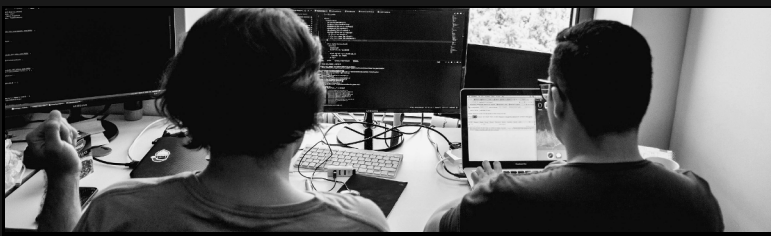
Serverless Application on AWS

Course Navigation



Welcome to Fullstack Serverless Application on AWS

Components:

- Front end layer
- Application or Business Logic Layer
- Data persistence Layer

In this course we'll be spending a lot of time on the following:

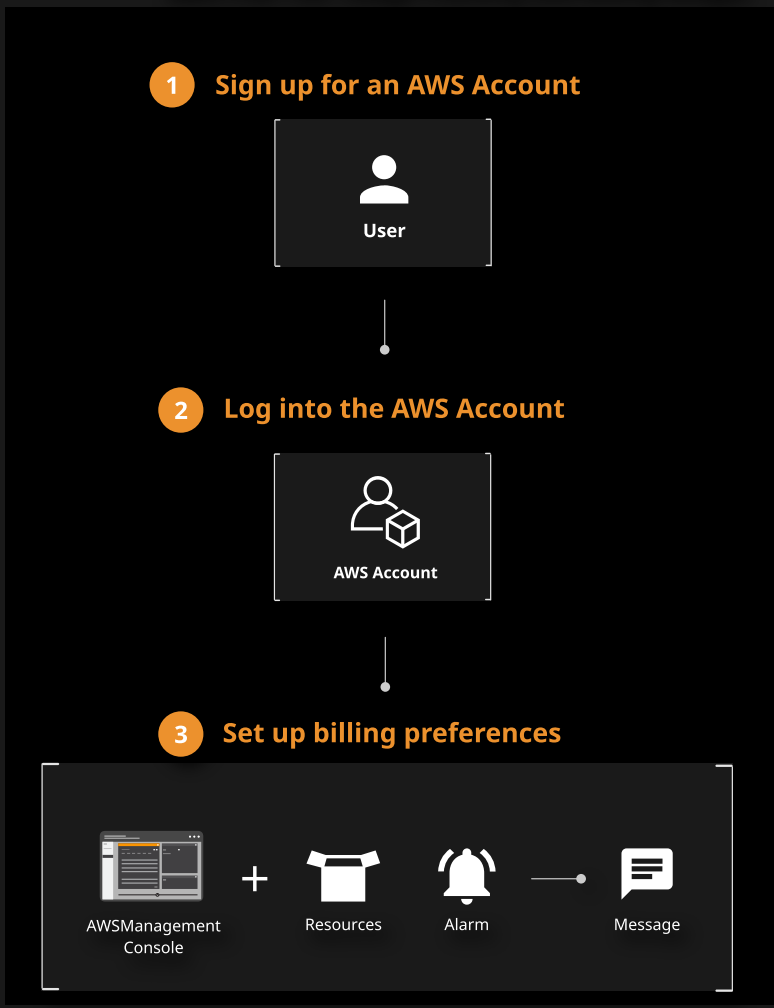
- AWS CLI API calls
- S3 & S3 Policies
- CloudFront
- Lambda Functions and event triggers
- Aurora RDS Serverless DB
- Linux command line

Moosa Khalid
AWS Training Architect

Course Introduction

AWS Free Tier: Usage tracking and Billing Widget

Course Navigation



Back to Main

Linux Academy

Course Introduction

What You Should Know Beforehand: Prerequisites

Before taking this course, here are a few things you should be familiar with:

- AWS services such as **IAM and S3 policies**
- **Python 3 and the Boto AWS SDK** — take our **Automating AWS with Lambda, Python, and Boto3** course to learn more
- **Linux CLI** — we'll execute a lot of Linux CLI commands and AWS CLI commands on Linux OS in the videos and hands-on labs

Python
(Boto)

Back to Main

Serverless Defined

Why Use Serverless?

- No servers to **manage**
- Inherent **scaling** and **HA**
- No idle capacity — **lower costs**
- Quick **deployment** and updates
- **Pay-as-you-go** model
- Event/microservices **architectures**

Serverless Defined

Serverless Architectures

Serverless App Components

Front End | Application | Data Persistence

A **serverless application** doesn't necessarily need all the components defined above to work. It all depends on your **use case and app business logic.**

Serverless App Architectures

Event-Driven Architecture + Microservices-Based Architecture

Serverless services are excellent for microservices- and event-based architectures, where you can break down the proverbial monolithic monster and focus on **independent, self-contained components** that are excellent at doing one task very efficiently.

