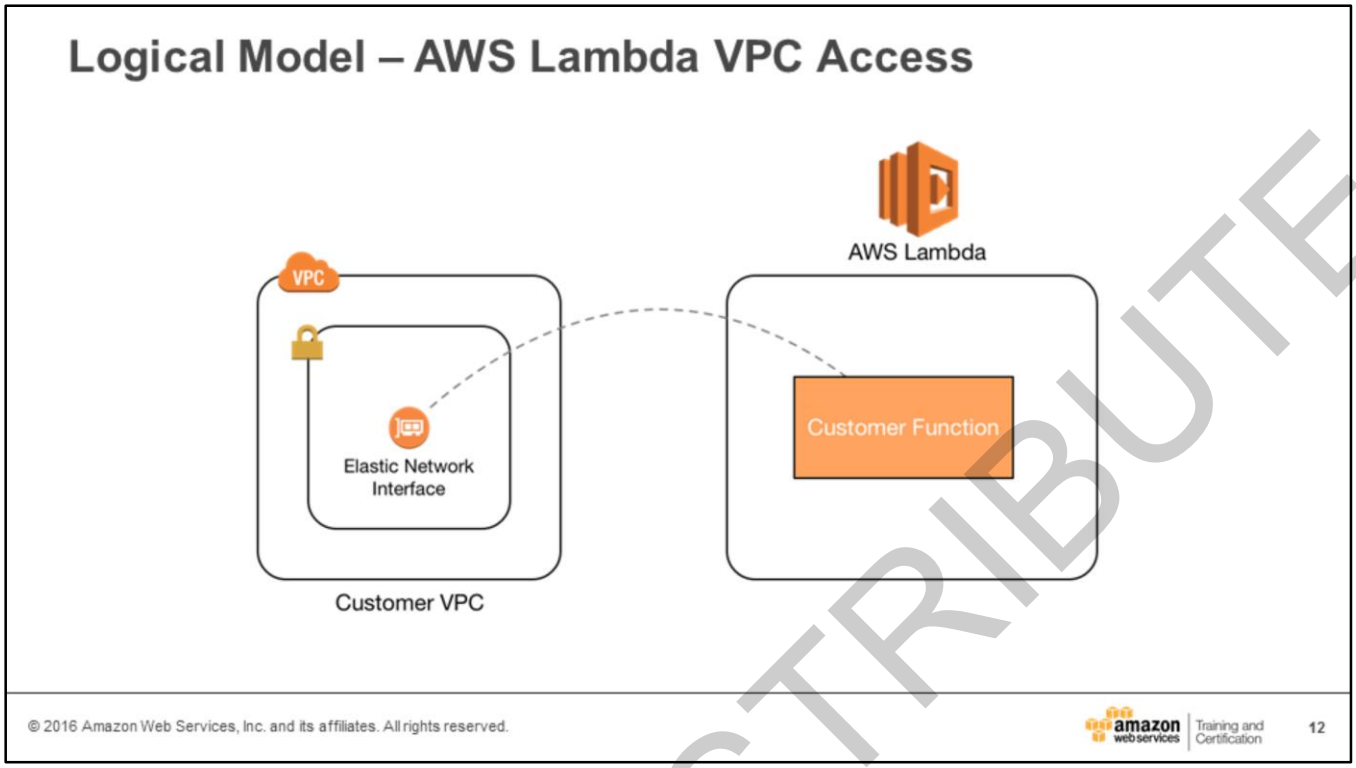


VPC configuration is done for each function. A given function can either run within the system-managed VPC or be configured to access a single VPC from your account. You cannot have the same function run in multiple VPCs at once.

When configuring a function to access VPC resources, you must define which VPC you want to access, one or more subnets and one or more security groups. All configured security groups will apply to each instance of your function, but each instance will only run in one of the configured subnets.

