

AWS Academy Cloud Architecting

Module 13: Building Microservices and Serverless Architectures

© 2020, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



Welcome to Module 13: Building Microservices and Serverless Architectures.

Module overview



Sections

1. Architectural need
2. Introducing microservices
3. Building microservice applications with AWS container services
4. Introducing serverless architectures
5. Building serverless architectures with AWS Lambda
6. Extending serverless architectures with Amazon API Gateway
7. Orchestrating microservices with AWS Step Functions

Demonstrations

- Creating an AWS Lambda function
- Using AWS Lambda with Amazon S3

Labs

- (Optional) Guided Lab 1: Breaking a Monolithic Node.js Application into Microservices
- Guided Lab 2: Implementing a Serverless Architecture on AWS
- Challenge Lab: Implementing a Serverless Architecture for the Café



Knowledge check

© 2020, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

2

This module includes the following sections:

1. Architectural need
2. Introducing microservices
3. Building microservice applications with AWS container services
4. Introducing serverless architectures
5. Building serverless architectures with AWS Lambda
6. Extending serverless architectures with Amazon API Gateway
7. Orchestrating microservices with AWS Step Functions

This module also includes:

- Two AWS Lambda demonstrations
- An optional guided lab where you refactor a monolithic application into microservices
- A guided lab where you implement a serverless architecture on AWS with Amazon S3, AWS Lambda, Amazon DynamoDB, and Amazon SNS
- A challenge lab where you use AWS Lambda and Amazon Simple Notification Service (Amazon SNS) to generate and send a daily sales report for the café.

Finally, you will be asked to complete a knowledge check that will test your understanding of key concepts covered in this module.

At the end of this module, you should be able to:

- Indicate the characteristics of microservices
- Refactor a monolithic application into microservices and use Amazon ECS to deploy the containerized microservices
- Explain serverless architecture
- Implement a serverless architecture with AWS Lambda
- Describe a common architecture for Amazon API Gateway
- Describe the types of workflows that AWS Step Functions supports

At the end of this module, you should be able to:

- Indicate the characteristics of microservices
- Refactor a monolithic application into microservices and use Amazon ECS to deploy the containerized microservices
- Explain serverless architecture
- Implement a serverless architecture with AWS Lambda
- Describe a common architecture for Amazon API Gateway
- Describe the types of workflows that AWS Step Functions supports

Module 13: Building Microservices and Serverless Architectures

Section 1: Architectural need

© 2020, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



Introducing Section 1: Architectural need.

Café business requirement

The café wants to get daily reports via email about all the orders that were placed on the website. They want this information so they can anticipate demand and bake the correct number of desserts going forward (reducing waste). They also want to identify any patterns in their business (analytics).



Frank and Martha want to get daily reports via email about all the orders that were placed on the website. Frank wants to anticipate demand so he can bake the correct number of desserts going forward (reducing waste). Martha wants to identify any patterns in the café's business (analytics). Currently, Sofia has set up a cron job on the web server instance that sends these daily order report email messages to Frank and Martha. However, the cron job is resource-intensive and reduces web server performance.

Olivia advises Sofia and Nikhil that non-business-critical reporting tasks should be kept separate. Sofia and Nikhil want to further decouple the architecture and move the cron job into a managed, serverless environment that will scale well and reduce costs.

Module 13: Building Microservices and Serverless Architectures

Section 2: Introducing microservices

© 2020, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



Introducing Section 2: Introducing microservices.

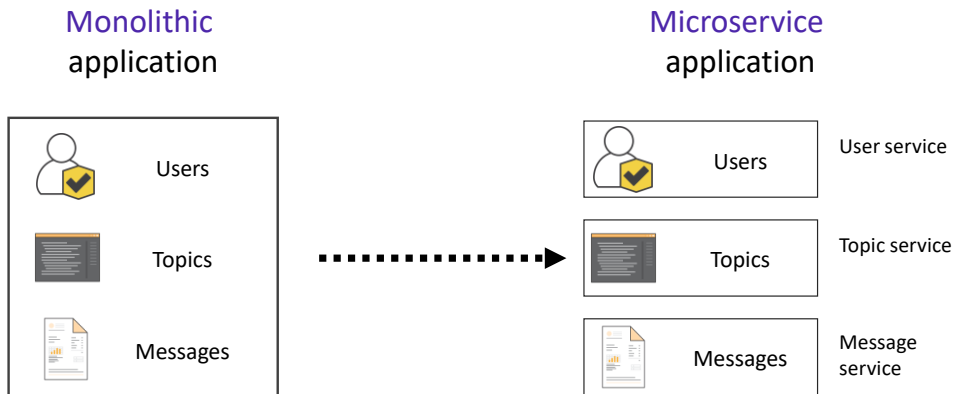
What are microservices?

Applications that are composed of **independent services**
that communicate over **well-defined APIs**

Microservices are an architectural and organizational approach to software development where applications are composed of independent services that communicate over well-defined application programming interfaces (APIs). This approach is designed to speed up deployment cycles.

The microservices approach fosters innovation and ownership, and improves the maintainability and scalability of software applications.

Monolithic versus microservice applications



To understand the benefits of microservices, consider first a monolithic application.