In addition to the HA that serverless services inherently offer, having a modular, event-based microservices architecture application on top of it also provides **added stability and disaster recovery.**

Back

Next

Back to Main

Serverless Defined

Serverless Architectures

Serverless App Components

Front End | **Application** | **Data Persistence**

The **front-end layer** of a serverless ecosystem is basically the presentation layer or the **GUI** (graphical user interface) for the application. A serverless application can just be a **front-end layer** (e.g., an S3 static standalone website, a mobile app's GUI, etc.).

Serverless App Architectures

Event-Driven Architecture + **Microservices-Based Architecture**

Serverless services are excellent for microservices- and event-based architectures, where you can break down the proverbial monolithic monster and focus on **independent, self-contained components** that are excellent at doing one task very efficiently.

In addition to the HA that serverless services inherently offer, having a modular, event-based microservices architecture application on top of it also provides **added stability and disaster recovery.**

Serverless Defined

Serverless Architectures

Serverless App Components

Front End | **Application** | **Data Persistence**

The **application layer** (also known as the business logic layer or middleware) is where the logical decisions of your app are made. In short, **this is where you use "compute" resources.** For example, if you have an online bookshop where users create accounts and buy books, this part of the application will compute what action the user is performing and **direct the inputs and outputs to be stored in a database,** as well as issue receipts and email notifications to users.

Serverless App Architectures

Event-Driven Architecture + **Microservices-Based Architecture**

Serverless services are excellent for microservices- and event-based architectures, where you can break down the proverbial monolithic monster and focus on **independent, self-contained components** that are excellent at doing one task very efficiently.

In addition to the HA that serverless services inherently offer, having a modular, event-based microservices architecture application on top of it also provides **added stability and disaster recovery.**

Serverless Defined

Serverless Architectures

Serverless App Components

Front End | **Application** | **Data Persistence**

The application or business logic layer of a serverless app is usually stateless, which means the state of incoming requests or inputs to the function/code is **not saved at the application layer, hence the need for a data persistence layer** (usually some kind of database or data store). For instance, the application code will not remember what product Customer A bought on an online shopping website and hence stores that data persistently to query it at a later time as required. Examples of serverless data layer services include **AWS Aurora Serverless and AWS DynamoDB.**

Event-Driven Architecture + **Microservices-Based Architecture**

Serverless services are excellent for microservices- and event-based architectures, where you can break down the proverbial monolithic monster and focus on **independent, self-contained components** that are excellent at doing one task very efficiently.

In addition to the HA that serverless services inherently offer, having a modular, event-based microservices architecture application on top of it also provides **added stability and disaster recovery.**

Serverless Defined

Serverless Architectures

Serverless App Components

Front End | **Application** | **Data Persistence**

A **serverless application** doesn't necessarily need all the components defined above to work. It all depends on your **use case and app business logic.**

Serverless App Architectures

Event-Driven Architecture + **Microservices-Based Architecture**

- **Actions/code/functions** are triggered in reaction to **events.**
- An event could be a **change in state.**
- Normally, events are triggered via **front-end layer** and consumed by **application and data persistence layers.**
- Allow for **flexible and microservice-based designs.**

Serverless App Components

Front End | Application | Data Persistence

A **serverless application** doesn't necessarily need all the components defined above to work. It all depends on your **use case and app business logic.**

Serverless App Architectures

Event-Driven Architecture + Microservices-Based Architecture

- Components within application are **loosely coupled.**
- **Modular design** — components are independently deployable.
- Components communicate via **lightweight protocols** such as HTTP.
- Works well with **continuous delivery methodologies and event-based architectures.**

... And the Limitations

- Testing and debugging is **challenging**
- Security **concerns**
- Not built for **long-running** processes
- Cold start times can affect resources/functions that are not used frequently, **causing latency**
- Best for applications with short-running processes, as you're charged for the **time your process/code runs**

Serverless on the Rise

With an **increasing** amount of users, the serverless community is growing at a fast pace and finding ways to **mitigate or eradicate** all the limits mentioned above as much as possible. For example, with AWS Lambda serverless functions (FaaS), you can now use AWS X-Ray service to debug in depth as well as use **CloudWatch** Events to keep your functions ready and "warmed" up.

Front-End Serverless Layer: S3 and CloudFront

Traditional Web Server Front End

- With traditional server "full" architectures, **scaling needs to be monitored** — extra stress
- Servers to **maintain!**
- Overkill for **single-page**, static website

Serverless Front End

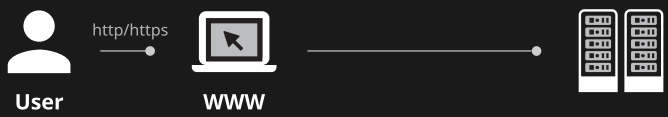
- S3 scales **automatically** as traffic bursts
- Highly available thanks to being **AWS S3** backed
- No servers to **maintain** at all
- Developers can easily leverage **SDK/API** for end-to-end setup process