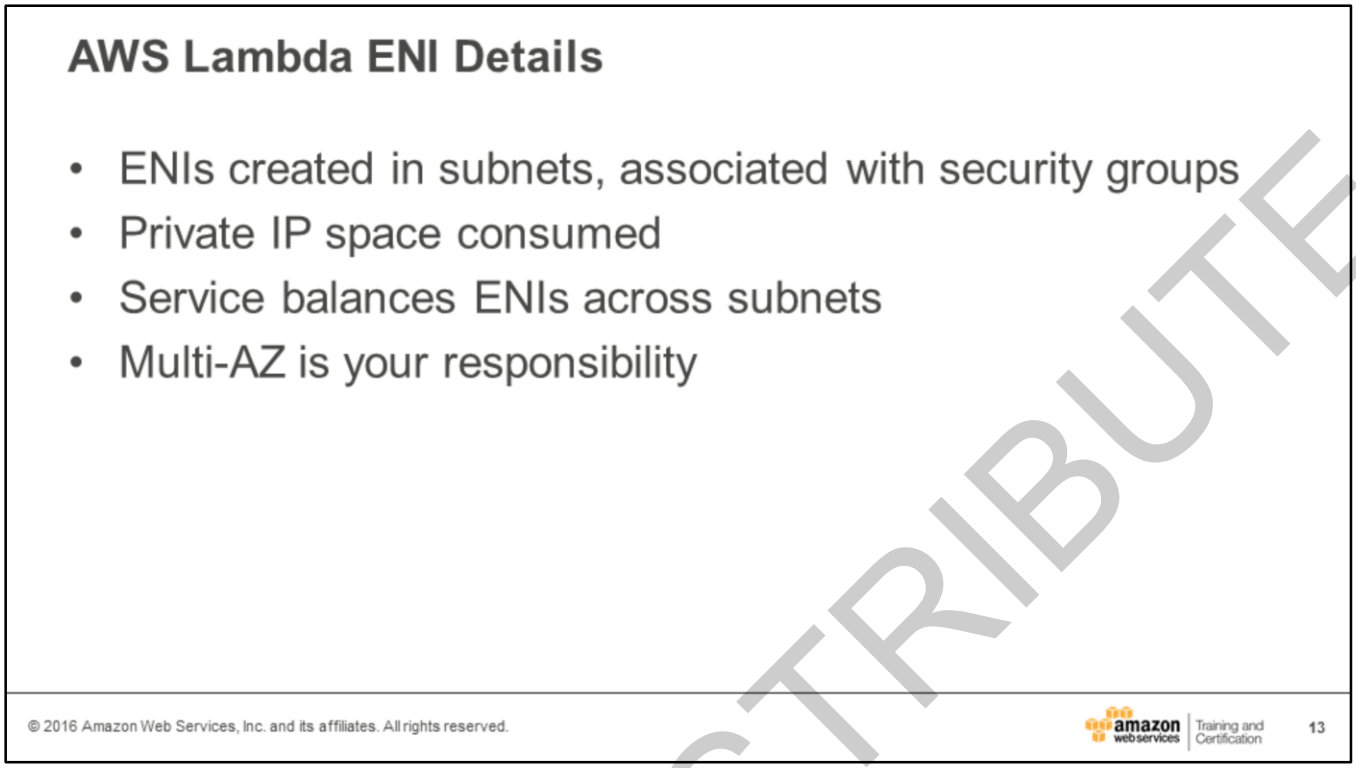
DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



AWS Lambda sets up elastic network interfaces (ENIs) that enable your function to connect securely to other resources within your private VPC. The rules that govern connectivity for these ENIs are the same as for interfaces attached to EC2 instances.