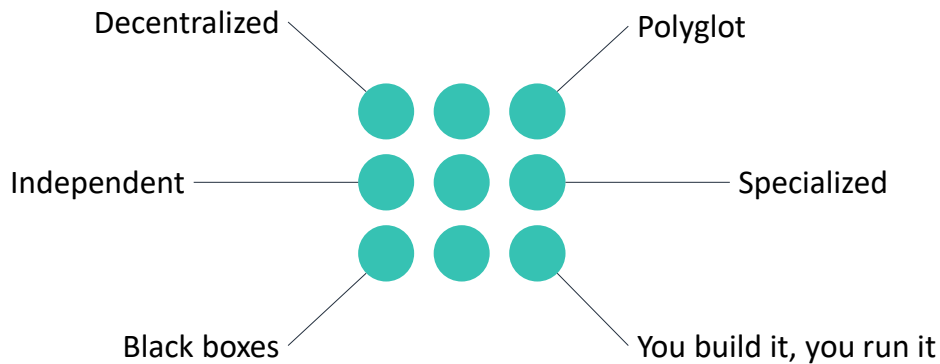
In the example on the left, the three processes (users, topics, and messages) of a monolithic forum application are tightly coupled. They run as a single service. If one process of the application experiences a spike in demand, the entire architecture must be scaled. Adding or improving features becomes more complex as the code base grows, which limits experimentation and makes it difficult to implement new ideas. The availability of monolithic applications is also at risk because many dependent and tightly coupled processes increase the impact of a single process failure.

Now, suppose that the same application runs in a microservice architecture. Each process of the application is built as an independent component that runs as a service. The services communicate 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 provides much quicker iteration, automation, and overall agility. Start fast, fail fast, and recover fast.

For an overview of microservices on AWS, see [What are Microservices?](#)

Characteristics of microservices



Microservices share some common characteristics:

- **Decentralized** – Microservice architectures are distributed systems with decentralized data management. They don't rely on a unifying schema in a central database. Each microservice has its own view about data models. Microservices are also decentralized in the way they are developed, deployed, managed, and operated.
- **Independent** – Each component service in a microservice architecture can be changed, upgraded, or replaced independently without affecting the function of other services. Services do not need to share any of their code or implementation with other services. Similarly, the teams responsible for different microservices can act independently from each other.
- **Specialized** – Each component service is designed for a set of capabilities and focuses on a specific domain. If the code for a particular component service reaches a certain level of complexity, then the service can be split into two or more services.
- **Polyglot** – Microservices don't follow a single approach. Teams have the freedom to choose the best tool for their specific problem. As a consequence, microservice architectures take a heterogeneous approach to operating systems, programming languages, data stores, and tools. This approach is called polyglot persistence and programming.

- Black boxes – Individual component services are designed as black boxes, which mean that the details of their complexity are hidden from other components. Any communication between services happens through well-defined APIs to prevent implicit and hidden dependencies.
- You build it, you run it – DevOps is a key organizational principle for microservices, where the team responsible for building a service is also responsible for operating and maintaining it in production.

Section 2 key takeaways



10

- Microservice applications are composed of independent services that communicate over well-defined APIs
- Microservices share the following characteristics –
 - Decentralized
 - Independent
 - Specialized
 - Polyglot
 - Black boxes
 - You build it, you run it

© 2020, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

Some key takeaways from this section of the module include:

- Microservice applications are composed of independent services that communicate over well-defined APIs
- Microservices share the following characteristics –
 - Decentralized: Microservices are decentralized in the way they are developed, deployed, managed, and operated
 - Independent: Each component service in a microservices architecture can be developed, deployed, operated, and scaled without affecting the function of other services
 - Specialized: Each component service is designed for a set of capabilities and focuses on solving a specific problem
 - Polyglot: Microservice architectures take a heterogeneous approach to operating systems, programming languages, data stores, and tools
 - Black boxes: The details of the complexity of microservice components are hidden from other components
 - You build it, you run it: DevOps is a key organizational principle for microservices

Module 13: Building Microservices and Serverless Architectures

Section 3: Building microservice applications with AWS container services

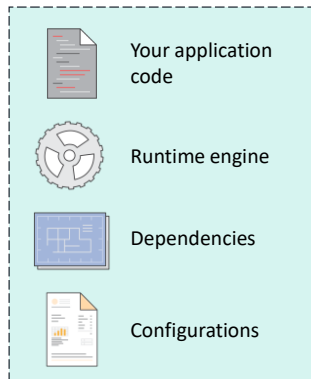
© 2020, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



Introducing Section 3: Building microservice applications with AWS container services.

What is a container?

Your container



When you build a microservice architecture, you can use containers for the processing power.

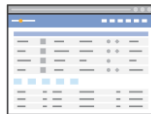
Containers are a method of operating system virtualization that enables you to run an application and its dependencies in resource-isolated processes. A container is a lightweight, standalone software package. It contains everything that a software application needs to run, such as the application code, runtime engine, system tools, system libraries, and configurations.

A problem that containers solve

Getting software to **run reliably in different work environments**



Developer's
workstation



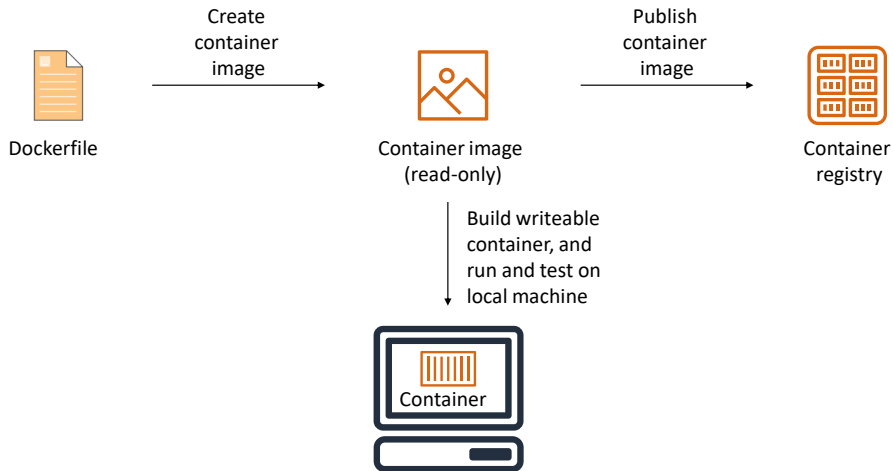
Production
environment



Test
environment

Containers can help ensure that applications deploy quickly, reliably, and consistently, regardless of deployment environment. Containers also give you more granular control over resources, which improves the efficiency of your infrastructure.

