SWINBURNE UNIVERSITY OF TECHNOLOGY

COS80011

Cloud-Native Application Architectures

Lecture 5 Cloud Native Computation

includes material from

ACD Module 7 – Introduction to Containers

ACD Module 9 – Developing Event-driven Solutions with AWS Lambda

Assignment 1 - Due date: Tues 20 April 9 am

- **Deploying a web app by three different approaches in 3 parts of this assignment.**
  - **VM implementation using AWS EC2**
  - **Serverless implementation using AWS Lambda**
  - **Container based deployment using AWS Elastic Container Service and Elastic Container Registry**
- **Structured Assignment**
  - **Much of the code is provided. Emphasis on deployment models and architecture**
- **Discussion board provided**
  - **Feel free to ask and answer questions**
- **Due date: 9 am 20 April** to Canvas. Demonstration required.
  Late submission penalty: 10% of total available marks per day.

SWINBURNE UNIVERSITY OF TECHNOLOGY

Last week

- **Relational vs non-relational 'NoSQL' databases**
  - High-level Comparison
  - NoSQL data models
- ***RDS (in brief)***
- **Dynamo DB**
  - Key Concepts
  - Partitions and data distribution
  - Secondary indexes
  - Read/write throughput
  - Streams and global tables
  - Basic operation on tables

Quizzes:
ACF Mod 8
ACA Mod 5
ACD Mod 5

SWINBURNE UNIVERSITY OF TECHNOLOGY

This week – Cloud native Computation

- Containers (ACD Module 7)
  - ☐ Introduction to containers
  - ☐ Containers vs. hardware virtualization
  - ☐ Microservices: Use case for containers
  - ☐ Amazon container orchestration services
- Serverless computation (ACD Module 9)
  - ☐ Introduction to serverless computing with AWS Lambda
  - ☐ Execution models for invoking Lambda functions
  - ☐ AWS Lambda permissions
  - ☐ Overview of authoring and configuring Lambda functions
  - ☐ Overview of deploying Lambda functions

Quizzes:
ACD Mod 7 Containers / Docker
ACD Mod 9 Serverless / Lambda

SWINBURNE UNIVERSITY OF TECHNOLOGY

Introduction to containers

© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

aws academy

Part 1: Introduction to containers

Module objectives

- Describe the history, technology, and terminology behind containers
- Differentiate containers from bare-metal servers and virtual machines
- Identify the characteristics of a microservices architecture
- Recognize the drivers for using container-based workloads
- Host a basic website by using Docker containers

© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

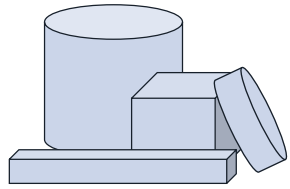
After completing this module, you will be able to:

- Describe the history, technology, and terminology behind containers
- Differentiate containers from bare-metal servers and virtual machines
- Identify the characteristics of a microservices architecture
- Recognize the drivers for using container-based workloads
- Host a basic website by using Docker containers

Shipping with containers

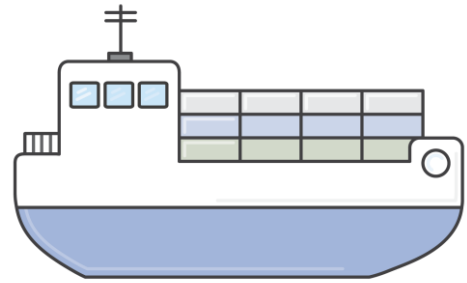
Before shipping containers:

- Goods were shipped in a variety of vessels with no standardized weight, shape, or size.
- Transporting goods was slow, inefficient, and costly.



After shipping containers:

- Uniform size of shipping containers made it more efficient to load, unload, and stack.
- Containers improved efficiency, increased productivity, and reduced costs.



Container ship

© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

In the physical world, a *container* is a standardized unit of storage. It sounds like a generic term, but it's important to understand the enormous impact the container had on the shipping industry.

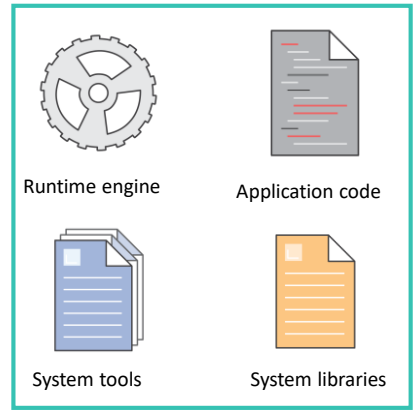
Before the introduction of the modern shipping container in 1956, transporting goods from one point to another was a challenge. Goods were shipped in sacks, crates, cartons, drums, casks, barrels, and boxes of various weights, shapes, and sizes. They often had to be loaded by hand into whatever vessel was carrying them. The vessel wouldn't know how much cargo it could take until all the cargo was loaded. Transporting goods this way was slow, inefficient, and costly.

Shipping containers revolutionized the shipping industry. Their uniform size made it much more efficient to load, unload, and stack them. Containers could also be easily moved between ships, trucks, and railroad cars. In this way, containers improved efficiency, increased productivity, and reduced costs.

Source: http://www.worldshipping.org/about-the-industry/history-of-containerization/the-birth-of-intermodalism

What is a container?

A standardized unit of software

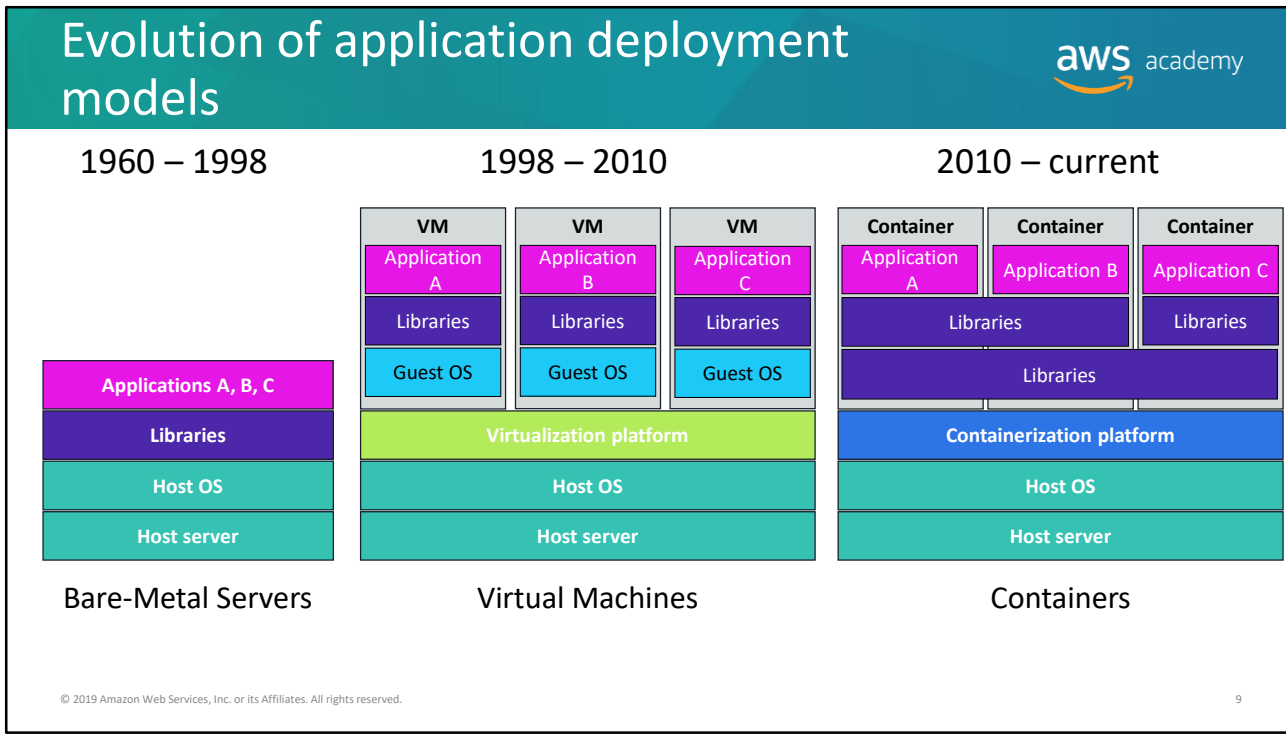


© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

Now, move from the physical world to the virtual world. In computing platforms, a *container* is a standardized unit of software designed to run quickly and reliably on any computing environment that is running the containerization platform.

Containers are a method of operating system virtualization that allow you to run an application and its dependencies in resource-isolated processes. A container is a lightweight, standalone software package that contains everything a software application needs to run, such as the application code, runtime, system tools, system libraries, and settings. Containers can help ensure that applications deploy quickly, reliably, and consistently regardless of deployment environment.

A single server can host several containers that all share the underlying host system's operating system (OS) kernel. These containers might be services that are part of a larger enterprise application, or they might be separate applications that are running in their isolated environment.



Technical maturity is often associated with increased levels of abstraction.

With *bare-metal servers*, the architectural layers, such as the infrastructure and application software layers, are built. For example, you install an OS on top of your server hardware, install any shared libraries on the OS, and then install the applications that use those libraries. The issue with this architecture is that it's inefficient. Your hardware costs are the same whether you are running at 0 percent utilization or 100 percent utilization. All of your applications must compete for the same resources, and you must keep the versions of your libraries in sync with all your applications. If one application requires an updated version of a library that is incompatible with other applications running on that host, then you run into problems.

You can increase agility by putting a virtualization platform over the host OS. Now, you have isolated applications and their libraries, which have their own full OS inside a virtual machine (VM). This improves utilization because you can add more VMs to run on top of the existing hardware, which greatly reduces your physical footprint. The downside to VMs is that the virtualization layer is heavy. In this example, you now have three operating systems on the physical host server, instead of one. That means more patching, more updates, and significantly more space being taken up on the physical host. There's also significant redundancy: you have installed potentially the same OS three times, and potentially the same library three times.

Enter containers. The container runtime shares the host operating system's kernel. This enables you to create container images using file system layers. Containers are lightweight, efficient, and fast. They can be spun up and spun down faster than VMs, which allows for better utilization of the underlying hardware. You can share libraries when needed, but you can also have library isolation for your applications. Containers are also highly portable.

Because containers isolate software from other layers, their code runs identically across different environments—from developing and staging all the way to production.

Docker as a virtualization platform

- Is a lightweight container virtualization platform
- Provides tools to create, store, manage, and run containers
- Integrates with automated build, test, and deployment pipelines

© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

Containers have become popular due to the rise of Docker as a virtualization platform. Docker was released in March 2013. It is a lightweight container virtualization platform that provides tooling for creating, storing, managing, and running containers. It is easy to integrate with automated build, test, and deployment pipelines.

Docker container benefits

- Portable runtime application environment
- Application and dependencies can be packaged in a single, immutable artifact
- Ability to run different application versions with different dependencies simultaneously
- Faster development and deployment cycles
- Better resource utilization and efficiency

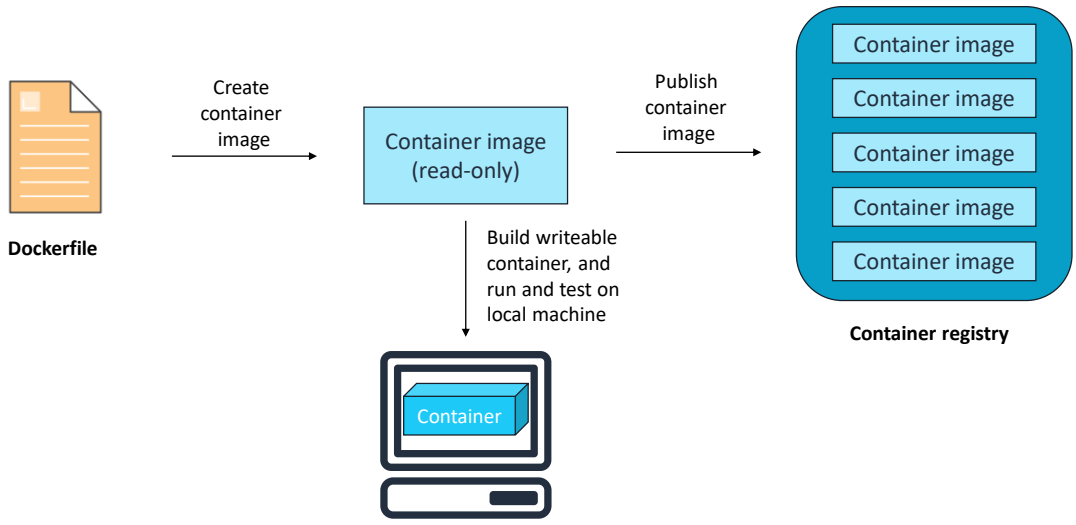
© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

To summarize some of the most important benefits of Docker containers:

- Docker is a portable runtime application environment.
- You can package an application and its dependencies into a single, immutable artifact called an image.
- After you create a container image, it can go anywhere that Docker is supported.
- You can run different application versions with different dependencies simultaneously.

These benefits lead to much faster development and deployment cycles, and better resource utilization and efficiency. All of these abilities come back to agility.