AWS - CloudFront S3 - Website

Next

Front-End Serverless Layer: S3 and CloudFront

Traditional Web Server Front End



- With traditional server "full" architectures, **scaling needs to be monitored** — extra stress
- Servers to **maintain!**
- Overkill for **single-page**, static website

- Scalable
- Integration with AWS CloudFront
- Easy integration with AWS Route 53 DNS
- For individual web pages or client-side scripts
- Programmatic access via SDK/APIs

AWS - CloudFront **S3 - Website**

Front-End Serverless Layer: S3 and CloudFront

- Content delivery network of edge locations
- Caches static files for lower latency
- DDos protection
- Offloads website
- Integrates well with AWS S3
- Offers HTTPS termination via ACM
- Programmatic access via SDK/APIs

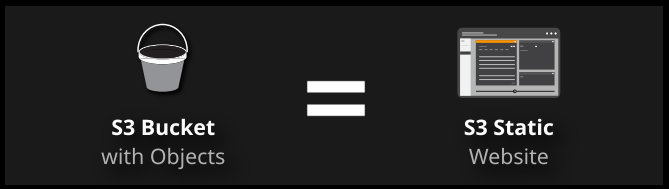
Serverless Front End

- S3 scales **automatically** as traffic bursts
- Highly available thanks to being **AWS S3** backed
- No servers to **maintain** at all
- Developers can easily leverage **SDK/API** for end-to-end setup process

AWS - CloudFront **S3 - Website**

Next

Setting Up AWS S3 as a Standalone Website via Console

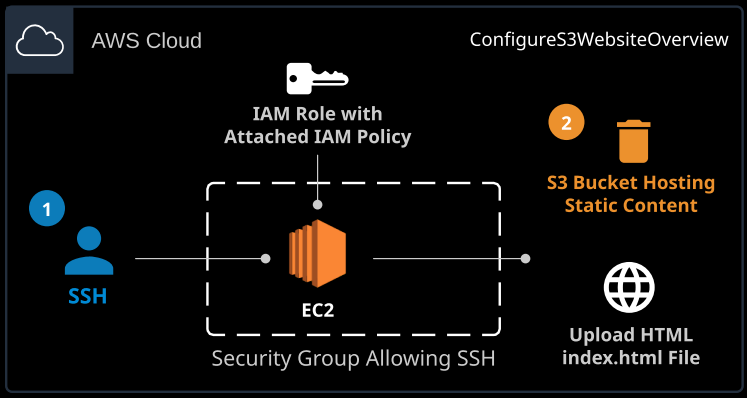


- **Static website:** A website with fixed content. Shows the same content to all viewers.
- **S3 static website bucket** is simply a bucket with a property enabled to allow it to host content.
- **Remember:** S3 website URL is not the same as an S3 bucket URL!
- Content hosted by a website bucket has to be **publicly readable** — i.e., s3:GetObject allowed on it.
- You cannot have **server-side code execution** since there's no back-end servers to run code on. Only **client-side scripting and individual web pages** can be hosted.
- **S3 policies** and **ACLs** still apply here, so beware of allowing read access on your publicly visible content within the bucket.

Configuring S3 via CLI

Front-End Serverless Layer: S3 and CloudFront

Setting Up AWS S3 as a Standalone Website via Console



Summary Breakdown of Steps

1. **Have AWS CLI API installed. In our case, we log in via SSH to an EC2 instance already set up with it.**

   Intermediate but important step:
   Createandattachan**IAM role**(allowingS3read/writeaccess)to your**EC2 instance**soitcanassumethatroleandmakesuccessfulAPI calls.

2. **Issue the correct AWS CLI commands to create and modify S3 bucket and upload your website files in the most basic instance, namely: index.html.**

   Supports two URL formats:
   1. <bucket-name>.s3-website.<AWS-region>.amazonaws.com
   2. <bucket-name>.s3-website-<AWS-region>.amazonaws.com
      - S3 static website requires public read access on bucket
      - Does not support HTTPS

Back

Back to Main

CloudFront via CLI

Amazon CloudFront

- Worldwide group of **edge** (caching) locations.
- On initial creation, a **CloudFront distribution** takes about 20 minutes to sync across the globe.
- Works via **"web distributions"** in our scenario, which are hooked to the origin (server, bucket, EC2 instance, etc.) from where your content is to be served and cached from.
- User can change value of **"viewer protocol policy"** in web distribution to redirect all HTTP traffic to HTTPS.
- CloudFront provides default **SSL certificates** you can use for **HTTPS**; however, users can upload their own custom certs and route to CloudFront from a **DNS** such as Route 53.