Container terminology



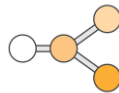
A container is created from a read-only template that is called an *image*. Images are typically built from a Dockerfile, which is a plaintext file that specifies all the components that are included in the container. You can create images from scratch, or you can use images that others created and published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

A container image is the snapshot of the file system that is available to the container. For example, you might have the Debian operating system as a container image. When you run this container, a Debian operating system is available to it. You can also package all your code dependencies in the container image and use it as your code artifact.

Container images are stored in a *registry*. You can download the images from the registry and run them on your cluster. Registries can exist in or outside your AWS infrastructure.



Orchestrates when containers run



Maintains and scales the fleet of instances that run your containers



Removes the complexity of standing up the infrastructure

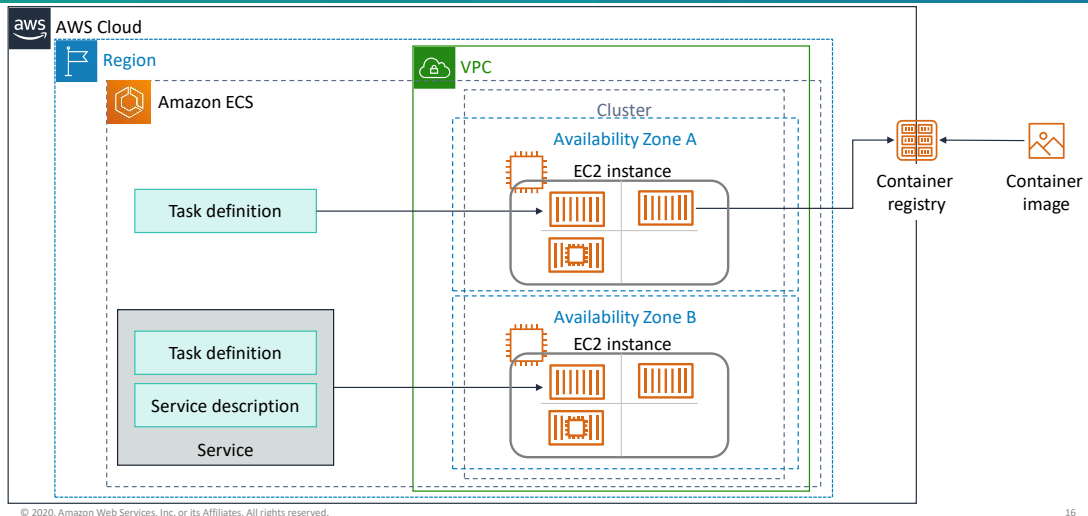
You can run your containers on Amazon Elastic Container Service (Amazon ECS). Amazon ECS is a highly scalable, high-performance, container-management service. It supports Docker containers and enables you to easily run applications on a managed cluster of Amazon Elastic Compute Cloud (Amazon EC2) instances.

Amazon ECS is a scalable cluster service for hosting containers that:

- Can scale up to thousands of Docker containers in seconds
- Monitors container deployment
- Manages the state of the cluster that runs the containers
- Schedules containers by using a built-in scheduler or third-party scheduler (Apache Mesos, Blox)
- Is extensible by using APIs
- Can be launched with either AWS Fargate or Amazon EC2 [launch types](#)

You can run ECS clusters at scale by mixing Spot Instances with On-Demand Instances and Reserved Instances.

Amazon ECS orchestrates containers



© 2020, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

16

Amazon ECS is a regional service that simplifies running application containers in a highly available manner across multiple Availability Zones within a Region. You can create ECS clusters in a new or existing virtual private cloud (VPC). A *cluster* is a logical grouping of resources.

After a cluster is up and running, you can define task definitions and services that specify which Docker container images to run across your clusters.