DO NOT DISTRIBUTE



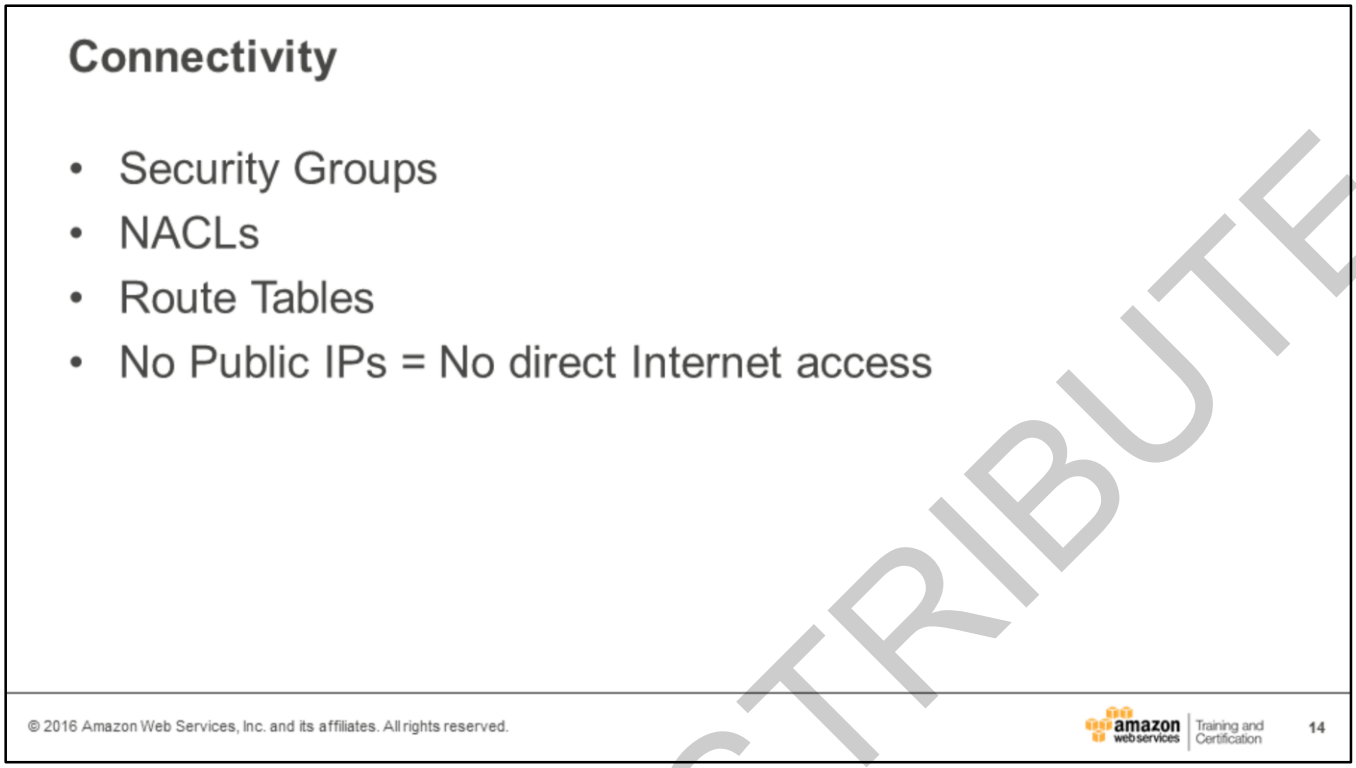
AWS Lambda will create ENIs in the subnets you configure. Each ENI will be associated with all of the security groups you define in your function's VPC configuration. Keep in mind that these ENIs will consume private address space within your subnets. If there are no IP addresses in any of the subnets you have configured, your Lambda functions will fail to execute.

When you configure multiple subnets for a Lambda function, the ENIs are balanced across those subnets. It's important that each of the subnets you configure for a given function allow all of the necessary traffic to flow to and from your function so that you don't end up with spurious failures.

It is a best practice to configure your Lambda functions to use subnets across multiple availability zones (AZs). By doing this, you ensure that your function can still run in the event that an AZ is unreachable.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



You can use security groups, NACLs, and route tables to control the network connectivity of your Lambda functions. These constructs work the same way as they do for EC2 instances or other VPC resources.

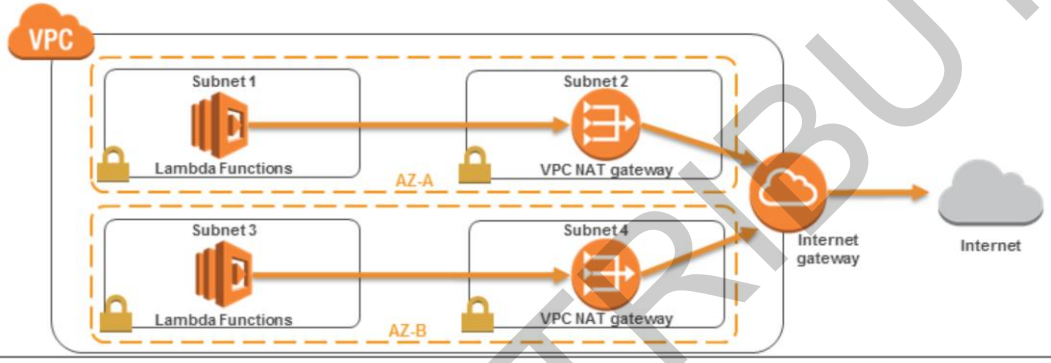
Also, keep in mind that because the ENIs used by AWS Lambda do not have public IP addresses, you cannot route Internet-bound traffic through an IGW. Instead, you should use a managed NAT Gateway or NAT instance for Internet access.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.

Internet Access via AWS NAT Gateway

- Use NAT Gateway to NAT Lambda-sourced traffic to the Internet.
- Create NAT Gateway for each availability zone in use.



© 2016 Amazon Web Services, Inc. and its affiliates. All rights reserved.
amazon web services | Training and Certification 15

If you want your Lambda functions to be able to access the Internet, the default route for all of the subnets your function runs in must point to a NAT gateway or NAT instance.