Low-Latency Access to Content via Caching

DDoS Prevention

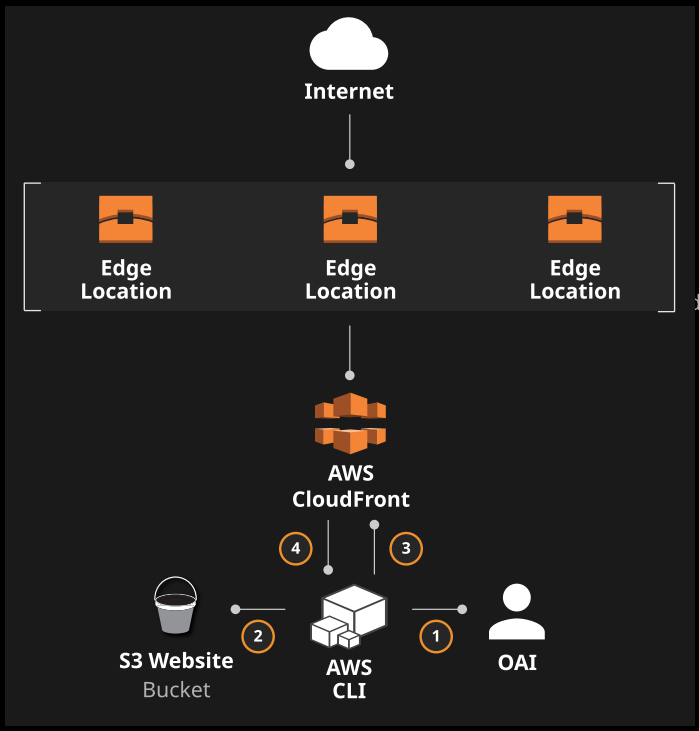
HTTPS Endpoint

Back

Front-End Serverless Layer: S3 and CloudFront

AWS CloudFront as an HTTPS Endpoint Provider

CloudFront via CLI



Back

CloudFront via CLI

Amazon CloudFront

- Worldwide group of **edge** (caching) locations.
- On initial creation, a **CloudFront distribution** takes about 20 minutes to sync across the globe.
- Works via **"web distributions"** in our scenario, which are hooked to the origin (server, bucket, EC2 instance, etc.) from where your content is to be served and cached from.
- User can change value of **"viewer protocol policy"** in web distribution to redirect all HTTP traffic to HTTPS.
- CloudFront provides default **SSL certificates** you can use for **HTTPS**; however, users can upload their own custom certs and route to CloudFront from a **DNS** such as Route 53.

Low-Latency Access to Content via Caching

DDoS Prevention

HTTPS Endpoint

1

Create an **origin access identity** in CloudFront. It is a special CloudFront user that can access an **S3 bucket on behalf of users** accessing objects in a website bucket via the CloudFront service.

Back

Front-End Serverless Layer: S3 and CloudFront

AWS CloudFront as an HTTPS Endpoint Provider

CloudFront via CLI

Amazon CloudFront

- Worldwide group of **edge** (caching) locations.
- On initial creation, a **CloudFront distribution** takes about 20 minutes to sync across the globe.
- Works via **"web distributions"** in our scenario, which are hooked to the origin (server, bucket, EC2 instance, etc.) from where your content is to be served and cached from.
- User can change value of **"viewer protocol policy"** in web distribution to redirect all HTTP traffic to HTTPS.
- CloudFront provides default **SSL certificates** you can use for **HTTPS**; however, users can upload their own custom certs and route to CloudFront from a **DNS** such as Route 53.

Low-Latency Access to Content via Caching | DDoS Prevention | HTTPS Endpoint

2

Attach an **S3 bucket policy** to the already created S3 bucket website to allow CloudFront OAI user **s3:GetObject** access on all objects within the bucket.

Back

CloudFront via CLI

Amazon CloudFront

- Worldwide group of **edge** (caching) locations.
- On initial creation, a **CloudFront distribution** takes about 20 minutes to sync across the globe.
- Works via **"web distributions"** in our scenario, which are hooked to the origin (server, bucket, EC2 instance, etc.) from where your content is to be served and cached from.
- User can change value of **"viewer protocol policy"** in web distribution to redirect all HTTP traffic to HTTPS.
- CloudFront provides default **SSL certificates** you can use for **HTTPS**; however, users can upload their own custom certs and route to CloudFront from a **DNS** such as Route 53.

Low-Latency Access to Content via Caching | DDoS Prevention | HTTPS Endpoint

3

Create a **CloudFront** distribution using **CLI**. It caches and serves your content via edge locations.

Back

AWS CloudFront as an HTTPS Endpoint Provider

CloudFront via CLI

Amazon CloudFront

- Worldwide group of **edge** (caching) locations.
- On initial creation, a **CloudFront distribution** takes about 20 minutes to sync across the globe.
- Works via **"web distributions"** in our scenario, which are hooked to the origin (server, bucket, EC2 instance, etc.) from where your content is to be served and cached from.
- User can change value of **"viewer protocol policy"** in web distribution to redirect all HTTP traffic to HTTPS.
- CloudFront provides default **SSL certificates** you can use for **HTTPS**; however, users can upload their own custom certs and route to CloudFront from a **DNS** such as Route 53.