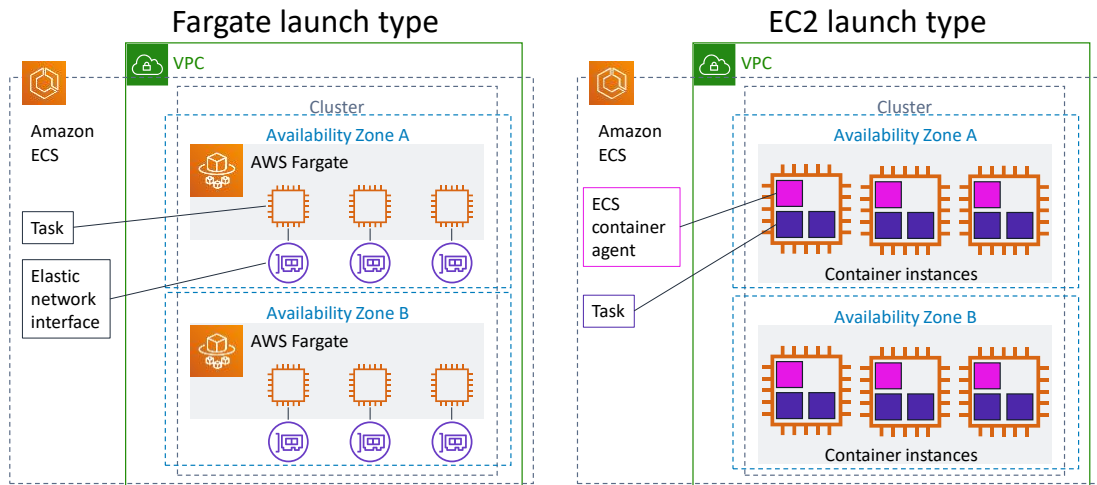
A *task definition* is a text file in JavaScript Object Notation (JSON) format. It describes one or more containers, up to a maximum of 10, that form your application. You can think of it as a blueprint for your application. Task definitions specify parameters for your application—for example, which containers and launch type to use. Other parameters include which ports should be opened for your application and what data volumes should be used with the containers in the task.

A *service* enables you to specify how many copies of your task definition to run and maintain in a cluster. You can optionally use an Elastic Load Balancing load balancer to distribute incoming traffic to containers in your service. Amazon ECS maintains that number of tasks and coordinates task scheduling with the load balancer.

After you create a task definition for your application, you can specify the number of tasks that will run on your cluster. A *task* is the instantiation of a task definition within a cluster. When you use Amazon ECS to run tasks, you place them in a cluster.

Amazon ECS downloads your container images from a registry that you specify, and runs those images within your cluster.

Amazon ECS launch types

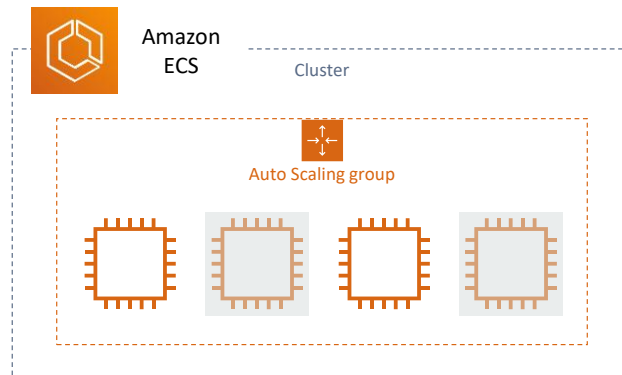


Amazon ECS offers two launch types for hosting your containerized applications.

You can use the Fargate launch type to host your cluster on a serverless infrastructure that Amazon ECS manages. You only need to package your application in containers, specify the CPU and memory requirements, define networking and AWS Identity and Access Management (IAM) policies, and launch the application.

Alternatively, if you want more control, you can use the EC2 launch type to host your tasks on a cluster of EC2 container instances that *you* manage. A *container instance* is an EC2 instance that is running the *Amazon ECS container agent*. You can use Amazon ECS to schedule the placement of containers across your cluster based on your resource needs, isolation policies, and availability requirements. For information about different scheduling options, see [Scheduling Amazon ECS Tasks](#). Amazon ECS keeps track of all the CPU, memory, and other resources in your cluster. It also finds the best server for a container to run on based on your specified resource requirements.

For more information about the Fargate and EC2 launch types, see [Amazon ECS Launch Types](#).



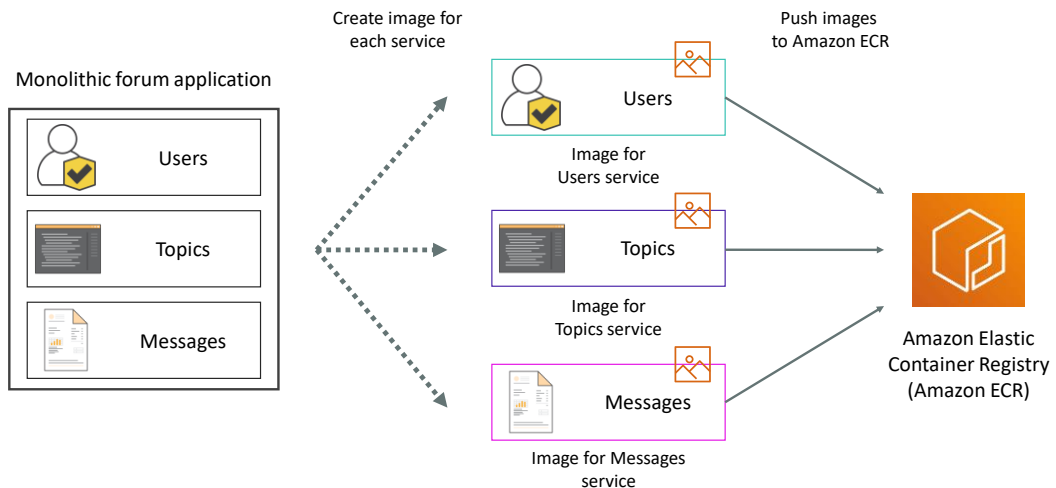
You can [create an Auto Scaling group for an Amazon ECS cluster](#). The Auto Scaling group contains container instances that you can scale out (and in) by using Amazon CloudWatch alarms. If you configure your Auto Scaling group to remove container instances, any tasks that are running on the removed container instances are stopped. If your tasks are running as part of a service, Amazon ECS restarts those tasks on another instance if the required resources are available. Examples of such required resources include CPU, memory, ports. However, tasks that were started manually are not restarted automatically.