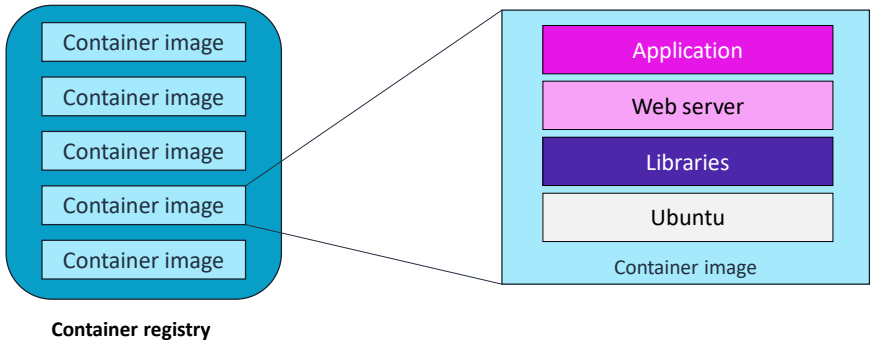
Docker container terminology



© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

Docker containers are created from a read-only template that is called an *image*. A Docker container is an instance of an image. Images are built from a *Dockerfile*, which is a plain text file that specifies all of the components that are included in the container. You can create images from scratch, or you can use images that others created and published to a public or private container registry.

Images: Templates for containers



© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

An image is usually based on another image, with some customization. For example, you might build an image that's based on the Ubuntu Linux image in the registry. However, it installs a web server and your application, in addition to the essential configuration details to make your application run.

Dockerfile example 1

```
# Start with the Ubuntu latest image
FROM ubuntu:latest

# Output hello world message
CMD echo "Hello World!"
```

© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

To build your own image, you create a Dockerfile using a simple syntax to define how to create the image and run it. Each instruction in a Dockerfile creates a read-only layer in the image. Here are a few examples.

In this simple example, you start with the Ubuntu latest image that is already created for you, and hosted on Docker Hub or some other site. The only thing you do is add a command to echo the message *Hello World!* after the container is spun up.

Dockerfile example 2

```
# Start with open JDK version 8 image
FROM openjdk:8

# Copy the jar file that contains your code from
your system to the container
COPY /hello.jar /usr/src/hello.jar

# Call Java to run your code
CMD java –cp /usr/src/hello.jar
Org.example.App
```

© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

Here's a slightly more involved example where you want to run a Java application. You start with the open Java Development Kit (JDK) version 8 image, copy the .jar file that contains your code from your system to the container, and then call Java to run your code. When this container is instantiated, it runs the Java application.

Dockerfile example 3

```
# Start with CentOS7 image
FROM centos:7

# Update the OS and install Apache
RUN yum –y update && yum –y install httpd

# Expose Port 80—the port that the web server “listens to”
EXPOSE Port 80

# Copy shell script and give it executable permissions
ADD run-httpd.sh /run-httpd.sh
RUN chmod –v +x /run-httpd.sh

# Run shell script
CMD [“/run-httpd.sh”]
```

© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

Here’s a more real-world example of a Dockerfile. You start with the CentOS 7 image. Next, you update the OS and install Apache. Then, you expose Port 80. Finally, you copy your shell script for your application and give it executable permissions. After the container is instantiated, the command will run the shell script.

Dockerfile example 3

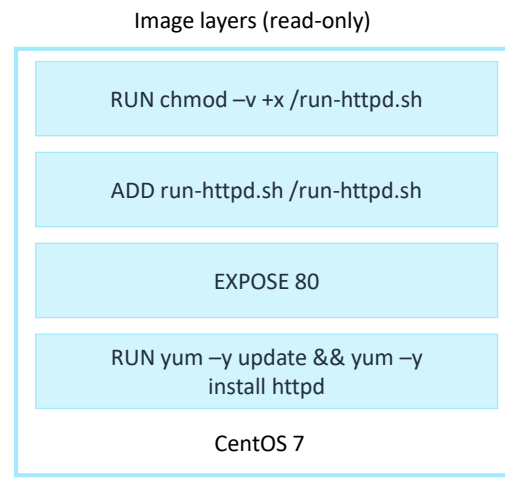
```
# Start with CentOS7 image
FROM centos:7

# Update the OS and install Apache
RUN yum –y update && yum –y install httpd

# Expose Port 80
EXPOSE Port 80

# Copy shell script and give it executable
permissions
ADD run-httpd.sh /run-httpd.sh
RUN chmod –v +x /run-httpd.sh

CMD [“/run-httpd.sh”]
```



Each line of the Dockerfile adds a layer to the image.

© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

Here’s what the previous example looks like in terms of layers. Each instruction in the Dockerfile creates a layer. The first layer is the base layer, which includes the software update and the Apache installation. The next layer opens and exposes Port 80. The third layer includes command to copy the shell script. Finally, the last layer makes the shell script executable. These layers are all read-only, which makes the container image an immutable object.

If you change the Dockerfile and rebuild the image, only the layers that have changed are rebuilt. This feature is part of what makes container images so lightweight, small, and fast compared to other virtualization technologies.

Dockerfile example 3

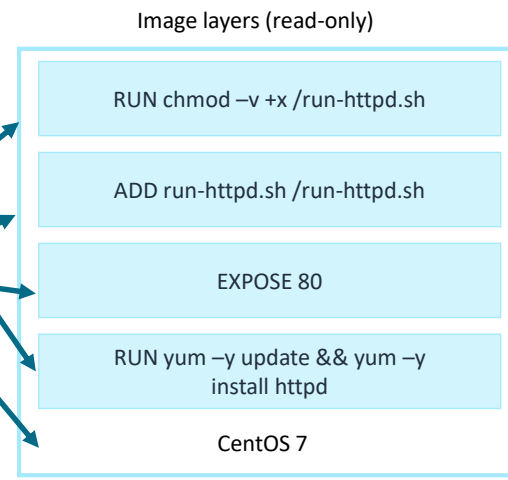
```
# Start with CentOS7 image
FROM centos:7

# Update the OS and install Apache
RUN yum –y update && yum –y install httpd

# Expose Port 80
EXPOSE Port 80

# Copy shell script and give it executable
permissions
ADD run-httpd.sh /run-httpd.sh
RUN chmod –v +x /run-httpd.sh

CMD [“/run-httpd.sh”]
```



Each line of the Dockerfile adds a layer to the image.

© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

Here’s what the previous example looks like in terms of layers. Each instruction in the Dockerfile creates a layer. The first layer is the base layer, which includes the software update and the Apache installation. The next layer opens and exposes Port 80. The third layer includes command to copy the shell script. Finally, the last layer makes the shell script executable. These layers are all read-only, which makes the container image an immutable object.

If you change the Dockerfile and rebuild the image, only the layers that have changed are rebuilt. This feature is part of what makes container images so lightweight, small, and fast compared to other virtualization technologies.

Docker CLI commands

aws academy

Command	Information	Command	Information
`docker build` *	Build an image from a Dockerfile.	`docker start`	Start a container.
`docker images`	List images on Docker host.	`docker logs`	View container log output.
`docker run`	Run an image.	`docker port`	List container port mappings.
`docker ps` *	List running containers.	`docker tag` *	Tag an image.
`docker exec`	Run a command in a container.	`docker push` *	Push image to a registry.
`docker stop`	Stop a running container.	`docker inspect`	Inspect container information.

* *indicates most common Docker commands.*

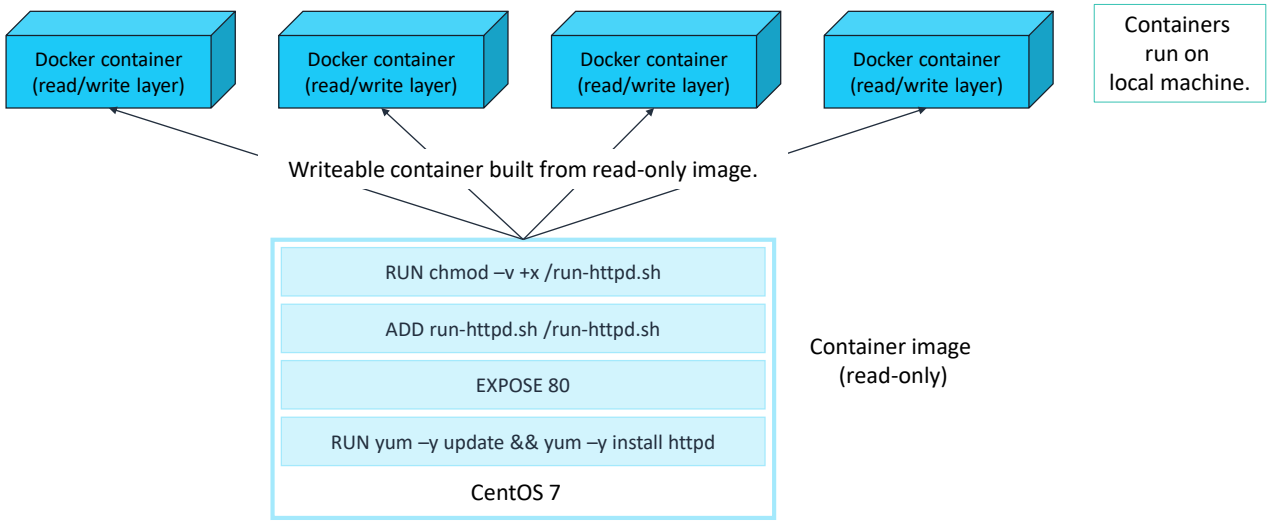
© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

You can run Docker command line interface (CLI) commands from a Bash terminal to manage your Docker images and containers.

For example, you can build an image from a Dockerfile by running *`docker build`*. You can then verify your image by running *`docker images`*. To launch a container from the image, run *`docker run`*. To verify that the container is running, enter *`docker ps`*. After you launch a container, you can interact with it in a variety of ways. To open a Bash prompt on your running container, use *`docker exec`*. You can also stop a running container (*`docker stop`*) and start it again (*`docker start`*). To view your container logs, enter *`docker logs`*. To list the port mappings for your container, run *`docker port`*. Run *`docker tag`* to tag your image and use *`docker push`* to push it to a registry.

For more information on Docker CLI commands, see the Docker documentation: https://docs.docker.com/engine/reference/commandline/cli/

Summary: Docker images vs. containers



© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

To review the terminology:

- A *container image* is a read-only, immutable template that is highly portable. You can port it to any environment that supports Docker, and it can be stored in a registry for easy reuse.
- A *container* is an instance of an image. You can spin up one container or several containers based on that image.
- Each container has a thin, read/write layer on top of the existing image when it is instantiated. This architecture is what makes the actual process of spinning up the containers fast. Most of the actual work is read-only because of the file system layers.

Docker uses a copy-on-write system, where changed files are written to the read/write layer of the container. The underlying image remains unchanged. This is why multiple containers can share access to the same underlying image, but still have their own data state. When the container is deleted, this writable container is also deleted. The read/write layer of the container enables your applications to function properly while they are running, but it's not designed for long-term data storage. Persistent data should be stored in a volume somewhere. Consider a container as a discrete compute unit, not a storage unit.

This week – Cloud native Computation

- Containers
  - Introduction to containers
  - Containers vs. hardware virtualization
  - **Microservices: Use case for containers**
  - Amazon container orchestration services
- Serverless computation
  - Introduction to serverless computing with AWS Lambda
  - Execution models for invoking Lambda functions
  - AWS Lambda permissions
  - Overview of authoring and configuring Lambda functions
  - Overview of deploying Lambda functions

SWINBURNE UNIVERSITY OF TECHNOLOGY

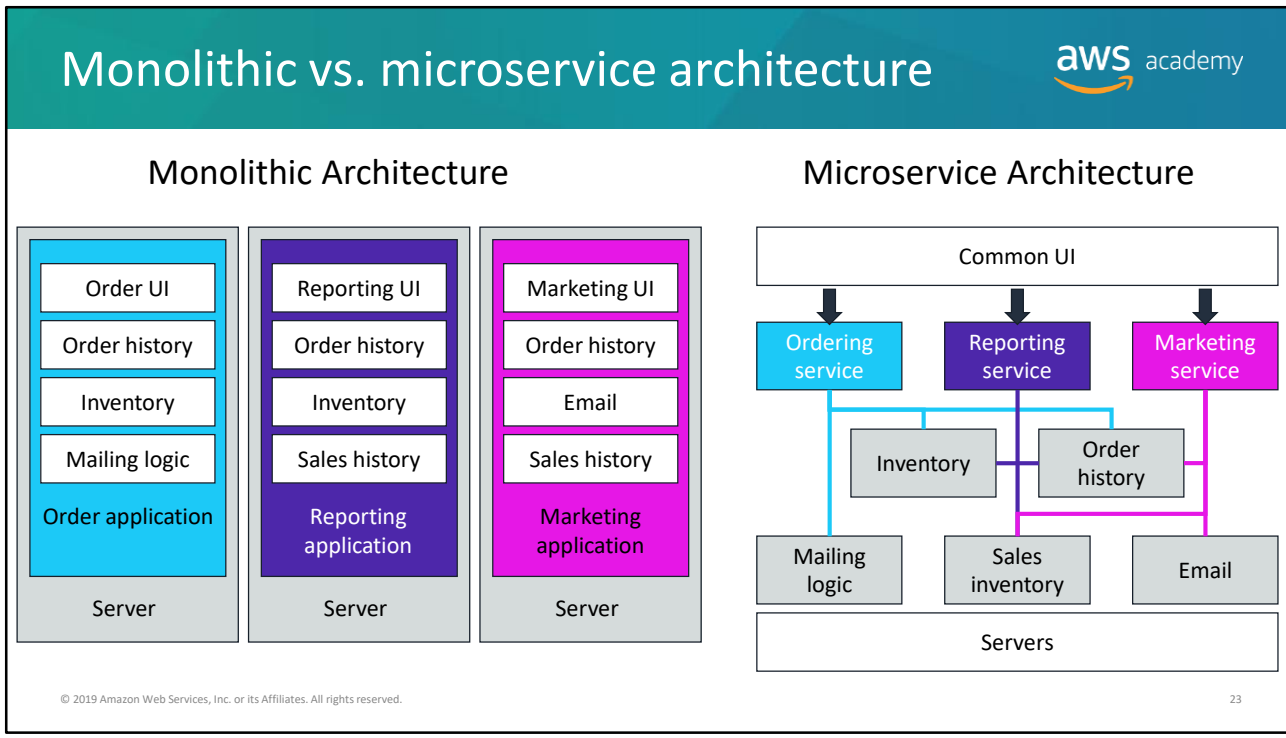
Microservices – Use case for containers

© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

aws academy

Part 3: Microservices – Use case for containers

Containers are often associated with microservice environments. In this section, you will learn what a microservices architecture is, and why containers are so well suited for it.