DO NOT DISTRIBUTE

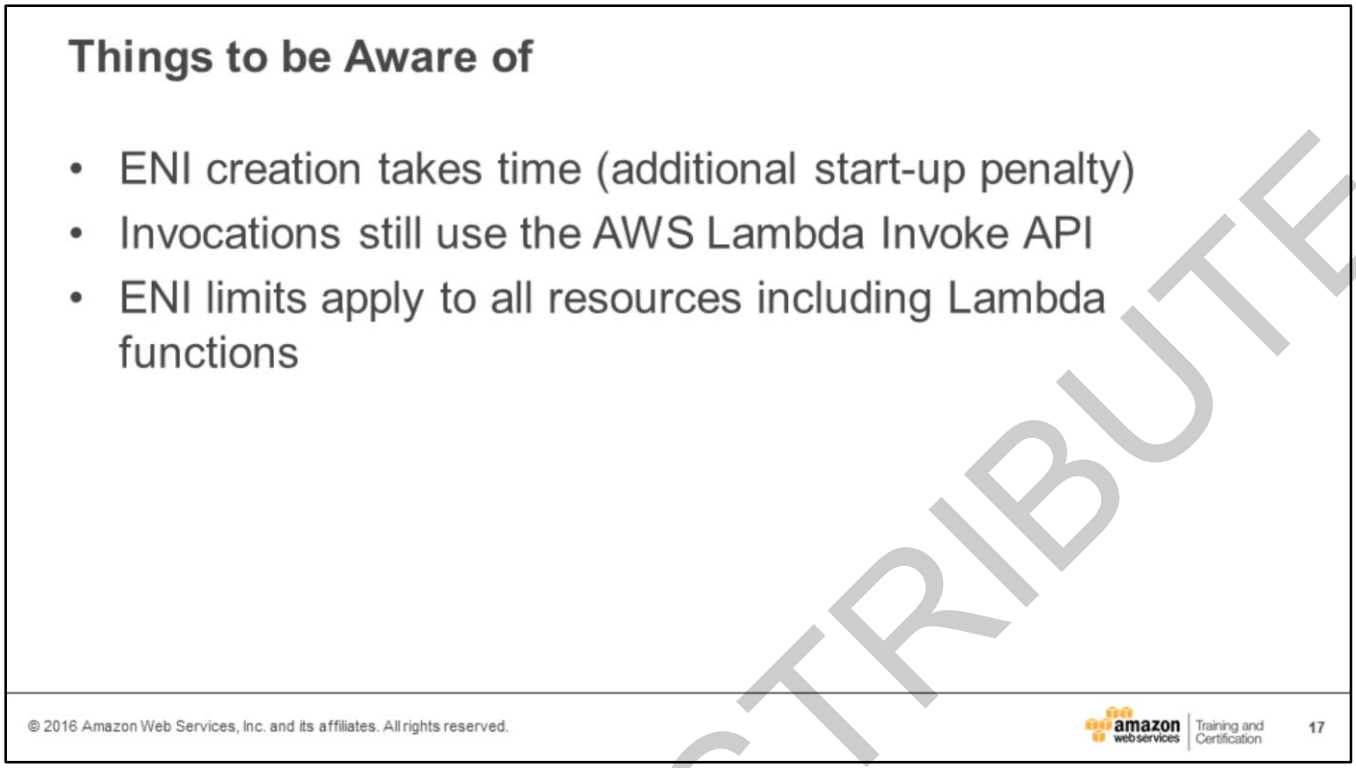
Default vs. VPC

Default	VPC
Runs in Service VPC	Runs in Customer VPC
Communication via VPC Private IPs Not Possible	Communication with VPC resources via private IPs IS Possible
Internet Connectivity Always Allowed	Internet Connectivity via NAT
Multi-AZ Managed by Service	Multi-AZ Dependent on Subnet Selection
No IP consumption concerns	Subnet private IPs consumed

© 2016 Amazon Web Services, Inc. and its affiliates. All rights reserved.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