You can also take advantage of [Amazon ECS cluster auto scaling](#), which gives you more control over how you scale tasks in a cluster. It increases the speed and reliability of cluster scale-out. It gives you control over the amount of spare capacity that is maintained in your cluster, and automatically manages instance termination on scale-in.

With cluster auto scaling, you can configure Amazon ECS to scale your Auto Scaling group in and out automatically. Cluster auto scaling relies on capacity providers, which link your ECS cluster to the Auto Scaling groups that you want to use. Each Auto Scaling group is associated with a capacity provider, and each capacity provider has only one Auto Scaling group. However, many capacity providers can be associated with one ECS cluster. To scale the entire cluster automatically, each capacity provider manages the scaling of its associated Auto Scaling group.

For more information about cluster auto scaling, see the [Amazon ECS Cluster Auto Scaling AWS News Blog post](#).

Decomposing monoliths – Step 1: Create container images



Again, consider the monolithic forum application that you saw earlier where the entire application runs as a single service. To rearchitect this application by using a microservice architecture, you can run each application process as a separate service within its own container. With a microservice architecture, the services can scale and be updated independently of the others.

To deploy the monolithic application as a microservice application, first build and tag an image for each service. Then, register the images with Amazon Elastic Container Registry (Amazon ECR).

Decomposing monoliths – Step 2: Create service task definition and target groups

Service Task Definition

- Launch type = [EC2 or Fargate]
- Name = [service-name]
- Image = [service ECR repo URL]:version
- CPU = [256]
- Memory = [256]
- Container port = [3000]
- Host port = [0]

Service Target Group

- Name = [service-name]
- Protocol = [HTTP]
- Port = [80]
- VPC = [vpc-name]



Amazon
ECS

Cluster



Target groups

EC2 instance with
service containers



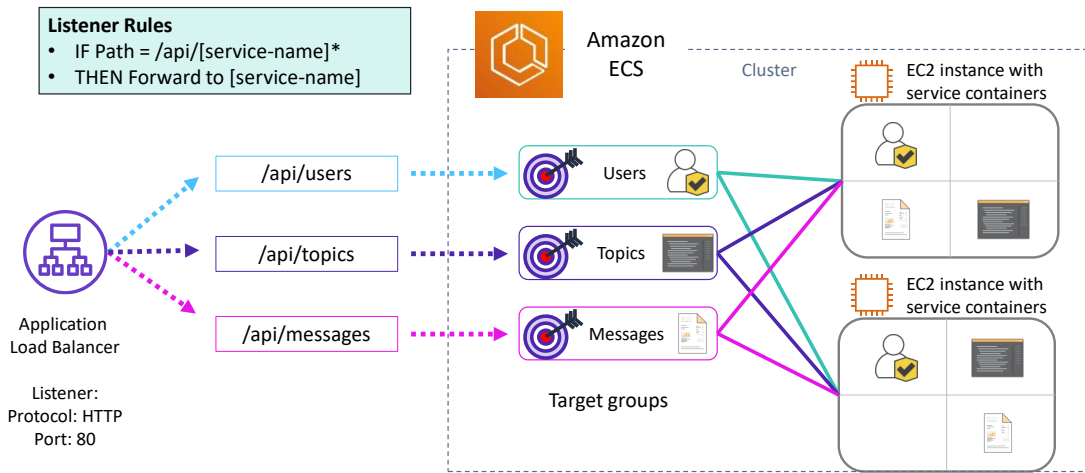
EC2 instance with
service containers



Next, choose a launch type and create a new service for each piece of the original monolithic application. Amazon ECS deploys each service into its own container across an ECS cluster.

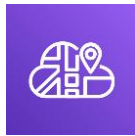
Then, create a target group for each service. The target group tracks the instances and ports of each container that is running for that service.

Decomposing monoliths – Step 3: Connect load balancer to services



Finally, create an Application Load Balancer and configure listener rules to connect to the services. The listener checks for incoming connection requests to your load balancer and uses the rules to route traffic appropriately. In the example, the listener for the Application Load Balancer listens for HTTP service requests on Port 80 and routes them to the appropriate service.

Tools for building highly available microservice architectures



AWS Cloud Map

- Is a fully managed discovery service for cloud resources
- Can be used to define custom names for application resources
- Maintains updated location of dynamically changing resources, which increases application availability



AWS App Mesh

- Captures metrics, logs, and traces from all your microservices
- Enables you to export this data to Amazon CloudWatch, AWS X-Ray, and compatible AWS Partner Network (APN) Partner and community tools
- Enables you to control traffic flows between microservices to help ensure that services are highly available

© 2020, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

22

AWS Cloud Map and AWS App Mesh are two tools that can help you build highly available microservice architectures.

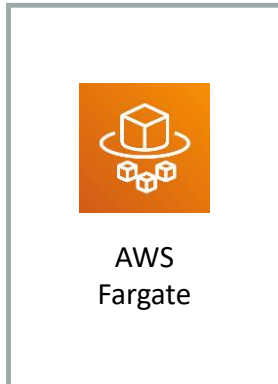
[AWS Cloud Map](#) is a fully managed discovery service for cloud resources. You can use it to define custom names for your application resources (such as databases, queues, microservices, and other cloud resources). AWS Cloud Map maintains the updated location of these dynamically changing resources. This location maintenance increases the availability of your application because your web service always discovers the most up-to-date locations of its resources. You can add and register any resource with minimal manual intervention of mappings. AWS Cloud Map assists with service discovery, continuous integration, and health monitoring of your microservices and applications.