One of the strongest factors driving the growth of containers is the rise of microservices architectures. Microservices are an architectural and organizational approach to software development that is designed to speed up deployment cycles. The microservices approach fosters innovation and ownership, and improves the maintainability and scalability of software applications.

Consider this example of a *traditional monolithic architecture*. For each application, all the processes are tightly coupled and run as a single service. This means that if one process of an application experiences a spike in demand, the entire architecture must be scaled. Adding or improving features becomes more complex as the code base grows, which limits experimentation and makes it difficult to implement new ideas. Monolithic architectures also add risk for application availability because having many dependent and tightly coupled processes increase the impact of a single process failure. You can see that there is redundancy of function across different applications.

Now, consider the same three applications running in a *microservices architecture*. Each application is built as an independent component that runs as a service, and communicates by using lightweight API operations. Each service performs a single function that can support multiple applications. Because the services run independently, they can be updated, deployed, and scaled to meet the demand for specific functions of an application.

A microservice architecture allows for much quicker iteration, automation, and overall agility. Start fast, fail fast, and recover fast.

Characteristics of microservices

- Decentralized, evolutionary design
- Smart endpoints, dumb pipes
- Independent products, not projects
- Designed for failure
- Disposable
- Development and production parity

© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

When you consider the characteristics of a well-designed microservice architecture, you could see why containers and microservices go so well together.

- *Decentralized, evolutionary design* – Each container uses the language and technology that is best suited for the functioning of the service, instead of requiring users to use a specific language or a specific technology. Each component or system in the architecture is evolved separately, instead of updating the system in a monolithic style.
- *Smart endpoints and dumb pipes* – There is no enterprise service bus. Data is not transformed when it's going between services. The service that receives the data should be smart enough to handle whatever data is sent.
- *Independent products, not projects* – In contrast to the traditional waterfall project model, think of a microservice as a separate product with its own inputs and outputs. No assumptions are made about the runtime environment. Containers enable you to package all your dependencies and libraries into a single immutable object.
- *Designed for failure* – Everything fails all the time. Services are designed to be resilient, redundant. Services are also designed to handle bad input, or to handle cases when the service that the microservice wants to communicate with is not there.
- *Disposable* – If something goes wrong, you can gracefully shut down what has failed and spin up a new instance. You start fast, fail fast, and release any file handlers. The development pattern is like a circuit breaker. In traditional architectures, servers are named, deployed, evaluated, and remain pretty stable. By contrast, the cloud architecture is disposable and transitory. Containers are added and removed, workloads change, and resources are temporary because they constantly change.
- *Development and production parity* – Containers can make development, testing, and production environments consistent. This facilitates DevOps, in which a containerized application that works on a developer's system will work the same way on a production system.

To summarize, microservices and containers go well together. Containers are the underlying technology that powers modern microservice architectures. Likewise, with microservice architectures, developers can take full advantage of containers.

This week – Cloud native Computation

- Containers
  - Introduction to containers
  - Containers vs. hardware virtualization
  - Microservices: Use case for containers
  - **Amazon container orchestration services**
- Serverless computation
  - Introduction to serverless computing with AWS Lambda
  - Execution models for invoking Lambda functions
  - AWS Lambda permissions
  - Overview of authoring and configuring Lambda functions
  - Overview of deploying Lambda functions

SWINBURNE UNIVERSITY OF TECHNOLOGY

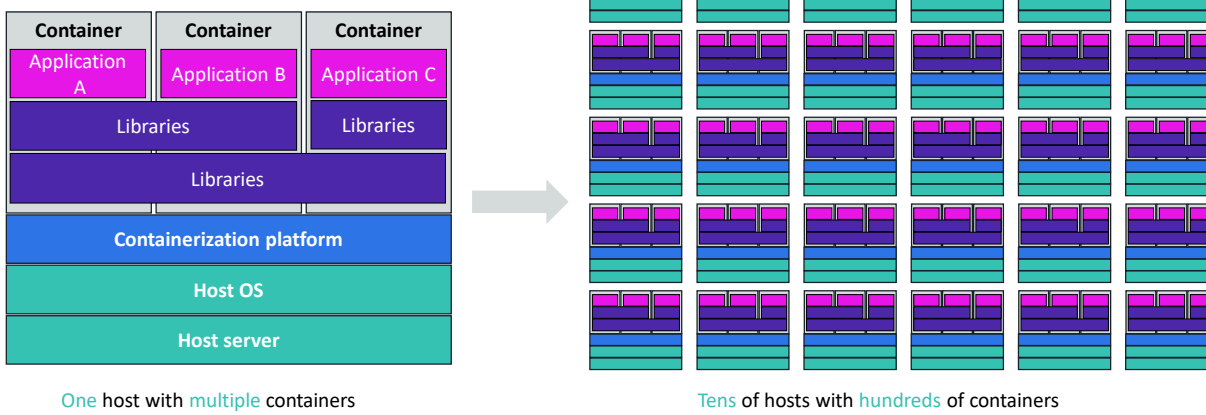
Amazon container services

© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

aws academy

Part 4: Amazon container services

The challenge with managing containers



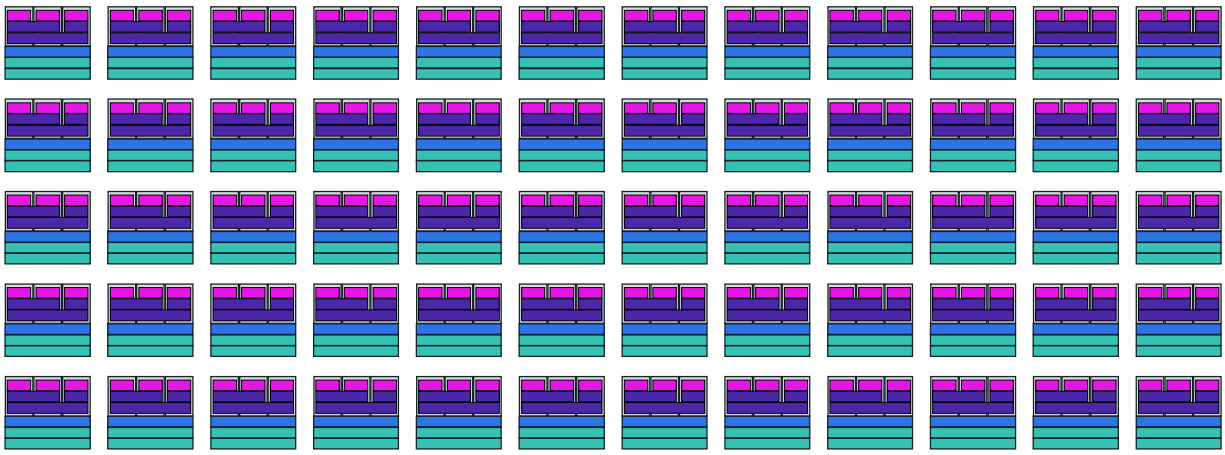
One host with multiple containers

Tens of hosts with hundreds of containers

© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

Running one or two containers on a single host is simple. However, what happens when you move into a staging environment where you have tens of hosts with possibly hundreds of containers?

The challenge with managing containers



Hundreds of hosts with thousands of containers

© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

Now, imagine a full production environment with hundreds of hosts and maybe thousands of containers. You are now managing an enterprise-scale, clustered environment, and cluster management is difficult. You need a way to place your containers intelligently on instances to maximize availability, resilience, and performance, which means that you must know the state of everything in your system. You need a way to manage your containers at scale.

Container management platforms

- Scheduling and placement
- Service integration

Amazon ECS

Docker swarm

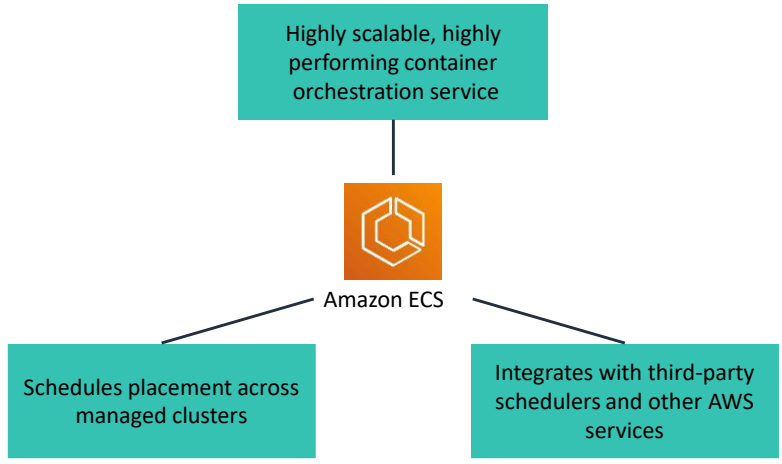
Kubernetes

© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

This situation is where *container management platforms* come in. They handle the scheduling and placement of containers based on the underlying hardware infrastructure and needs of the application. The container management platforms integrate with other services, such as services for networking, persistent storage, security, monitoring, and logging.

Numerous options exist, such as Amazon Elastic Container Service (Amazon ECS), native tools such as Docker Swarm, and open-source platforms such as Kubernetes. The management platform is arguably the most important choice you will make when you architect a container-based workload.

Amazon Elastic Container Service

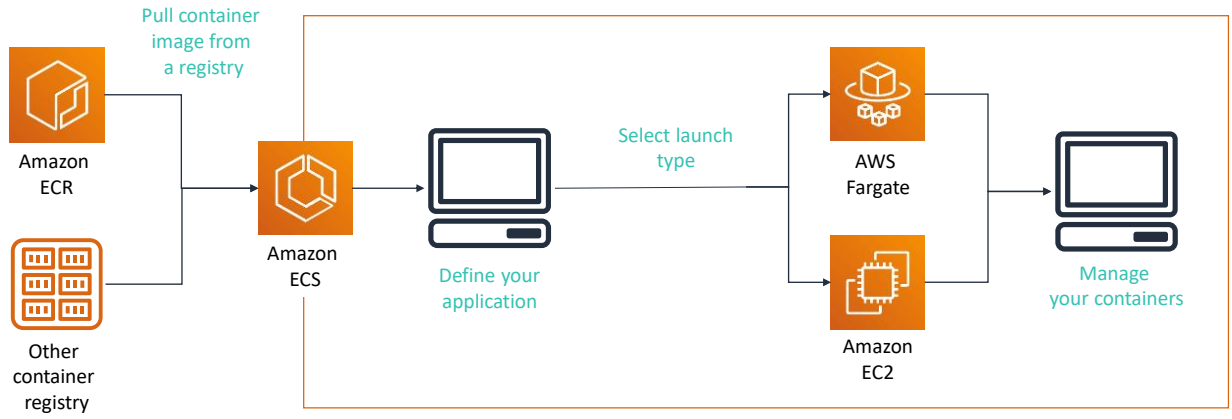


© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

Amazon ECS is a highly scalable, highly performing container orchestration service that supports Docker containers. The service enables you to run and scale containerized applications on AWS. You can use Amazon ECS to schedule the placement of containers across a managed cluster of Amazon Elastic Compute Cloud (Amazon EC2) instances. Amazon ECS provides its own schedulers, but it can also integrate with third-party schedulers to meet business or application-specific requirements. Amazon ECS is also tightly integrated with other AWS services, such as AWS Identity and Access Management (IAM), Amazon CloudWatch, and Amazon Route 53.

For more information about Amazon ECS, see the product page: https://aws.amazon.com/ecs/

Amazon ECS solution architecture



© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

Here is a high-level overview of Amazon ECS.

First, container images are pulled from a registry. This registry can be Amazon Elastic Container Registry (Amazon ECR)—which is one of many AWS services that integrates with Amazon ECS—or a third-party or private registry.

Next, you define your application. Customize the container images with the necessary code and resources, and then create the appropriate configuration files to group. Then, define your containers as short-running tasks or long-running services within Amazon ECS.

When you are ready to bring your services online, you select one of two launch types. The Fargate launch type provides a near-serverless experience, where the infrastructure that supports your containers is completely managed by AWS. AWS manages the placement of your tasks on instances, and makes sure that each task has the appropriate amount of CPU and memory. With Fargate, you can focus on the tasks and the application architecture, instead of worrying about the infrastructure.

The EC2 launch type is useful when you want more control over the infrastructure that supports your tasks. When you use the EC2 launch type, you create and manage clusters of EC2 instances to support your containers. You also define the placement of containers across your cluster based on your resource needs, isolation policies, and availability requirements. You have more granular control over your environment without needing to operate your own cluster management and configuration management systems, or worry about scaling your management infrastructure.