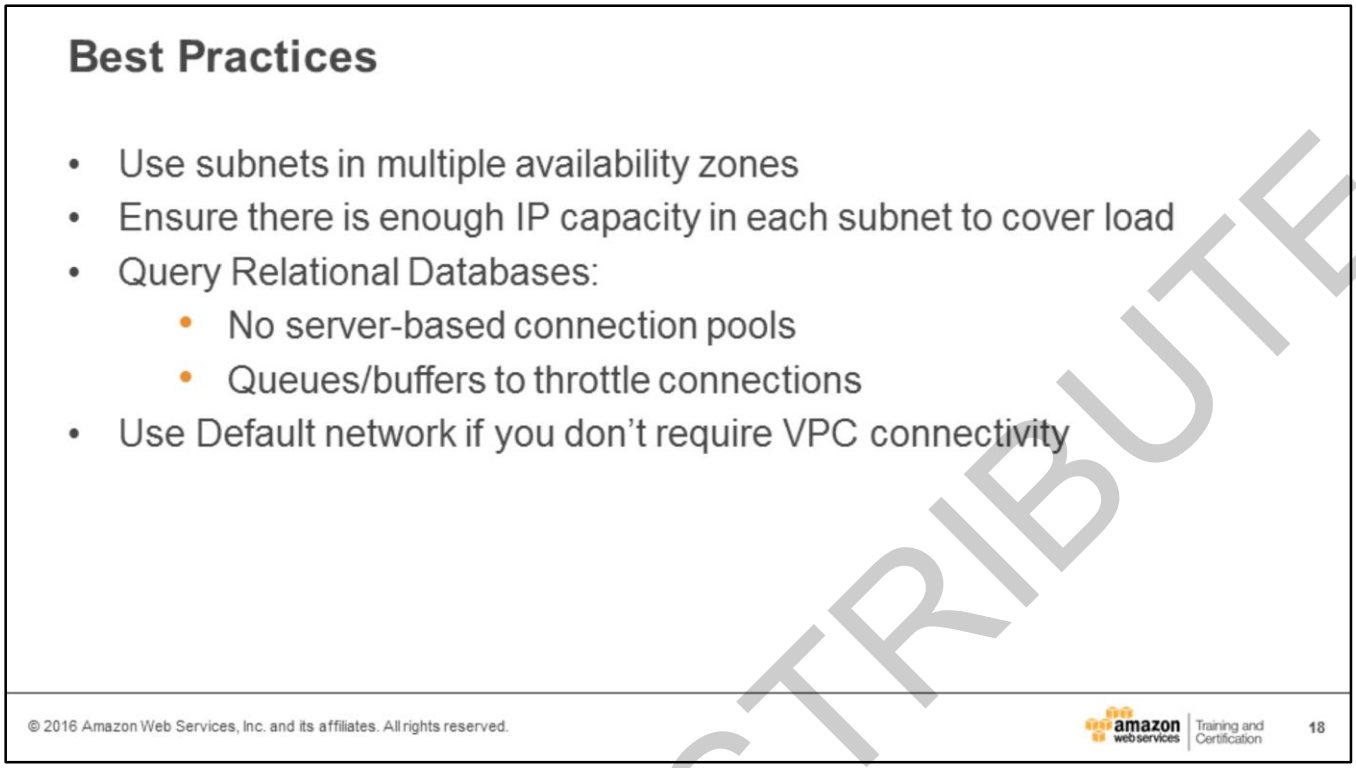
The first time you invoke a Lambda function that is configured to access a VPC, there will be an additional cold start penalty while AWS Lambda creates and configures an ENI in your VPC. This additional penalty occurs each time a new ENI is created, but ENIs are reused across function invocations.

Any process with access to the AWS Lambda API endpoints can still invoke your function regardless of its VPC configuration. You can limit who can invoke your function using IAM and resource policies.

There are limits on the number of ENIs you can provision per region. This limit applies to all VPC resources, including Lambda functions. This means if you have already provisioned your limit of ENIs in a region by launching EC2 instances, your Lambda functions that are configured for VPC access will not be able to run.

DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



You should configure your functions to use subnets in multiple availability zones. AWS Lambda will automatically shift invocations away from an availability zone that is not reachable.