Low-Latency Access to Content via Caching

DDoS Prevention

HTTPS Endpoint

4

Get the configuration of the **CloudFront distribution you just created,** modify it (PriceClass, OAI, redirect all to HTTPS), and update the original CloudFront distribution configuration with your **modified one — all via CLI.**

Back

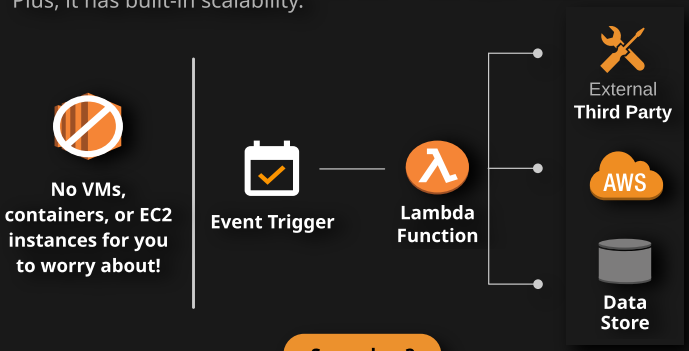
Application Layer: AWS Lambda Functions

Overview

Under the Hood: Lambda

Function as a Service (FaaS) is a type of serverless, event-driven computing.

No containers or **VMs** are present, and users just focus on the code. Plus, it has built-in scalability.



Serverless?

- Charged **based on requests + duration of execution**
- Supports **blue/green** deployments
- Allows useful triggers from **S3 and SNS** to execute
- A single dial configuration of memory, with which **CPU scales proportionally**
- Gels with **CloudFront** to offer Lambda@Edge
- Automatic scaling, **built-in fault tolerance**
- Bring your own code or language using **Lambda Layers**

Application Layer: AWS Lambda Functions

Overview

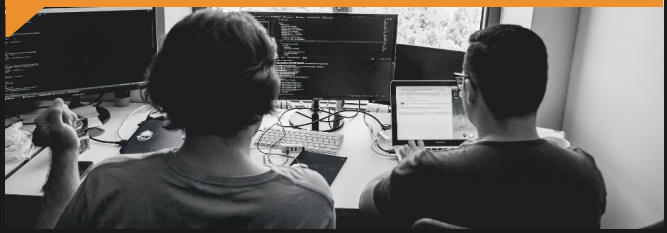
Under the Hood: Lambda

Function as a Service (FaaS) is a type of serverless, event-driven computing.

No containers or **VMs** are present, and users just focus on the code. Plus, it has built-in scalability.

Back

There really is no such thing as "serverless." Rather, it's someone else's fully managed execution environment you use at a fraction of the cost. Lambda function containers are like the Oompa Loompas in *Charlie and the Chocolate Factory* — always working and available in the back end but never seen or interacted with.



Back

Next

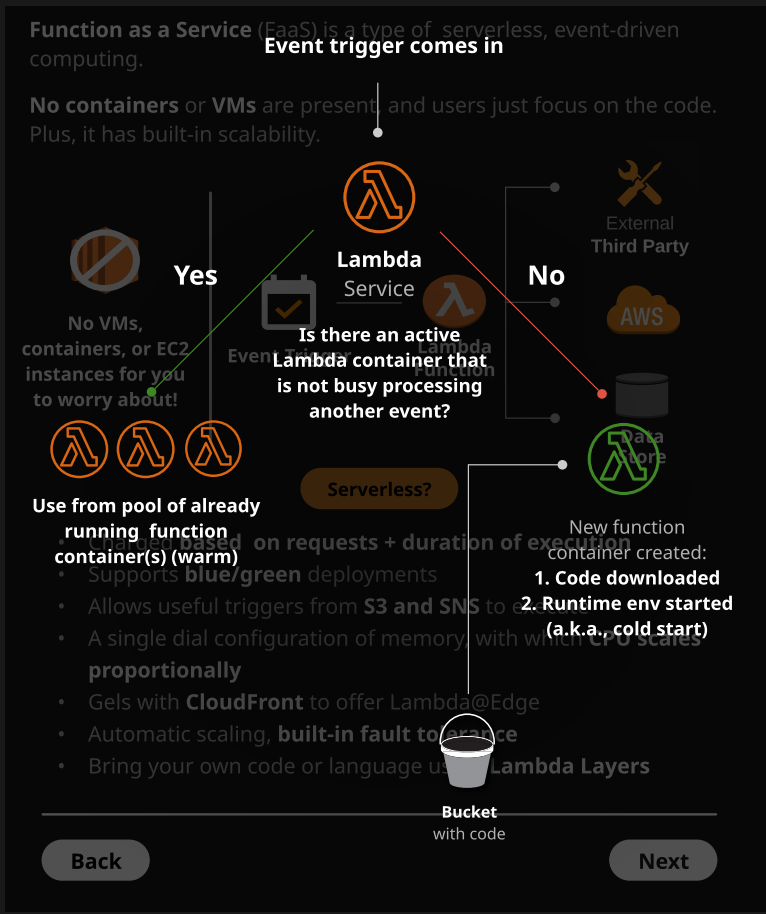
Application Layer: AWS Lambda Functions

Overview

Under the Hood: Lambda

Function as a Service (FaaS) is a type of serverless, event-driven computing.