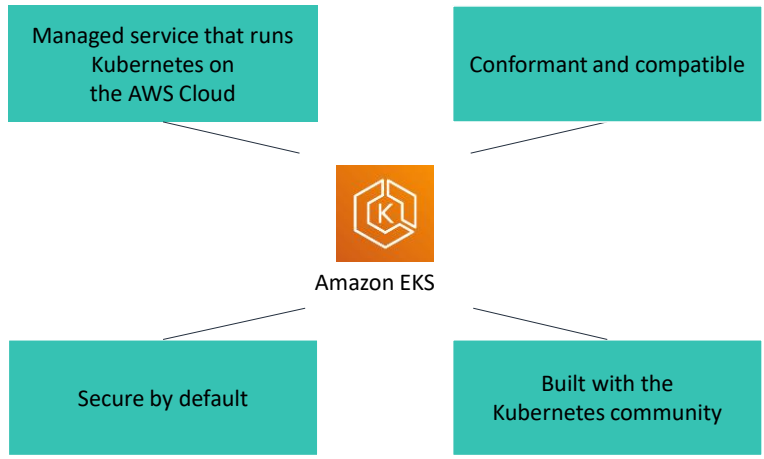
You can mix and match the two launch types as needed within your application. For example, you can launch services that have more predictable resource requirements by using EC2, and

launch other services that are subject to wide swings in demand by using Fargate. Regardless of the launch type that you use, Amazon ECS manages your containers for availability, and it can scale your application as necessary to meet demand.

Finally, you can use Amazon ECS to manage your containers. Amazon ECS scales your application and manages your containers for availability.

For more information about AWS Fargate, see the product page: AWS Fargate: https://aws.amazon.com/fargate/
For more information about the Fargate and EC2 launch types, see the AWS Documentation:
https://docs.aws.amazon.com/AmazonECS/latest/developerguide/launch_types.html

Amazon Elastic Container Service for Kubernetes



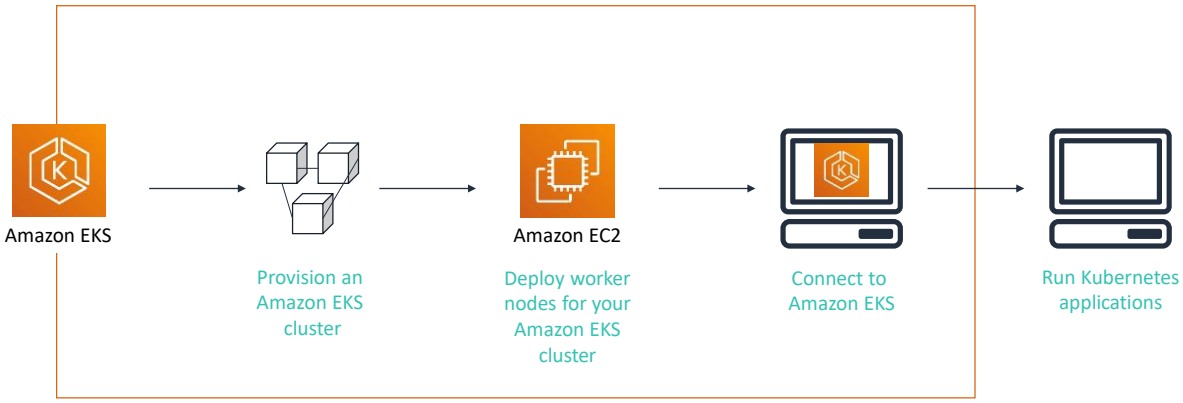
© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

As mentioned earlier, Kubernetes is an open-source container management platform that enables you to deploy and manage containerized applications at scale. Kubernetes manages clusters of EC2 instances, and it runs containers on those instances with processes for deployment, maintenance, and scaling. By using Kubernetes, you can run any type of containerized application by using the same toolset on premises and in the cloud.

You can use AWS services to run Kubernetes in the cloud, including scalable and highly available virtual machine infrastructure, community-baked service integrations, and Amazon Elastic Container Service for Kubernetes (Amazon EKS). Amazon EKS is a managed service that runs the Kubernetes management infrastructure across multiple AWS Availability Zones to reduce the chance of having a single point of failure. Amazon EKS is certified Kubernetes conformant, so you can use existing tooling and plugins from partners and the Kubernetes community. Applications that run on any standard Kubernetes environment are fully compatible, and they can be migrated to Amazon EKS. Amazon EKS automatically sets up secure and encrypted channels to your worker nodes, which makes your infrastructure running on Amazon EKS secure by default. AWS actively works with the Kubernetes community, including making contributions to the Kubernetes code base that help Amazon EKS users take advantage of AWS services and features.

For more information about Amazon EKS, see the product page: https://aws.amazon.com/eks/

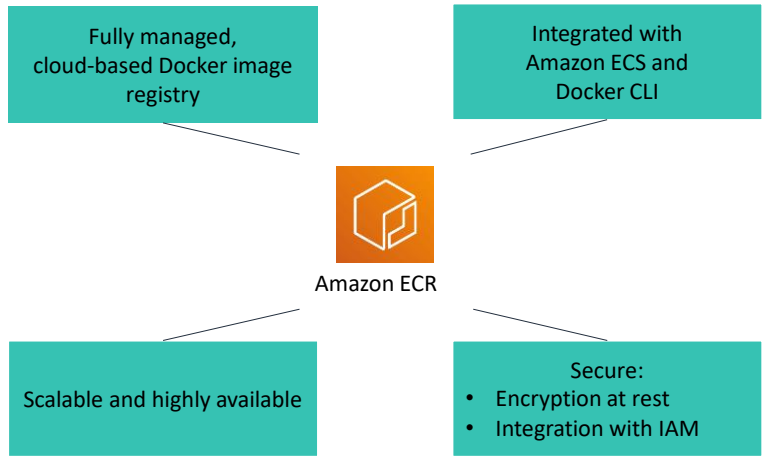
Amazon Elastic Container Service for Kubernetes



© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

When you select Amazon EKS as your container management service, you provision an Amazon EKS cluster and deploy Amazon EC2 worker nodes (that is, worker machines) for your Amazon EKS cluster. You then connect to Amazon EKS and run your Kubernetes applications.

Amazon Elastic Container Registry



© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

Amazon ECR is a fully-managed, cloud-based Docker image registry that makes it easy for you to store, manage, and deploy Docker container images. Amazon ECR integrates with Amazon ECS and the Docker CLI, which allows you to simplify your development and production workflows. You can push your container images to Amazon ECR by using the Docker CLI from your development machine, and Amazon ECS can pull them directly for production deployments. Amazon ECR reduces the need to operate your own container repositories or worry about scaling the underlying infrastructure. Amazon ECR hosts your images in a highly available and scalable architecture, which allows you to reliably deploy containers for your applications. Amazon ECR is also secure. Amazon ECR transfers your container images over HTTPS, and it automatically encrypts your images at rest. You can configure policies to manage permissions for each repository and restrict access to IAM users, roles, or other AWS accounts.

For more information about Amazon ECR, see the product page:
https://aws.amazon.com/ecr/

Creating an Amazon ECR repository and pushing an image

aws academy

```
# Create a repository called hello-world
> aws ecr create-repository --repository-name hello-world

# Build and tag an image
> docker build -t hello-world .
> docker tag hello-world aws_account_id.dkr.ecr.us-east-1.amazonaws.com/hello-world

# Authenticate Docker to your Amazon ECR registry
> aws ecr get-login
> docker login -u AWS -p <password> -e none aws_account_id.dkr.ecr.us-east-
1.amazonaws.com

# You can skip the `docker login` step if you have amazon-ecr-credential-helper
setup.

# Push an image to your repository
> docker push aws_account_id.dkr.ecr.us-east-1.amazonaws.com/hello-world
```

© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

You can programmatically access Amazon ECR from the AWS Command Line Interface (AWS CLI) and by using the application programming interfaces (APIs). You can use the AWS CLI and APIs to create, monitor, and delete repositories and set repository permissions. You can perform these same actions in the Amazon ECR console, which can be accessed from the Amazon ECS console. Amazon ECR integrates with the Docker CLI, which allows you to push, pull, and tag images on your development machine.

This example shows how to create a repository called *hello-world* with Amazon ECR. It uses Docker CLI commands to build and tag an image, and then push it into the repository. In order to push an image into Amazon ECR, you must first authenticate the Docker client to your Amazon ECR registry.

For information on:

- Using the AWS CLI with Amazon ECR, see the AWS Documentation: https://docs.aws.amazon.com/AmazonECR/latest/userguide/ECR_AWSCLI.html#AWSCLI_create_repository
- How to authenticate Amazon ECR repositories with the Docker CLI, see this blog post: https://aws.amazon.com/blogs/compute/authenticating-amazon-ecr-repositories-for-docker-cli-with-credential-helper/.

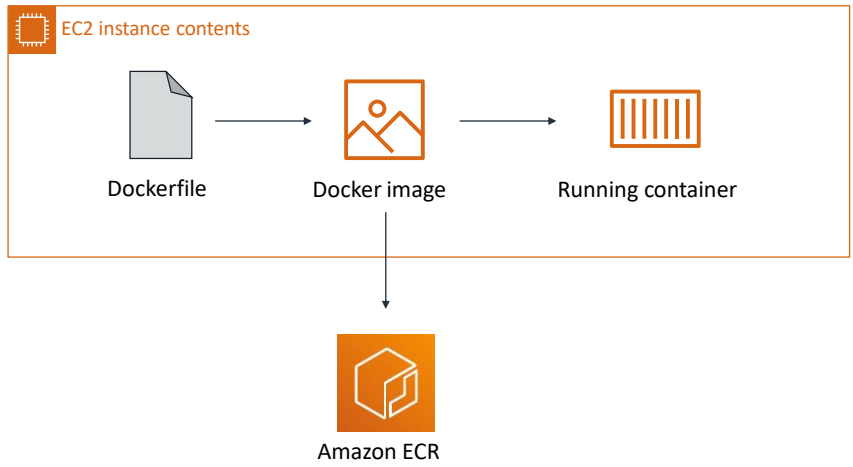
Lab: Working with Docker containers

© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

aws academy

In this lab, you will learn how to host a basic website by using Docker containers.

Lab overview



© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

After completing this lab, you will be able to:

- Launch an EC2 instance with user-data.
- Create a Dockerfile.
- Create a Docker image from the Dockerfile.
- Run a container from a Docker image.
- Interact with and administer your containers.
- Create an Amazon ECR repository.
- Authenticate the Docker client to Amazon ECR.
- Push a Docker image to the Amazon ECR repository.

This week – Cloud native Computation

- Containers
  - ☐ Introduction to containers
  - ☐ Containers vs. hardware virtualization
  - ☐ Microservices: Use case for containers
  - ☐ **Amazon container orchestration services**
- **Serverless computation (ACD Module 9)**
  - ☐ **Introduction to serverless computing with AWS Lambda**
  - ☐ **Execution models for invoking Lambda functions**
  - ☐ **AWS Lambda permissions**
  - ☐ **Overview of authoring and configuring Lambda functions**
  - ☐ **Overview of deploying Lambda functions**

SWINBURNE UNIVERSITY OF TECHNOLOGY

Introduction to serverless computing with AWS Lambda

© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

aws academy

Part 1: Introduction to serverless computing with AWS Lambda

In this section, you will learn about serverless computing.

Serverless computing

aws academy

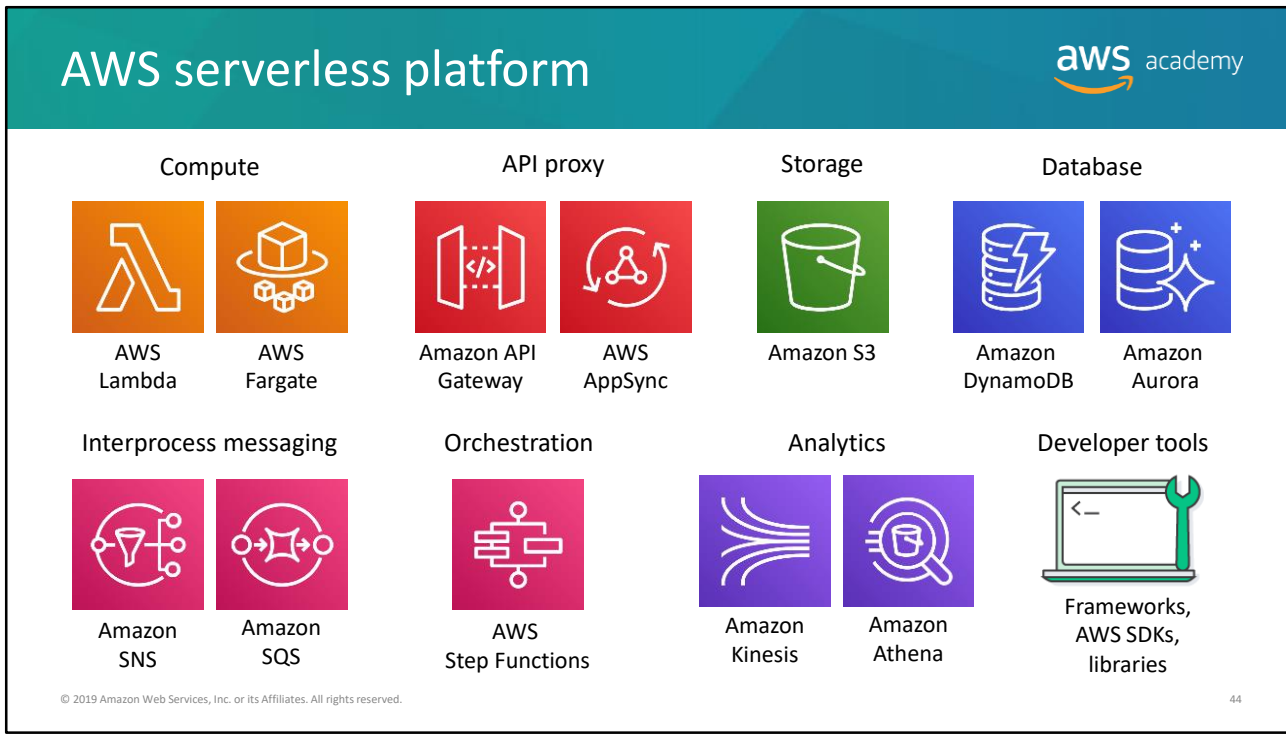
Traditional deployment and operations	Serverless deployment and operations
• Configure an instance	• Configure an instance
• Update OS	• Update OS
• Install application platform	• Install application platform
• Build and deploy applications	• Build and deploy applications
• Configure automatic scaling and load balancing	• Configure automatic scaling and load balancing
• Continuously patch, secure, and monitor servers	• Continuously path, secure, and monitor servers
• Monitor and maintain applications	• Monitor and maintain applications

© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

One of the benefits of cloud computing is that you can abstract the infrastructure layer so you don't have to worry about operational tasks. For example, you can choose an Amazon Elastic Compute Cloud (Amazon EC2) instance type and configure it with Elastic Load Balancing and EC2 Auto Scaling to handle demand. Serverless computing extends the infrastructure abstraction even further.