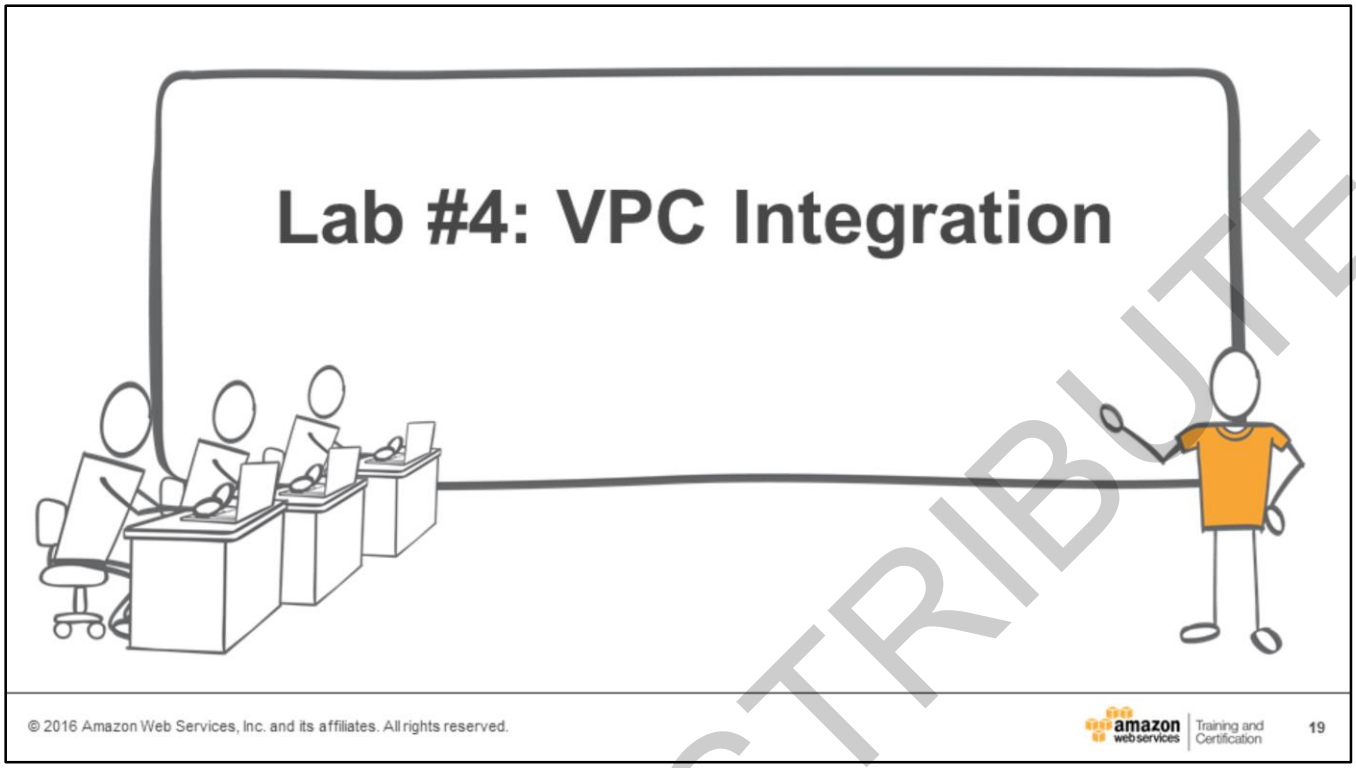
Make sure you size your subnets to handle the number of concurrent invocations you want to support. You will need an available private IP for every concurrent function invocation. If you have other elastic resources such as load balancers or Auto Scaling groups that run in the same subnet, be sure you consider the maximal address space your system will require.

Lambda functions are stateless, which means you cannot manage database connection pools the same way you would in a traditional server application.