No containers or **VMs** are present, and users just focus on the code. Plus, it has built-in scalability.



- Pricing based on requests + duration of execution
- Supports **blue/green** deployments
- Allows useful triggers from **S3 and SNS** to e...
- A single dial configuration of memory with which CPU scales **proportionally**
- Gels with **CloudFront** to offer Lambda@Edge
- Automatic scaling, **built-in fault tolerance**
- Bring your own code or language u... **Lambda Layers**

Back

Next

Back to Main

Linux Academy

Application Layer: AWS Lambda Functions

Creating a Lambda Function and Lambda IAM Execution Role

1. **Recipe for Lambda function creation:**
   - Provide the **runtime or language** you want to code in.
   - Provide an **IAM role Lambda** can assume to carry out tasks, such as creating logs in CloudWatch Logs and streaming to the logs in **CloudWatch** log groups created by it.
2. **Other options set by default but can be modified:**
   - Memory size (one dial for controlling it all)
   - **Concurrency** across AWS account
   - Env variables/debugging (SQS DLQ/AWS X-Ray)
   - ZIP file or **S3** location of your code
   - Add triggers to **invoke the function**

Lambda IAM Execution Role

The **Lambda service** requires an IAM role with specific permissions so it can query or reach out to other AWS services on your behalf through your code. It's best practice to allow least possible access to any resource within AWS. At a minimum, Lambda requires **access to CloudWatch Logs** for **log streaming**.

Console vs. CLI: Lambda

Back

Next

Back to Main

Application Layer: AWS Lambda Functions

Creating a Lambda Function and Lambda IAM Execution Role

1. **Recipe for Lambda function creation:**
   - Provide the **runtime or language** you want to code in.
   - Provide an **IAM role Lambda** can assume to carry out tasks, such as creating logs in CloudWatch Logs and streaming to the logs in **CloudWatch** log groups created by it.
2. **Other options set by default but can be modified:**
   - Memory size (one dial for controlling it all)
   - **Concurrency** across AWS account
   - Env variables/debugging (SQS DLQ/AWS X-Ray)
   - ZIP file or **S3** location of your code
   - Add triggers to **invoke the function**

A quick note about creating Lambda functions via the **AWS Management Console** as opposed to the **AWS CLI**: The AWS console makes it easy and seamless to create a Lambda role on the fly; whereas with the CLI, you need to **create a Lambda service IAM role** and attach the relevant IAM policy with it to then pass into your **create-function** invocation.

Console vs. CLI: Lambda

Back

Next

Back to Main

Application Layer: AWS Lambda Functions

Event Triggers for AWS Lambda

Basically, integrations with other services to invoke Lambda functions.

Each service that integrates with Lambda **sends data to your Lambda function in JSON** as an event, which is passed as an "event" object into your handler function. It can be set from within **Lambda** functions or the **CloudWatch** events console.

Lambda can also:

- Read events from services such as: **Kinesis, DynamoDB and SQS**
- Be invoked by services **synchronously**
- Be invoked by services **asynchronously**

Synchronous Events **Asynchronous Events**

Application Layer: AWS Lambda Functions

Event Triggers for AWS Lambda

Event triggers can be from a number of sources, including S3, IoT, CloudFront, DynamoDB, and even external third-party tools, such as Datadog and PagerDuty via CloudWatch event bus and AWS EventBridge services.

Lambda can also:

- Read events from services such as: **Kinesis, DynamoDB and SQS**
- Be invoked by services **synchronously**
- Be invoked by services **asynchronously**

Synchronous Events **Asynchronous Events**

Back Next

Back to Main

Application Layer: AWS Lambda Functions

Event Triggers for AWS Lambda

Basically, integrations with other services to invoke Lambda functions.

Each service that integrates with Lambda **sends data to your Lambda function in JSON** as an event, which is passed as an "event" object into your handler function. It can be set from within **Lambda** functions or the **CloudWatch** events console.

Lambda can also:

- Read events from services such as: **Kinesis, DynamoDB and SQS**
- Be invoked by services **synchronously**
- Be invoked by services **asynchronously**

Synchronous Events **Asynchronous Events**

Synchronous services invoke Lambda, and then wait for a response or retry on failure. Examples of such services are **AWS ELB, CloudFront, and API Gateway.**

Application Layer: AWS Lambda Functions

Event Triggers for AWS Lambda

Basically, integrations with other services to invoke Lambda functions.

Each service that integrates with Lambda **sends data to your Lambda function in JSON** as an event, which is passed as an "event" object into your handler function. It can be set from within **Lambda** functions or the **CloudWatch** events console.