For more information about AWS Cloud Map, read this [AWS Open Source Blog post](#). To learn more about how you can use AWS Cloud Map to enable your containerized services to discover and connect with each other, read [AWS Fargate, Amazon EKS, and Amazon ECS now integrate with AWS Cloud Map](#).

When you create your task definitions, you can enable App Mesh integration. [AWS App Mesh](#) captures metrics, logs, and traces from all of your microservices. You can export this data to Amazon CloudWatch, AWS X-Ray, and compatible AWS Partner Network (APN) Partner and community tools for monitoring and tracing. AWS App Mesh also enables you to control how traffic flows between your microservices to make sure that every service is highly available during deployments, after failures, and as your application scales.

App Mesh enables you to configure microservices to connect directly to each other via a proxy instead of requiring code within the application or by using a load balancer. App Mesh uses Envoy, an open source service-mesh proxy, which is deployed alongside your microservice containers.

For more information about AWS Cloud Map and AWS App Mesh, see this [AWS YouTube video](#).



- Is a **fully managed** container service
- Works with **Amazon Elastic Container Service** (Amazon ECS) and **Amazon Elastic Kubernetes Service** (Amazon EKS)
- Provisions, manages, and scales your container clusters
- Manages runtime environment
- Provides automatic scaling

In this section, you have learned that Amazon ECS offers two launch types: EC2 and Fargate.

[AWS Fargate](#) is a fully managed container service that works with both Amazon ECS and Amazon Elastic Kubernetes Service (Amazon EKS). It enables you to run containers without needing to manage servers or clusters. With AWS Fargate, you no longer need to provision, configure, and scale clusters of virtual machines to run containers. As a result, you don't need to choose server types, decide when to scale your clusters, or optimize cluster packing. AWS Fargate reduces the need for you to interact with or think about servers or clusters. Fargate enables you to focus on designing and building your applications instead of managing the infrastructure that runs them.

Section 3 key takeaways



24



- **Amazon ECS** is a highly scalable, high-performance container management service. It supports Docker containers and enables you to easily run applications on a managed cluster of Amazon EC2 instances.
- **Cluster auto scaling** gives you more control over how you scale tasks in a cluster.
- **AWS Cloud Map** enables you to define custom names for your application resources. It maintains the updated location of these dynamically changing resources.
- **AWS App Mesh** is a service mesh that provides application-level networking. It enables your services to communicate easily with each other across multiple types of compute infrastructure.
- **AWS Fargate** is a fully managed container service that enables you to run containers without needing to manage servers or clusters.

© 2020, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

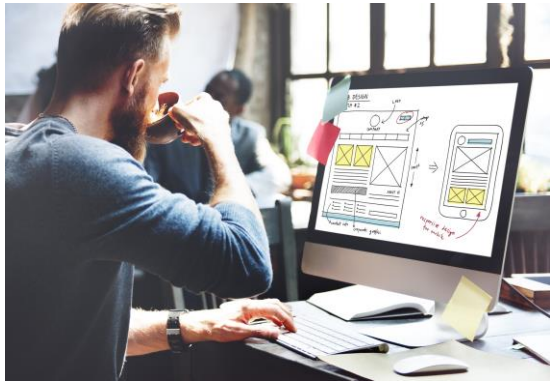
Some key takeaways from this section of the module include:

- Amazon ECS is a highly scalable, high-performance container management service. It supports Docker containers and enables you to easily run applications on a managed cluster of Amazon EC2 instances.
- Cluster auto scaling gives you more control over how you scale tasks within a cluster.
- AWS Cloud Map enables you to define custom names for your application resources. It maintains the updated location of these dynamically changing resources.
- AWS App Mesh is a service mesh that provides application-level networking to make it easy for your services to communicate with each other across multiple types of compute infrastructure.
- AWS App Mesh is a service mesh that provides application-level networking. It enables your services to communicate easily with each other across multiple types of compute infrastructure.
- AWS Fargate is a fully managed container service that enables you to run containers without needing to manage servers or clusters.

Module 13 – Guided Lab 1: Breaking a Monolithic Node.js Application into Microservices

(Optional lab)

25



© 2020, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

You might choose to complete Module 13 – Guided Lab 1: Breaking a Monolithic Node.js Application into Microservices. This lab is optional.