Serverless computing allows you to focus on the code for your applications. You do not have to manage instances, operating systems, or servers. Everything that is required to run and scale your application with high availability is handled for you. Serverless computing eliminates infrastructure management tasks, such as server or cluster provisioning, patching, operating system maintenance, and capacity provisioning. The reduced overhead lets you experiment and innovate faster.

Additionally, with serverless computing, your code only runs when it's needed. You don't pay for any infrastructure when the code isn't running.



The AWS serverless platform includes a number of fully managed services that are tightly integrated with Lambda and are well-suited for serverless applications. Developer tools—such as the AWS Serverless Application Model (AWS SAM)—help simplify the deployment of serverless applications.

For more information about AWS serverless offerings, see https://aws.amazon.com/serverless/.

AWS services defined:

- Amazon Simple Storage Service (Amazon S3)
- Amazon Simple Notification Service (Amazon SNS)
- Amazon Simple Queue Service (Amazon SQS)
- AWS software development kits (AWS SDKs)

AWS Lambda

AWS Lambda

- Lets you run code without provisioning or managing servers
- Triggers your code in response to events
- Scales automatically
- Provides built-in code monitoring and logging with Amazon CloudWatch

© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

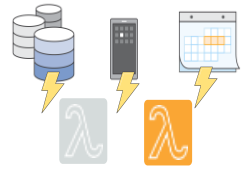
AWS Lambda is the compute service for serverless.

AWS Lambda:
- Lets you run code without provisioning or managing servers
- Triggers your code in response to events that you configure
- Scales automatically based on demand
- Incorporates built-in monitoring and logging with Amazon CloudWatch

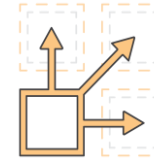
For more information about AWS Lambda, see the AWS Lambda product page:
https://aws.amazon.com/lambda/

AWS Lambda features

Enables you to bring your own code



Integrates with and extends other AWS services



Offers flexible resource and concurrency model

Offers flexible permissions model

Provides built-in availability and fault tolerance

Reduces need to pay for idle resources

© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

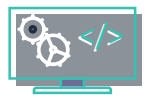
AWS Lambda provides the following features:

- *Enables you to bring your own code* – The code that you write for Lambda isn't written in a new language that you have to learn. Development in Lambda isn't tightly coupled to AWS, so you can easily port code in and out of AWS. With Lambda, you bring your code (written in your language of choice—such as Node.js, Java, Python, C#, Go, or Ruby), custom runtimes, and libraries.
- *Integrates with and extends other AWS services* – From within your Lambda function, you can do anything traditional applications can do, e.g., calling an AWS software development kit (SDK), invoking a third-party application programming interface (API), etc.
- *Offers flexible resource and concurrency model* – Lambda scales in response to events. You configure memory settings, and AWS handles the details, such as CPU, network, and I/O throughput.
- *Offers flexible permissions model* – Lambda uses AWS Identity and Access Management (IAM) to securely grant access to your resources and give you fine-grained control for invoking your functions.
- *Provides built-in availability and fault tolerance* – Because Lambda is a fully managed service, high availability and fault tolerance are integrated into the service, without any additional configuration on your part.
- *Reduces the need to pay for idle resources* – Lambda functions only run when triggered, so you never pay for idle capacity.

AWS Lambda use cases

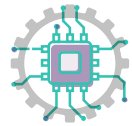
Web applications

- Static websites
- Complex web applications



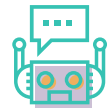
Backends

- Applications and services
- Mobile
- IoT



Data processing

- Real-time file processing
- Real-time streaming processing
- Machine learning inference



Chatbots

- Chatbot logic

Amazon Alexa

- Voice-enabled applications
- Alexa skills

IT automation

- Policy engines
- Extending AWS services
- Infrastructure management

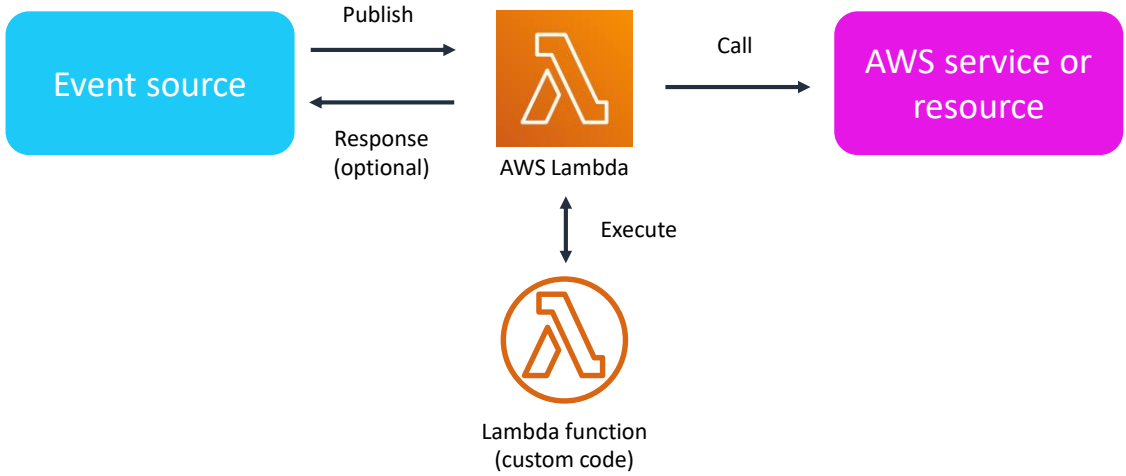
© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

You can use AWS Lambda in a variety of ways:

- *Web applications* – By combining AWS Lambda with other AWS services, you can build powerful web applications that automatically scale up and down and run in a highly available configuration across multiple data centers. There is no administrative effort required for scalability, backups, or multi-data-center redundancy.
- *Backends* – You can build serverless backends using AWS Lambda to handle web, mobile, Internet of Things (IoT), and third-party application programming interface (API) requests. You can build backends using AWS Lambda and Amazon API Gateway to authenticate and process API requests. Lambda makes it easy to create rich, personalized application experiences.
- *Data processing* – You can use AWS Lambda to execute code in response to triggers, such as changes in data, shifts in system state, or actions by users. Lambda can be directly triggered by AWS services, such as Amazon S3, Amazon DynamoDB, Amazon Kinesis, Amazon SNS, and Amazon CloudWatch. It can also be orchestrated into workflows by AWS Step Functions. This allows you to build a variety of real-time processing systems for serverless data.
- *Chatbots* – Amazon Lex is a service for building conversational interfaces into any application using voice and text. Amazon Lex uses the same deep learning technologies that power Amazon Alexa, and it integrates with AWS Lambda.
- *Amazon Alexa* – You can use AWS Lambda to build the backend for custom Alexa skills.
- *IT automation* – You can use AWS Lambda to automate repetitive processes by triggering them with events or by running them on a fixed schedule. You can automatically update the firmware of hardware devices, start and stop Amazon Elastic Compute Cloud (Amazon EC2) instances, schedule security group updates, or automate your test and deployment pipeline.

For case studies, see the AWS Lambda product page: https://aws.amazon.com/lambda/

How AWS Lambda works



© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

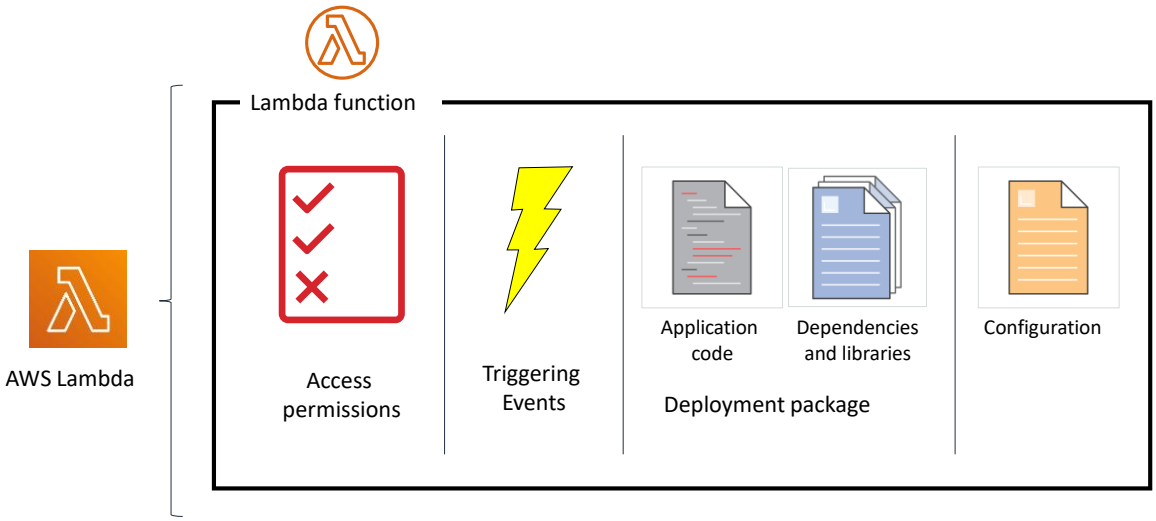
Lambda functions and event sources are the core components of AWS Lambda. An *event source* is the entity that publishes events to AWS Lambda, and a *Lambda function* is the custom code you provide that processes the events. AWS Lambda executes your Lambda function on your behalf.

Note about event sources: AWS Lambda integrates with other AWS services to invoke functions. You can configure triggers to invoke a function in response to resource lifecycle events, respond to incoming HTTP requests, consume events from a queue, or run on a schedule.

See the AWS Documentation for details about:

- Supported event sources: https://docs.aws.amazon.com/lambda/latest/dg/lambda-services.html
- Scheduled events: https://docs.aws.amazon.com/lambda/latest/dg/with-scheduled-events.html

Lambda functions



© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

As a developer, you create Lambda functions that are managed by the AWS Lambda service. When you create a function, you define the permissions for the function and specify which events trigger the function. You also create a deployment package that includes your application code and any dependencies and libraries that are necessary to execute your code. Finally, you configure execution parameters such as memory, time out, and concurrency. When your function is invoked, Lambda will provide an execution environment based on the runtime and configuration options that you selected.

For more information on the AWS Lambda execution environment, see the AWS Documentation: https://docs.aws.amazon.com/lambda/latest/dg/current-supported-versions.html

This week – Cloud native Computation

- Containers
  - Introduction to containers
  - Containers vs. hardware virtualization
  - Microservices: Use case for containers
  - Amazon container orchestration services
- Serverless computation (ACD Module 9)
  - Introduction to serverless computing with AWS Lambda
  - **Execution models for invoking Lambda functions**
  - AWS Lambda permissions
  - Overview of authoring and configuring Lambda functions
  - Overview of deploying Lambda functions

SWINBURNE UNIVERSITY OF TECHNOLOGY

Execution models for invoking Lambda functions

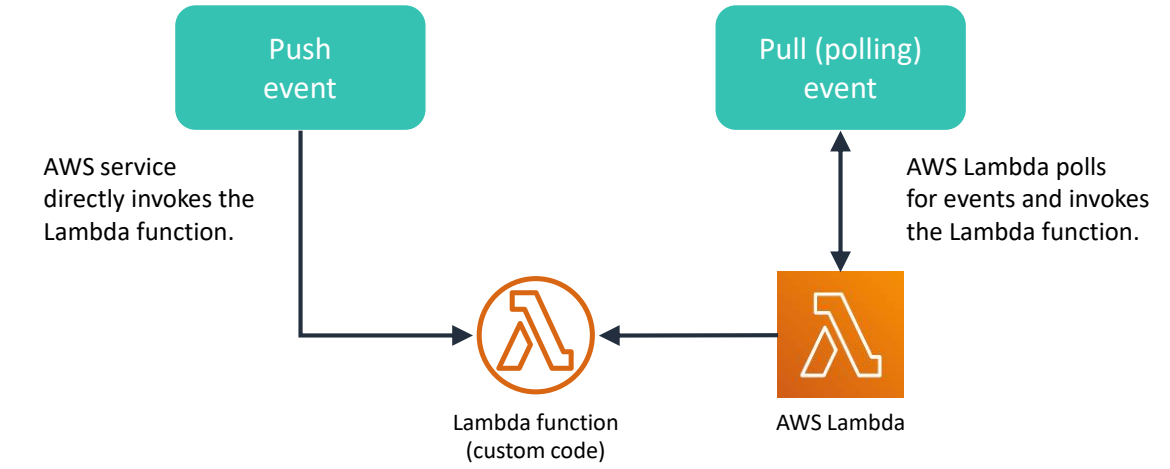
© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

aws academy

Part 3: Execution models for invoking Lambda functions

In this section, you will learn how to recognize AWS Lambda execution models.