Lambda can also:

- Read events from services such as: **Kinesis, DynamoDB and SQS**
- Be invoked by services **synchronously**
- Be invoked by services **asynchronously**

Synchronous Events **Asynchronous Events**

For **asynchronous** services, Lambda queues the events before passing it to the function, and the service gets a success response as soon as the event is queued. The async service **doesn't care what happens afterwards.** Examples of such services are Amazon S3, SNS, SES, and CloudFormation.

Application Layer: AWS Lambda Functions

Testing and Debugging Lambda Functions

AWS X-Ray

AWS SAM CLI

AWS Cloud9

AWS CloudTrail

Lambda Function

Amazon CloudWatch Logs

AWS has been introducing new ways to test and debug Lambda functions:

- In the cloud using CloudWatch Logs, AWS X-Ray, and Cloud9 IDE
- Locally via AWS SAM CLI
- In the Lambda AWS console directly with test JSON data (can also be done via AWS CLI)

- You can use **logging statements** within your code to help troubleshoot.
- Investigate issues using **AWS CloudWatch metrics** for Lambda.

Back

Database

- **Amazon Lambda** does not offer data persistence, and it does not save states between transactions, which is why a persistent data resource is required.
- **Lambda** can write out to a persistent resource.

Candidates for persistent layer:

- **Amazon S3**
- **Amazon RDS (Aurora Serverless)**
- **Amazon DynamoDB**

Continuance of **data existence** after the process that created it has terminated.

INSERT/UPDATE/DELETE/CREATE

Lambda Function

AWS Aurora RDBMS

The Lambda function code connects with and carries out **modification actions** on the Aurora Serverless **RDBMS**.

Next

Features

Aurora Serverless manages a **warm pool of resources** in an AWS region to **minimize scaling time.**

You're billed in **Aurora capacity units (ACUs)**, which is a combination of processing and memory capacity. In creating Aurora Serverless, you set a minimum and maximum **ACU** that it will be scaled between.

Like other RDS databases, users receive a **database endpoint** after provisioning. The difference is there's no need to specify **DB instance size or class** when creating Aurora Serverless instances.

Features

- MySQL and PostgreSQL compatible.
- DB storage automatically varies between 10 GiB to 64 TiB.
- DB cluster parameter groups can be used to modify DB instance parameters.
- If there are no connections or transactions coming in, Aurora Serverless billing is paused. You only pay for the storage.
- By default, service is paused after five minutes of inactivity (can be modified).

Like other RDS databases, users receive a **database endpoint** after provisioning. The difference is there's no need to specify **DB instance size or class** when creating Aurora Serverless instances.

Aurora Serverless: Behind the Scenes

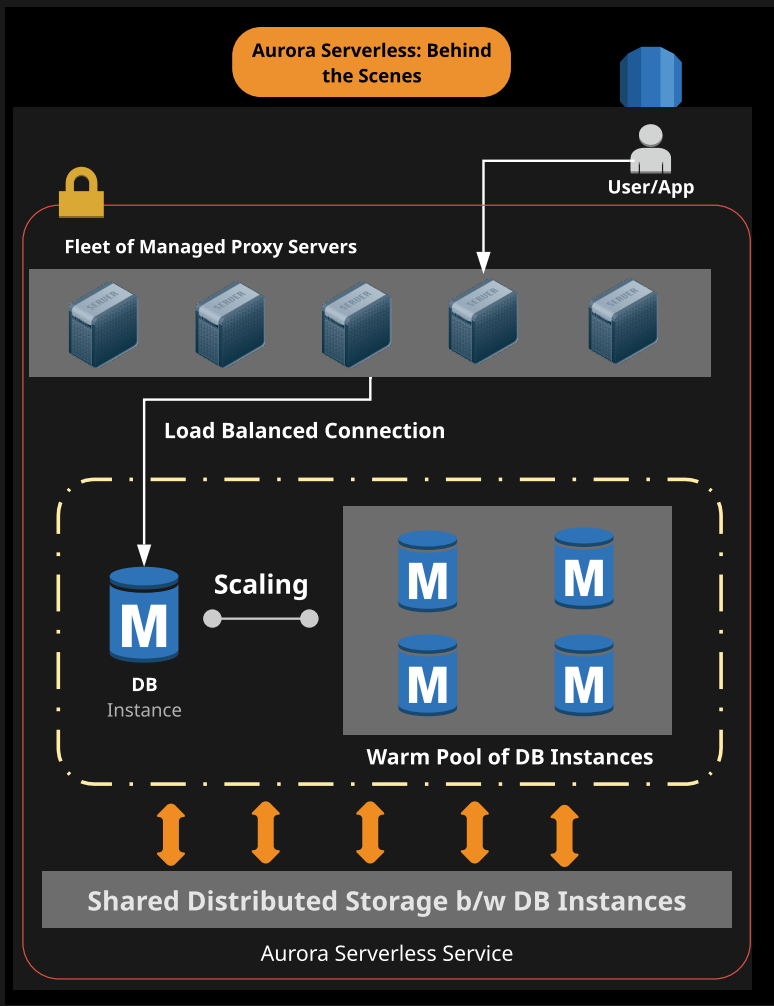
Aurora Serverless

Why RDS Aurora Serverless?