Guided lab 1: Tasks

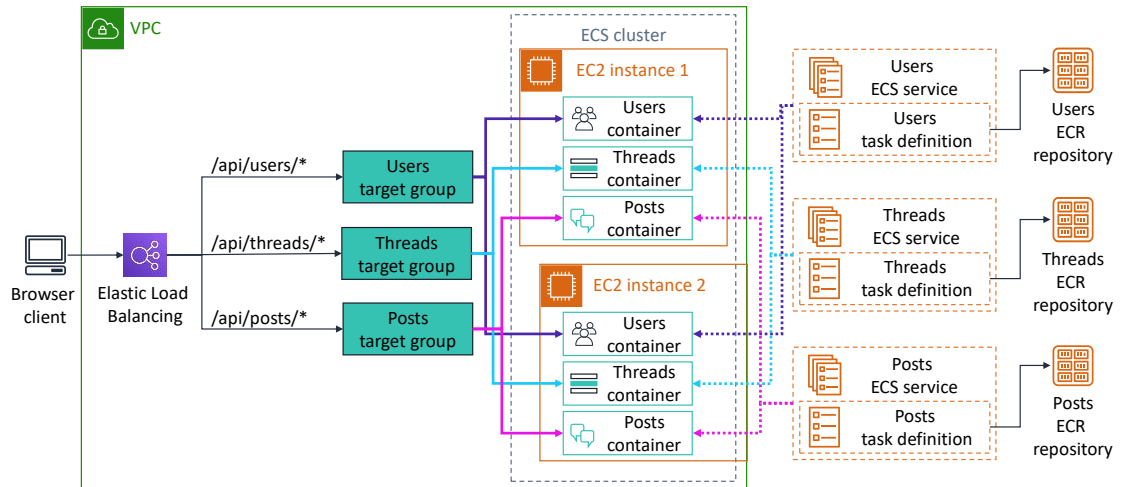


1. Prepare the AWS Cloud9 development environment
2. Run a monolithic application on a basic Node.js server
3. Containerize the monolith for Amazon ECS
4. Deploy the monolith to Amazon ECS
5. Refactor the monolith into containerized microservices

In this guided lab, you will complete the following tasks:

1. Prepare the AWS Cloud9 development environment
2. Run a monolithic application on a basic Node.js server
3. Containerize the monolith for Amazon ECS
4. Deploy the monolith to Amazon ECS
5. Refactor the monolith into containerized microservices

Guided lab 1: Final product



The diagram summarizes what you will have built after you complete the lab.



~ 3 hours



Begin Module 13 – Guided Lab 1: Breaking a Monolithic Node.js Application into Microservices

It is now time to start the optional guided lab.

Guided lab 1 debrief: Key takeaways



Your educator might choose to lead a conversation about the key takeaways from this guided lab after you have completed it.

Module 13: Building Microservices and Serverless Architectures

Section 4: Introducing serverless architectures

© 2020, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



Introducing Section 4: Introducing serverless architectures.

What does serverless mean?



A way for you to build and run applications and services
without thinking about servers

So far, you have learned that you can use Amazon ECS to build your microservice applications by using containers. Amazon ECS is a container orchestration service where you manage your application code, data source integrations, security configuration, updates, network configuration, firewall, and management tasks. You also learned that you can use the Fargate launch type to host your cluster on a *serverless* infrastructure that Amazon ECS manages.

But what does serverless mean?

Serverless is the native architecture of the cloud that enables you to shift more operational responsibilities to AWS, which can increase your agility and innovation. Serverless enables you to build and run applications and services without thinking about servers. Your application still runs on servers. However, AWS does all the server management tasks, such as server or cluster provisioning, patching, operating system maintenance, and capacity provisioning.

Tenets of serverless architectures



No infrastructure provisioning,
no management



Automatic scaling

Pay for value



Highly available and secure



The tenets that define serverless as an operational model include:

- No infrastructure to provision or manage (no servers to provision, operate, or patch)
- Automatically scales by unit of consumption (scales by unit of work or consumption rather than by server unit)
- *Pay-for-value* pricing model (you pay only for the duration that a resource runs, rather than by server unit)
- Built-in availability and fault tolerance (no need to architect for availability because it is built into the service)

For more information about what serverless is, see [this AWS website](#).

Benefits of serverless



Lower total cost
of ownership



Focus on your application,
not configuration



Build **microservice**
applications

Serverless enables you to build [modern applications](#) with increased agility and lower total cost of ownership (TCO). By using a serverless architecture, you can focus on your core product. You don't need to worry about managing and operating servers or runtimes, either in the cloud or on premises. This reduced overhead enables you to reclaim time and energy, which you can spend on developing products that scale and are reliable. Finally, serverless architectures enable you to build microservice applications.

AWS serverless offerings

Compute



AWS Lambda and
Lambda@Edge



AWS Fargate

Storage



Amazon S3



Amazon EFS

Data Stores



Amazon
DynamoDB



Amazon
Aurora



Amazon
RDS Proxy

API Proxy



Amazon
API Gateway

Application integration



Amazon SNS



AWS AppSync



Amazon SQS



Amazon
EventBridge

Orchestration



AWS Step
Functions

Analytics



Amazon Kinesis



Amazon Athena

AWS has many offerings that you can use to build serverless architectures on AWS. So far in this course, you have already learned about several of them.

The rest of this module focuses on how you can use AWS Lambda, Amazon API Gateway, and AWS Step Functions to build serverless architectures.

Section 4 key takeaways



35

- **Serverless computing** enables you to build and run applications and services without provisioning or managing servers
- Serverless architectures offer the following **benefits** –
 - Lower total cost of ownership (TCO)
 - You can focus on your application
 - You can use them to build microservice applications

© 2020, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

Some key takeaways from this section of the module include:

- Serverless computing enables you to build and run applications and services without provisioning or managing servers
- Serverless architectures offer the following benefits –
 - Lower TCO
 - You can focus on your application
 - You can use them to build microservice applications

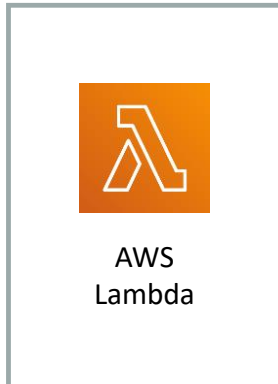
Module 13: Building Microservices and Serverless Architectures

Section 5: Building serverless architectures with AWS Lambda

© 2020, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



Introducing Section 5: Building serverless architectures with AWS Lambda.