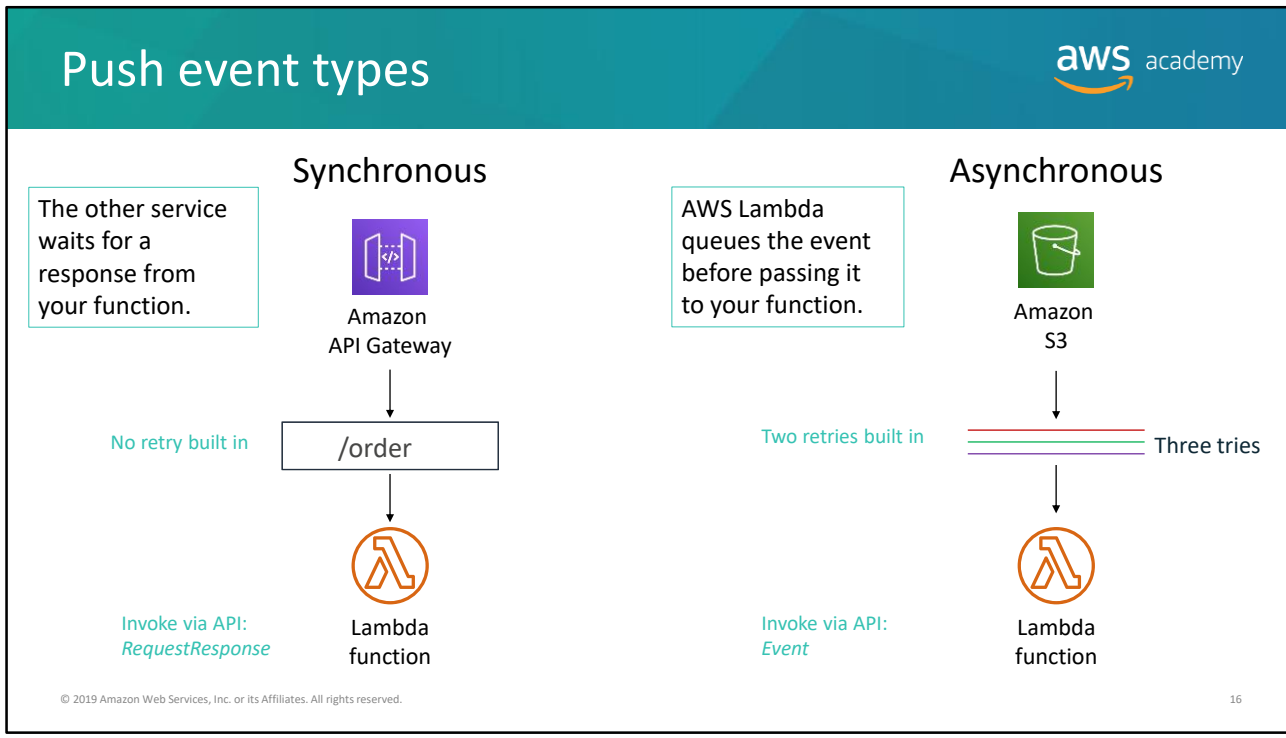
Push and pull execution models



© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

Each event source invokes a Lambda function by using either a push or a pull execution model:

- *Push model* – An event source directly invokes the Lambda function when the event occurs.
- *Pull (or polling) model* – An event source puts information into a stream or queue. AWS Lambda polls the stream or queue and invokes your Lambda function when it detects an event.



Some services invoke your function directly.

For *synchronous* invocation, the other service waits for the response from your function. Consider an example where Amazon API Gateway is the event source. In this case, when a client makes a request to your API, that client expects a response immediately. With this execution model, there is no built-in retry in Lambda. You must manage your retry strategy in your application code.

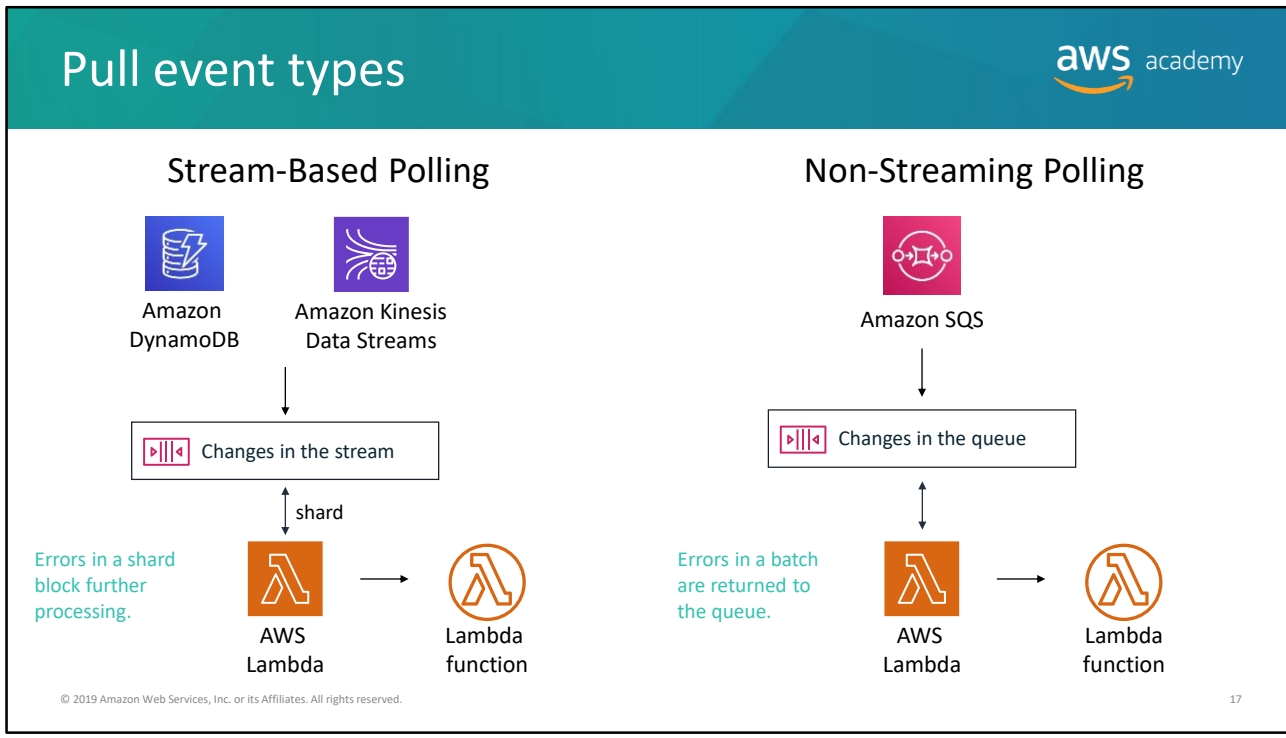
- Examples of synchronous event sources: Elastic Load Balancing, Amazon Cognito, Amazon Lex, Amazon Alexa, Amazon API Gateway, AWS CloudFormation, and Amazon CloudFront, and Amazon Kinesis Data Firehose.

For *asynchronous* invocation, AWS Lambda queues the event before passing it to your function. The other service gets a success response as soon as the event is queued and isn't aware of what happens afterwards. If an error occurs, AWS Lambda will automatically retry the invocation twice. It can send failed events to a dead-letter queue that you configure.

- Examples of asynchronous event sources: Amazon Simple Storage Service (Amazon S3), Amazon Simple Notification Service (Amazon SNS), Amazon Simple Email Service (Amazon SES), AWS CloudFormation, Amazon CloudWatch Logs, Amazon CloudWatch Events, AWS CodeCommit, and AWS Config

In both cases, you invoke your Lambda function using the *Invoke* operation, and you can specify the invocation type as synchronous or asynchronous. When you use an AWS services as a trigger, the invocation type is predetermined for each service. You have no control over the invocation type that these event sources use when they invoke your Lambda function.

For more information how to invoke Lambda functions, see the AWS Lambda Developer Guide: https://docs.aws.amazon.com/lambda/latest/dg/invoking-lambda-functions.html



In the polling model, event sources put information into a stream or queue. AWS Lambda polls the stream or queue. If it finds records, it will deliver the payload and invoke the Lambda function. In this model, the Lambda service itself is pulling data from the stream or queue for processing by the Lambda function.

- Stream-based event sources include Amazon DynamoDB and Amazon Kinesis Data Streams. Stream records are organized into shards. AWS Lambda polls the stream for records and attempts to invoke the function. If there's a failure, AWS Lambda will not read any new shards until the failed batch of records expires or is processed successfully.
- The non-streaming event source is Amazon SQS. AWS Lambda polls the queue for records. If the invocation fails or times out, then the failed message is returned to the queue. Lambda will keep retrying the failed message until it's processed successfully or the message retention period expires. If the message expires, it will either go to the dead letter queue (if configured) or it will be discarded.

You can create a mapping between an event source and your Lambda function. AWS Lambda will read items from the event source and trigger the function. For more information about the *CreateEventSourceMapping* operation, see the AWS Lambda Developer Guide: https://docs.aws.amazon.com/lambda/latest/dg/API_CreateEventSourceMapping.html

This week – Cloud native Computation

- Containers
  - Introduction to containers
  - Containers vs. hardware virtualization
  - Microservices: Use case for containers
  - Amazon container orchestration services
- **Serverless computation (ACD Module 9)**
  - **Introduction to serverless computing with AWS Lambda**
  - **Execution models for invoking Lambda functions**
  - **AWS Lambda permissions**
  - **Overview of authoring and configuring Lambda functions**
  - **Overview of deploying Lambda functions**

This week – Cloud native Computation

- Containers
  - ☐ Introduction to containers
  - ☐ Containers vs. hardware virtualization
  - ☐ Microservices: Use case for containers
  - ☐ Amazon container orchestration services
- Serverless computation (ACD Module 9)
  - ☐ Introduction to serverless computing with AWS Lambda
  - ☐ Execution models for invoking Lambda functions
  - ☐ **AWS Lambda permissions**
  - ☐ Overview of authoring and configuring Lambda functions
  - ☐ Overview of deploying Lambda functions

SWINBURNE UNIVERSITY OF TECHNOLOGY

AWS Lambda permissions

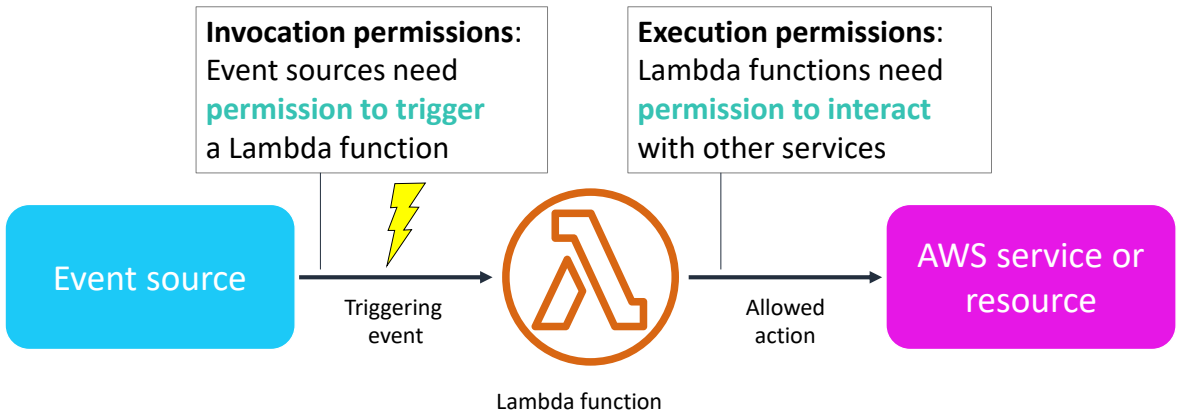
© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

aws academy

Part 4: AWS Lambda permissions

In this section, you will learn about the permissions that allow access to your Lambda functions.

AWS Lambda permissions



© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

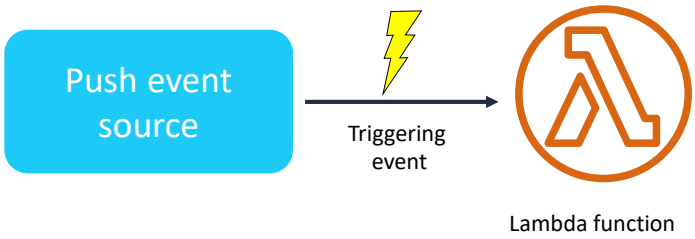
AWS Lambda has two types of permissions. First, event sources need permission to trigger a Lambda function (invocation permissions). Next, Lambda functions need permission to interact with other AWS services and resources (execution permissions).