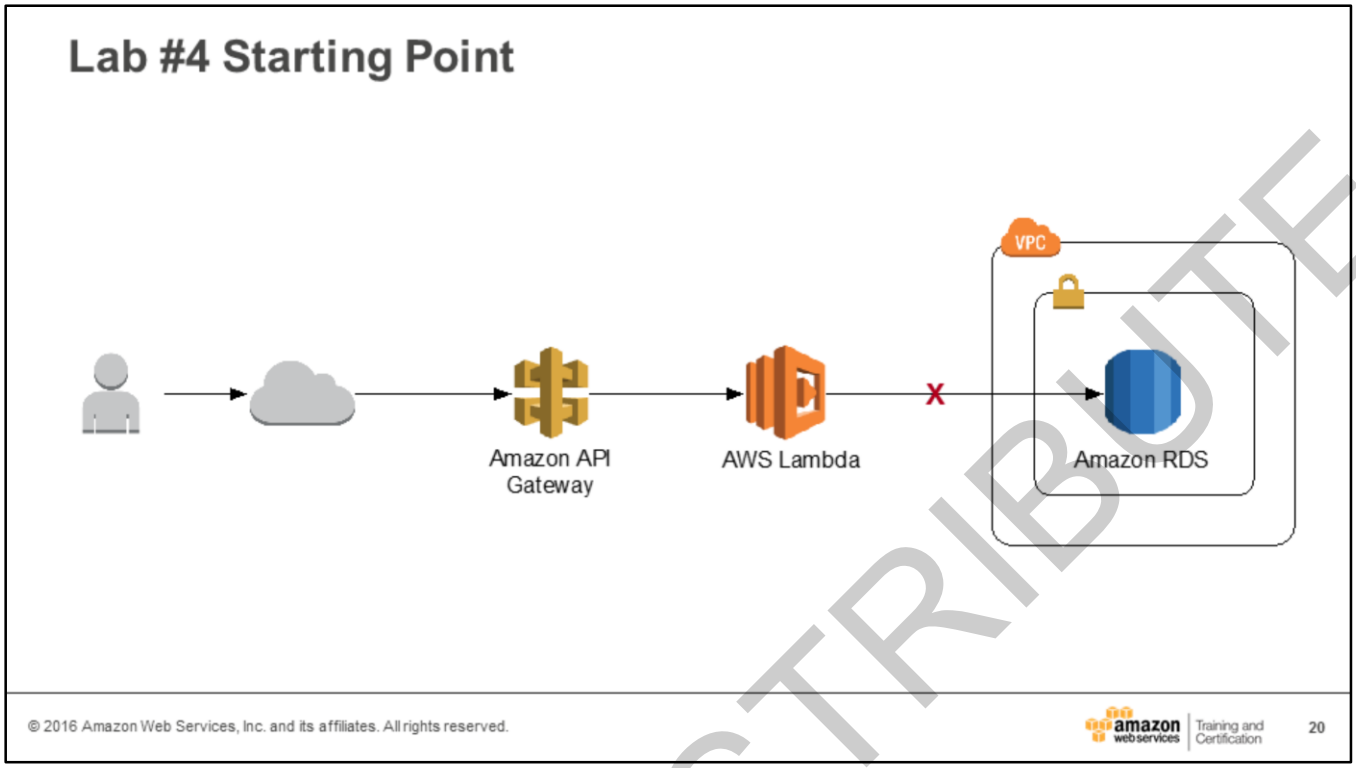
Don't configure your functions to use a VPC unless you have to access private resources. Functions that run in the system-managed VPC start faster and have fewer constraints to consider.

DO NOT DISTRIBUTE

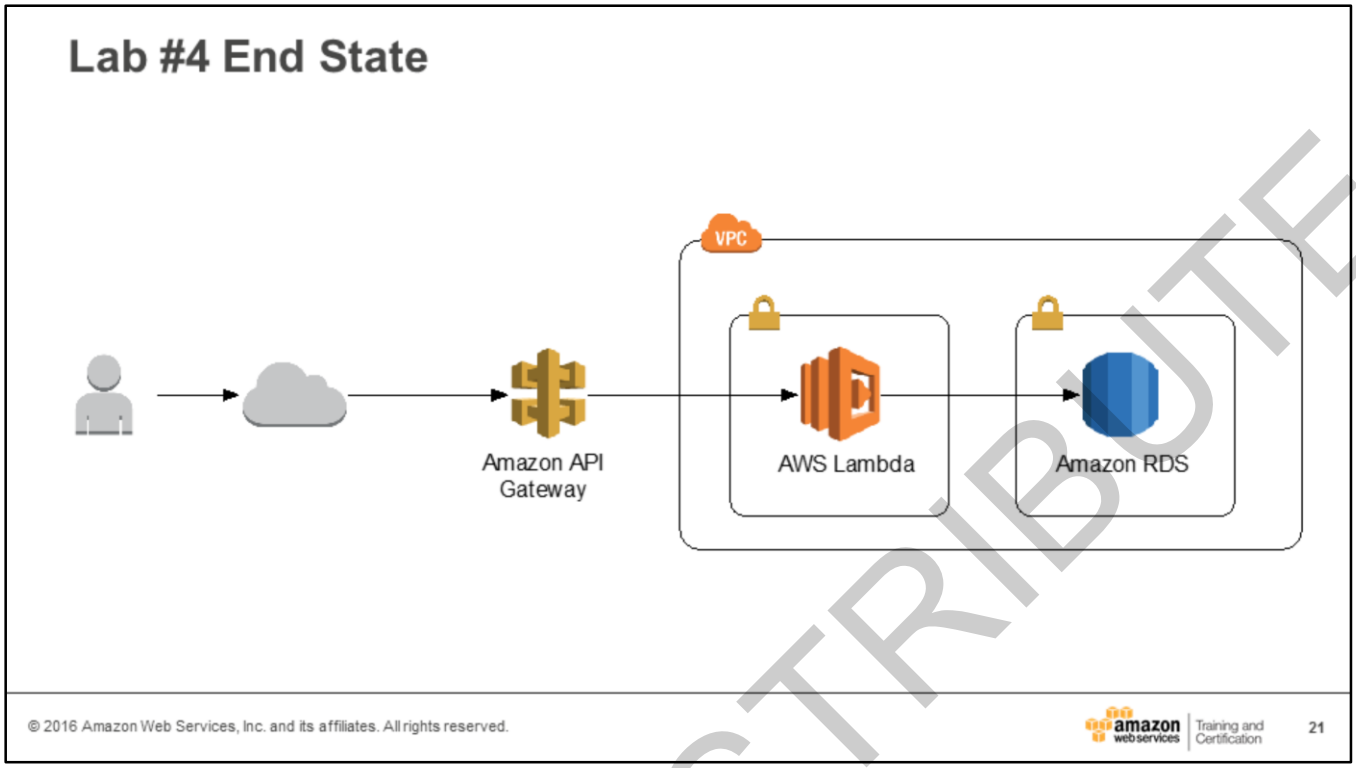
© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.