- Provisioning for **peak** capacity ==> **Expensive**
- Provisioning for **less than peak** capacity ==> **Business impact**
- Monitoring and scaling **manually** ==> **Risk/requires expertise**

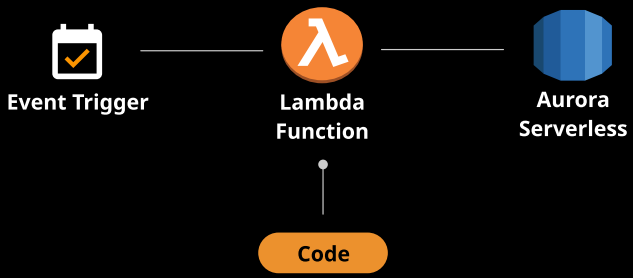
Aurora Serverless in a Nutshell

- Smooth scaling, no downtime.
- Scales as your load **increases/decreases**.
- RDS Data API allows integration with Lambda and other **AWS services.**
- Warm buffer pool is **replicated via shared distributed storage** so cutover to newly scaled instances is quick.
- Scaling is generally done if >70% CPU utilization or >90% connection utilization for a period of **five minutes**.
- A monitoring service actively monitors **all DB instances** in the back end for performance, resources, etc.
- You can choose to pause service or take capacity to zero to cut down costs when there's **no incoming traffic**.
- Excellent for intermittent, unpredictable **workloads.**



In almost all cases, you would want your database to be **restricted** and **only** accessible to your business logic applications or functions such as AWS Lambda.

Integration Workflow



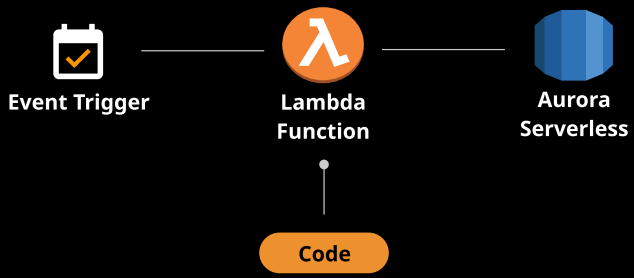
In our case, we will be using **Boto3**, which is an AWS Python SDK, in our Lambda code.

Points to note:

- **RDS credentials (user/pass) are stored in Secrets Manager as a secret, so Lambda will require access to Secrets Manager to get the value.**
- **RDS will need access to RDSDataService via the `rds-data:ExecuteStatement` action.**

In almost all cases, you would want your database to be **restricted** and **only** accessible to your business logic applications or functions such as AWS Lambda.

Integration Workflow

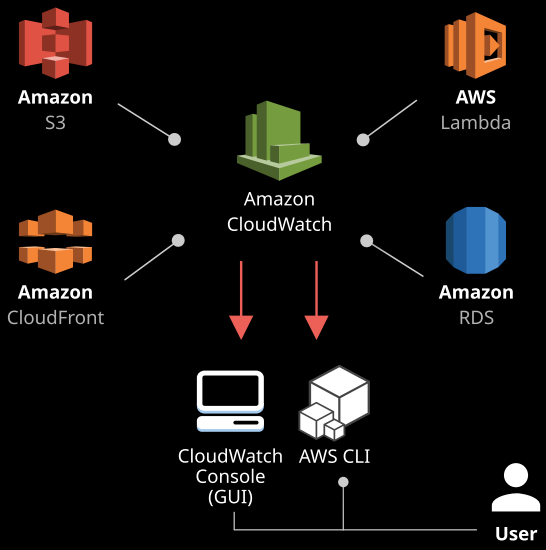



```
#NOT ACTUAL CODE
import boto3
client = boto3.client('rds-data')
.....
.....
def lambda_handler(event,context):
    response = client.execute_statement(database,
            resourceArn,
            secretArn,
            sql)
    return response
```

AWS CloudWatch and Metrics

Serverless Metrics

- **Monitoring service** for AWS cloud
- CloudWatch provides **useful metrics for all serverless services**, such as S3, CloudFront, Lambda, and Aurora RDS
- Allows creating **customized dashboards**
- Integrates with other AWS services for **alarms and notifications**



Back

If this course piqued your interest in IAM , Lambda or even Python Boto SDK for AWS be sure to check out our courses:

- **AWS IAM Deep Dive**

  (https://linuxacademy.com/cp/modules/view/id/180)
- **Lambda Deep Dive**

  (https://linuxacademy.com/cp/modules/view/id/204)
- **Automating AWS with Lambda, Python and Boto3**

  (https://linuxacademy.com/cp/modules/view/id/313)

Connect with us and your fellow learners!

Linux Academy community Slack
https://linuxacademy-community-slack.herokuapp.com

Linux Academy Community
https://linuxacademy.com/cp/socialize