Invocation and execution permissions are handled through AWS Identity and Access Management (IAM).

For more information about AWS Lambda permissions, see the AWS Documentation: https://docs.aws.amazon.com/lambda/latest/dg/lambda-permissions.html

Invocation permissions

IAM resource policy gives permission to invoke the function.



Resource (function) policy:

- Associated with a push event source
- Created automatically when you add a trigger to a Lambda function
- Allows the event source to take the *lambda:InvokeFunction* action

© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

An *IAM resource policy* tells the Lambda service which push event sources have permission to invoke the Lambda function. Resource policies also make it easy to grant access to the Lambda function across AWS accounts. For example, if you need an S3 bucket in account A to invoke your Lambda function in account B, you can create a resource policy that allows account A to invoke the function in account B.

The resource policy for a Lambda function is called a *function policy*. When you add a trigger to your Lambda function from the AWS Lambda console, the function policy will be generated automatically. It allows the event source to take the *lambda:InvokeFunction* action.

For information about using resource-based policies for AWS Lambda, see the AWS Documentation: https://docs.aws.amazon.com/lambda/latest/dg/access-control-resource-based.html

Resource (function) policy example

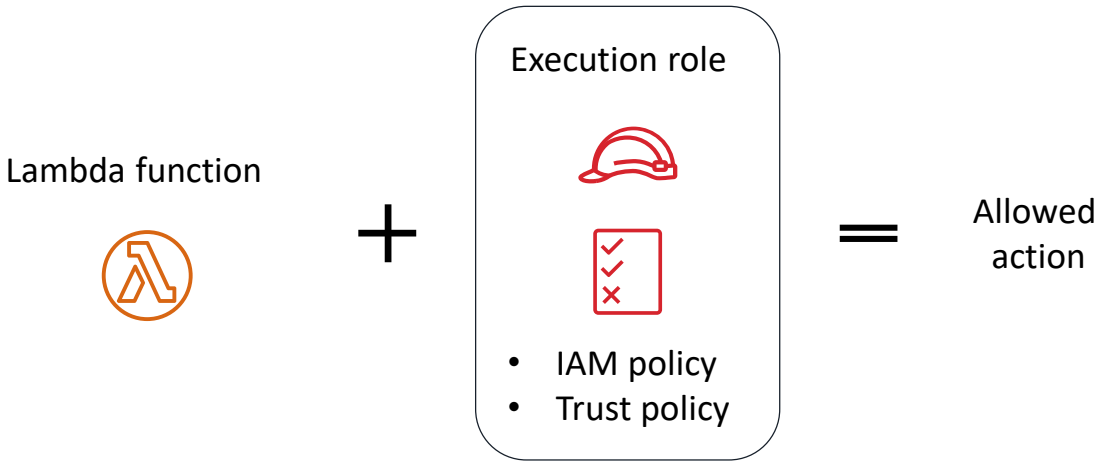
```
{
        "Version": "2012-10-17",
        "Id": "default",
        "Statement": [
                {
                        "Sid": "lambda-fd269e28-988b-4d2b-96ae-eabcd7dc399c",
                        "Effect": "Allow",
                        "Principal": {
                                "Service": "s3.amazonaws.com"
                        },
                        "Action": "lambda:InvokeFunction",
                        "Resource": "arn:aws:lambda:function:myFirstFunction",
                        "Condition": {
                                "ArnLike": {
                                        "AWS:SourceARN": "arn:aws:s3:::myBucket1"
                                }
                        }
                }
        ]
}
```

© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

In this example, the resource policy gives Amazon S3 permission to invoke the Lambda function called *myFirstFunction*.

Execution permissions

IAM execution role specifies what the Lambda function is permitted to do.



© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

The *IAM execution role* grants your Lambda function permission to access AWS services and resources. You select or create the execution role when you create a Lambda function. Two policies are attached to the execution role:

- *IAM policy* – Defines the actions that the Lambda function is allowed to take on another AWS service or resource, such as writing to a DynamoDB table.
- *Trust policy* – Allows the AWS Lambda service to assume the execution role.

To grant permission to AWS Lambda to AssumeRole, you must have permission for the *iam:PassRole* action.

For more information about the AWS Lambda execution role, see the AWS Documentation: https://docs.aws.amazon.com/lambda/latest/dg/lambda-intro-execution-role.html

Execution role example

aws academy

IAM Policy

```
{
        "Effect": "Allow",
        "Action": [
                "logs:CreateLogGroup",
                "logs:CreateLogStream",
                "logs:PutLogEvents"
        ],
        "Resource": "arn:aws:logs:*:*:*"
}
```

Trust Policy

```
{
    "Effect": "Allow",
    "Principal": {
                "Service": "lambda.amazonaws.com"
      },
    "Action": "sts:AssumeRole"
}
```

© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

Here are a couple of examples of the relevant policy lines for an execution role.

- In the example, the IAM policy allows your Lambda function to perform the following Amazon CloudWatch Logs actions: create a log group and log stream, and write to the log stream.
- In the example, the trust policy gives the AWS Lambda service permission to assume the role and invoke the Lambda function on your behalf.

This week – Cloud native Computation

- Containers
  - ☐ Introduction to containers
  - ☐ Containers vs. hardware virtualization
  - ☐ Microservices: Use case for containers
  - ☐ Amazon container orchestration services
- Serverless computation (ACD Module 9)
  - ☐ Introduction to serverless computing with AWS Lambda
  - ☐ Execution models for invoking Lambda functions
  - ☐ AWS Lambda permissions
  - ☐ **Overview of authoring and configuring Lambda functions**
  - ☐ **Overview of deploying Lambda functions**

SWINBURNE UNIVERSITY OF TECHNOLOGY

63

Overview of authoring and configuring Lambda functions

© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

aws academy

Part 5: Overview of authoring and configuring Lambda functions

In this section, you will learn the steps to author and configure and Lambda functions.

Create Lambda deployment package

Lambda function

Application code

Dependencies and libraries

Configuration

Supported runtimes:

- Node.js
- Python
- Ruby
- Java
- Go
- .NET
- Custom

© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

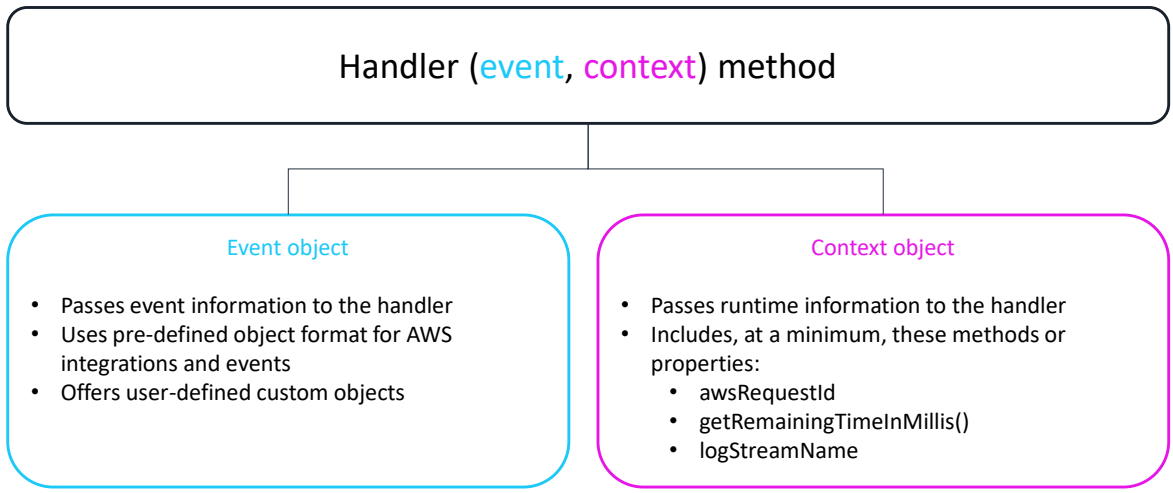
To create a Lambda function, you first create a Lambda function deployment package, which is a .zip or .jar file that consists of your code and any dependencies. With Lambda, you can use the programming language and integrated development environment (IDE) that you are most familiar with, and you can bring code that you've already written.

AWS Lambda supports multiple languages through the use of runtimes. Lambda supports the Node.js, Python, Ruby, Java, Go, and .NET runtimes. You can also implement a custom runtime if you want to use another language in Lambda. You choose a runtime when you create a function, and you can change runtimes by updating your function's configuration.

See the AWS Documentation for more information on:

- AWS Lambda runtimes: https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtimes.html
- How to create an AWS Lambda deployment package in your language of choice: https://docs.aws.amazon.com/lambda/latest/dg/deployment-package-v2.html

Specify the function handler



© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

When you create a Lambda function, you specify the function handler. The Lambda *function handler* is the entry point that AWS Lambda calls to start executing your Lambda function. The handler method always takes two objects: the event object and the context object.

- *Event object* – Provides information about the event that triggered the Lambda function. This could be a pre-defined object that an AWS service generates, or it could be a custom user-defined object in the form of a serializable string, such as a POJO (plain old Java object) or a JavaScript Object Notation (JSON) stream. The contents of the event object include all the data and metadata that your Lambda function needs to drive its logic. The contents and structure of the event object vary, depending on which event source created it. For example, an event that is created by API Gateway contains details that are related to the HTTPS request that was made by the API client, such as path, query string, and request body. However, an event that is created by Amazon includes details about the bucket and the new object.
- *Context object* – Generated by AWS and provides metadata about the execution. The context object allows your function code to interact with the Lambda execution environment. The contents and structure of the context object vary based on the language runtime that your Lambda function uses. However, at a minimum, the context object will contain:
    - *awsRequestId* – This property is used to track specific invocations of a Lambda function (important for error reporting or when contacting AWS Support).
    - *logStreamName* – The CloudWatch log stream that your log statements will be sent to.
    - *getRemainingTimeInMillis()* – This method returns the number of milliseconds that remain before the execution of your function times out.

Design a Lambda function: Example

```
def my_handler(event, context):
     message = 'Hello {} {}!'.format(event['first_name'],
                                     event['last_name'])
     return {
          'message' : message
     }
```

© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

This example is a Lambda function that is written in Python. It defines a handler function that is called *my_handler*. The function returns a message that contains data from the event it received as input.

Note that each language has its own requirements for how a function handler can be defined and referenced within the deployment package.

For more examples and details on how to create Lambda functions in your language of choice, see the AWS Lambda Developer Guide:
https://docs.aws.amazon.com/lambda/latest/dg/welcome.html

Design a Lambda function: Best practices

- Separate core business logic
- Write modular functions
- Treat functions as stateless
- Include only what you need
- Reuse execution context

© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

Here are some best practices to follow when you are designing your Lambda function:

- *Separate core business logic from the handler method* – This makes your code more portable and enables you to target unit tests at your code without worrying about the configuration of the function.
- *Write modular functions* – Create single-purpose functions.
- *Treat functions as stateless* – No information about state should be saved within the context of the function itself. Because your functions only exist when there is work to be done, it's important for serverless applications to treat each function as stateless.
- *Only include what you need* –
    - Minimize both the size and the dependencies of your deployment package. This can reduce the startup time for your function. For example, choose only the modules that you need—such as Amazon DynamoDB and Amazon S3 software development kit (SDK) modules, Lambda core libraries—and don't include an entire AWS SDK library in your deployment package.
    - Reduce the time it takes Lambda to unpack deployment packages that are authored in Java.
    - Minimize the complexity of your dependencies. Choose simpler Java dependency injection (IoC) frameworks. For example, choose Dagger or Guice instead of more complex frameworks like Spring Framework.
- *Reuse execution context to improve the performance of your function* – The execution context is a temporary runtime environment that initializes any external dependencies of your Lambda function code. Make sure any externalized configuration or dependencies that your code retrieves are stored and referenced locally after initial execution. Limit re-initializing variables and objects on every invocation. Keep alive and reuse connections—such as HTTP, database, etc.—that were established during a previous invocation.

Create a Lambda function: Best practices

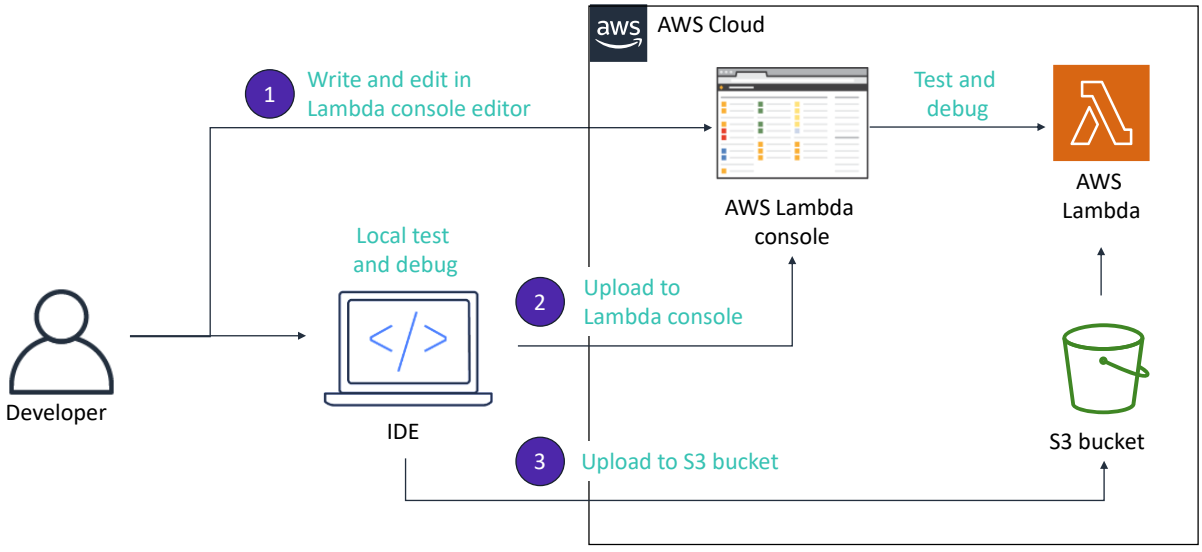
- Include logging statements
- Include results information
- Use environment variables
- Avoid recursive code

© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

Here are some best practices to follow when you are writing code:

- *Include logging statements* – Lambda functions can and should include logging statements that are written to CloudWatch.
- *Include results information* – Functions must give AWS Lambda information about the results of their execution.
- *Use environment variables* – Environment variables allow you to pass operational parameters and configuration settings to your function without making changes to the code itself. For example, if you are writing to an S3 bucket, configure the bucket name as an environment variable instead of hardcoding the bucket name. You can also use environment variables to store sensitive information that is required by the function.
- *Avoid using recursive code* – Avoid a situation where a function calls itself. This could continue to spawn new invocations that would make you lose control of your concurrency.