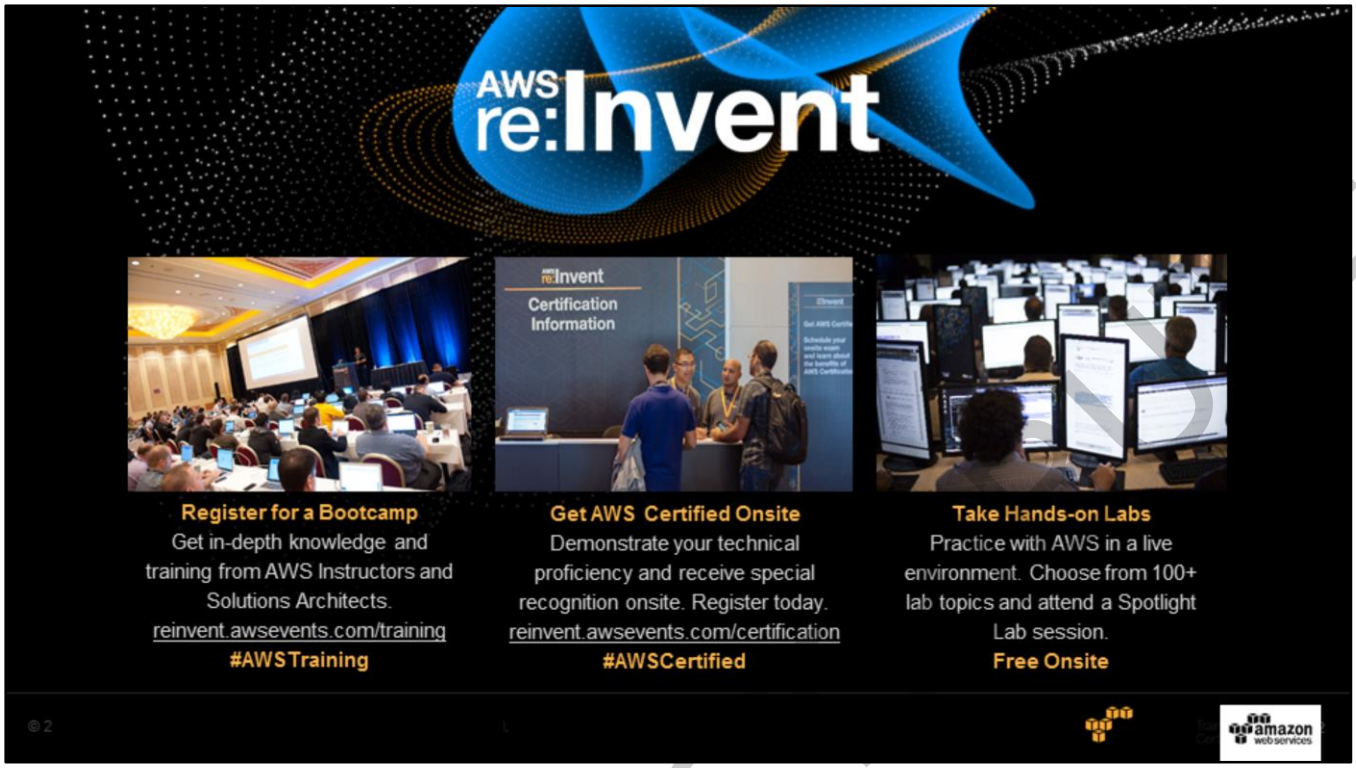
DO NOT DISTRIBUTE



DO NOT DISTRIBUTE



DO NOT DISTRIBUTE



DO NOT DISTRIBUTE

© 2016 Amazon Web Services, Inc. or its affiliates All rights reserved.