Upload Lambda deployment package

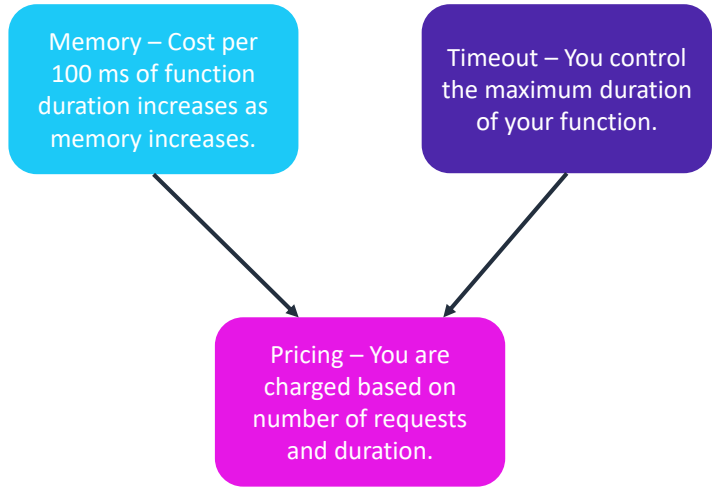


© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

You can author Lambda functions in three different ways:

- *Lambda console editor* – If your code does not require custom libraries (other than the AWS SDK), then you can edit your code inline through the AWS Management Console. The console will compress your code (with the relevant configuration information) into a deployment package that the Lambda service can run. This is the ideal deployment choice for simple scenarios.
- *Upload to the AWS Lambda console* – In an advanced scenario where your code requires custom libraries, you can create your deployment package in your IDE ,and then upload it through the AWS Lambda console.
- *Upload to Amazon S3* – You can also upload your deployment package to an S3 bucket and provide the S3 bucket, object key, and optional version for the object. AWS Lambda will load your code directly from Amazon S3.

Configure Lambda function



© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

Memory and timeout are configurations that determine how your Lambda function performs. These configurations affect your billing. With AWS Lambda, you are charged based on the number of requests for your functions (the total number of requests across all your functions) and the duration (the time it takes for your code to execute). The price depends on the amount of memory you allocate to your function.

- *Memory* – You specify the amount of memory you want to allocate to your Lambda function. Lambda then allocates CPU power that is proportional to the memory. Lambda is priced so that the cost per 100 ms of function duration increases as the memory configuration increases.
- *Timeout* – You can control the maximum duration of your function by using the timeout configuration. Using a timeout can prevent higher costs that come from long-running functions. You must find the right balance between not letting the function run too long and being able to finish under normal circumstances.

Follow these best practices:
- *Test the performance of your Lambda function* to make sure that you choose the optimum memory size configuration. You can view the memory usage for your function in Amazon CloudWatch Logs.
- *Load-test your Lambda function* to analyze how long your function runs and determine the best timeout value. This is important when your Lambda function makes network calls to resources that might not be able to handle the scaling of Lambda functions.

See the following resources for information about:
- AWS Lambda limits: https://docs.aws.amazon.com/lambda/latest/dg/limits.html
- AWS Lambda pricing: https://aws.amazon.com/lambda/pricing/

This week – Cloud native Computation

- Containers
  - ☐ Introduction to containers
  - ☐ Containers vs. hardware virtualization
  - ☐ Microservices: Use case for containers
  - ☐ Amazon container orchestration services
- Serverless computation (ACD Module 9)
  - ☐ Introduction to serverless computing with AWS Lambda
  - ☐ Execution models for invoking Lambda functions
  - ☐ AWS Lambda permissions
  - ☐ Overview of authoring and configuring Lambda functions
  - ☐ **Overview of deploying Lambda functions**

SWINBURNE UNIVERSITY OF TECHNOLOGY

72

Overview of deploying Lambda functions

© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

aws academy

Part 6: Overview of deploying Lambda functions

In this section, you will learn how to deploy Lambda functions.

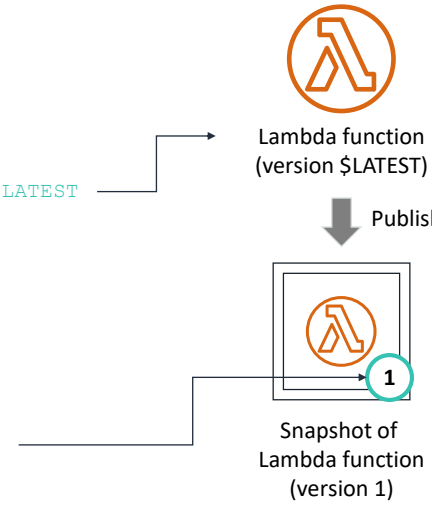
Versioning

- Immutable copies of Lambda function code and configuration
- Create: $LATEST version

```
arn:aws:lambda:aws-region:acct-id:function:helloworld:$LATEST
```

- Publish: Snapshot copy of $LATEST
- Each version has its own ARN with a new sequential version number

```
arn:aws:lambda:aws-region:acct-id:function:helloworld:1
```



© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

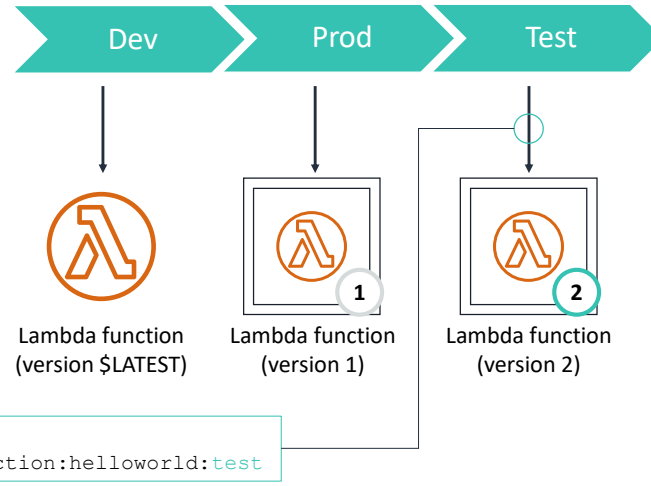
Versions are immutable copies of the code and configuration of your Lambda function. Versioning allows you to publish one or more versions of your Lambda function. As a result, you can work with different variations of your Lambda function in your development workflow, such as development, beta, and production.

When you create a Lambda function, there is only one version—the $LATEST version. You can refer to this function using its Amazon Resource Name (ARN). When you publish a new version, AWS Lambda makes a snapshot copy of the $LATEST version to create the new version. The new version has a unique ARN that includes a new sequential version number.

For more information on versioning, see the AWS Documentation: https://docs.aws.amazon.com/lambda/latest/dg/versioning-intro.html

Aliases

- Mutable pointers to versions
- Each alias has its own ARN



© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

You can create aliases for your Lambda function. Conceptually, an *alias* is a pointer to a specific Lambda function version. You can use the alias in the Amazon Resource Name (ARN) to reference the Lambda function version that is currently associated with that alias. Using an alias makes it easy to promote or roll back versions without changing any code. Aliases abstract the need to know which version is being referenced.

In this example, the test alias points to version 2 of the Lambda function.

For more information on aliases, see the AWS Documentation:
https://docs.aws.amazon.com/lambda/latest/dg/aliases-intro.html

Lambda layers

Centrally manage code and data that is shared across multiple functions.

- Reduce size of deployment packages
- Speed up deployments
- Limits:
  - Up to five layers
  - 250 MB

© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

When you build serverless applications, it is common to have code that is shared across Lambda functions. It can be custom code that is used by more than one function, or a standard library that you add to simplify the implementation of your business logic.

Previously, you had to package and deploy this shared code together with all the functions that used it. Now, you can configure your Lambda function to include additional code and content as layers. A *layer* is a .zip archive that contains libraries, a custom runtime, or other dependencies. With layers, you can use libraries in your function without needing to include them in your deployment package.

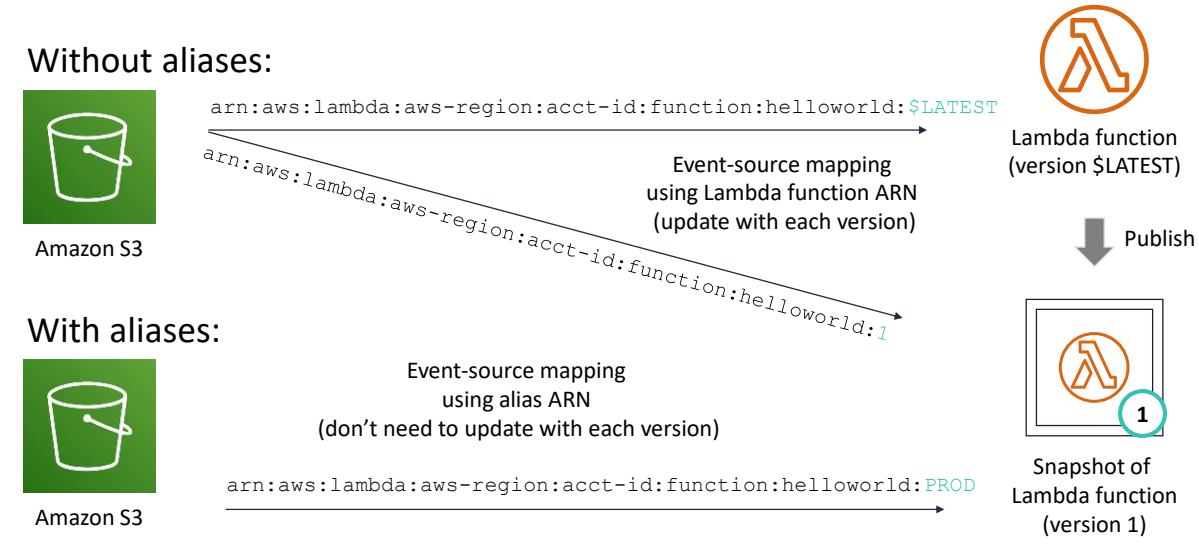
It is a best practice to have smaller deployment packages and to share common dependencies with layers. Layers let you keep your deployment package small, which makes development easier. You can avoid errors that can occur when you install and package dependencies with your function code. For Node.js, Python, and Ruby functions, you can develop your function code in the Lambda console as long as you keep your deployment package under 3 MB.

A function can use up to five layers at a time. The total unzipped size of the function and all layers can't exceed the unzipped deployment package size limit of 250 MB.

AWS published a public layer that includes NumPy and SciPy, which are scientific libraries for Python. This layer can help you with data processing and machine learning applications. To learn how to use this layer, read this AWS News Blog post: https://aws.amazon.com/blogs/aws/new-for-aws-lambda-use-any-programming-language-and-share-common-components/

To learn more about AWS Lambda layers, see the AWS Documentation: https://docs.aws.amazon.com/lambda/latest/dg/configuration-layers.html

Example of using versioning and aliases



© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

For example, suppose that Amazon S3 is the event source that invokes your Lambda function when new objects are created in a bucket. When Amazon S3 is your event source, you store the information for event-source mapping in the configuration for bucket notifications. In that configuration, you can identify the Lambda function ARN that Amazon S3 can invoke. However, in this case, you must update the notification configuration so that Amazon S3 invokes the correct version each time you publish a new version of your Lambda function.

Instead of specifying the function ARN, you can specify an alias ARN in the notification configuration (for example, you can specify the PROD alias ARN). As you promote new versions of your Lambda function into production, you only need to update the PROD alias to point to the latest stable version. You don't need to update the notification configuration in Amazon S3.

Module review

- Introduction to serverless computing with AWS Lambda
- Overview of how AWS Lambda works
- Execution models for invoking Lambda functions
- AWS Lambda permissions
- Overview of authoring and configuring Lambda functions
- Overview of deploying Lambda functions

- To finish this module, complete the **knowledge check**.

© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

This module addressed the following topics:

- Introduction to serverless computing with AWS Lambda
- Overview of how AWS Lambda works
- Execution models for invoking Lambda functions
- AWS Lambda permissions
- Overview of authoring and configuring Lambda functions
- Overview of deploying Lambda functions

To finish this module, please complete the corresponding knowledge check.