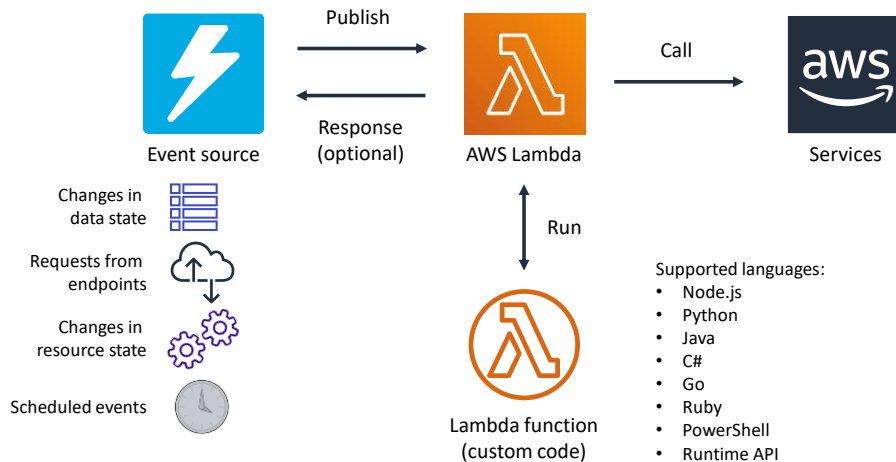
- Is a **fully managed** compute service
- Runs your code on a schedule or in **response to events** (for example, changes to an Amazon S3 bucket or an Amazon DynamoDB table)
- Supports Java, Go, PowerShell, Node.js, C#, Python, Ruby, and Runtime API
- Can run at edge locations closer to your users

[AWS Lambda](#) is a fully managed compute service that runs your code in response to events and automatically manages the underlying compute resources for you. Lambda runs your code on a high-availability compute infrastructure and performs all administration of the compute resources, including server and operating system maintenance, capacity provisioning, automatic scaling, code monitoring, and logging.

AWS Lambda natively supports Java, Go, PowerShell, Node.js, C#, Python, and Ruby code, and provides a Runtime API that enables you to use any additional programming languages to author your functions.

[Lambda@Edge](#) is a feature of Amazon CloudFront that enables you to run code closer to users of your application, which improves performance and reduces latency. Lambda@Edge runs your code in response to events that are generated by the Amazon CloudFront content delivery network (CDN). Lambda@Edge enables you to run Node.js and Python Lambda functions to customize content that Amazon CloudFront delivers. For information about how to add HTTP security response headers, read this [AWS Networking & Content Delivery Blog post](#).

How AWS Lambda works

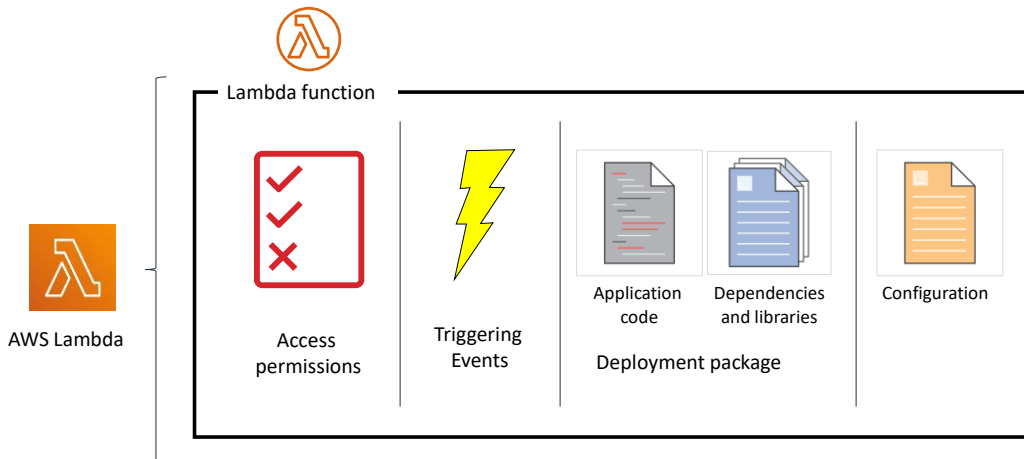


AWS Lambda integrates with other AWS services to invoke Lambda functions. A *Lambda function* is custom code that you write in one of the languages that Lambda supports. You can configure triggers to invoke a function in response to resource lifecycle events, respond to incoming HTTP requests, consume events from a queue, or run on a schedule.

An *event source* is the entity that publishes the event to Lambda. Your Lambda function processes the event, and Lambda runs your Lambda function on your behalf.

Lambda functions are *stateless*, which means that they have no affinity to the underlying infrastructure. Lambda can rapidly launch as many copies of the function as needed to scale to the rate of incoming events.

Lambda functions



When you create a Lambda 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 needed to run your code. Finally, you configure runtime parameters such as memory, time out, and concurrency. When your function is invoked, Lambda will run an environment based on the runtime and configuration options that you selected.

For more information about how AWS Lambda runs, see the [AWS Documentation](#).

Anatomy of a Lambda function



Handler()

Function to be run upon invocation

Event object

Data sent during Lambda function invocation

Context object

Methods available to interact with runtime information (request ID, log group, more)

```
import json

def lambda_handler(event, context):
    # TODO implement
    return {
        'statusCode': 200,
        'body': json.dumps('Hello world')
    }
```

When a Lambda function is invoked, the code begins running at the handler. The *handler* is a specific code method or function that you create and include in your package. You specify the handler when you create a Lambda function. Each supported language has its own requirements for how a function handler can be defined and referenced within the package. After the handler is successfully invoked inside your Lambda function, the runtime environment belongs to the code you wrote.

The handler always takes two objects: the event object and the context object.

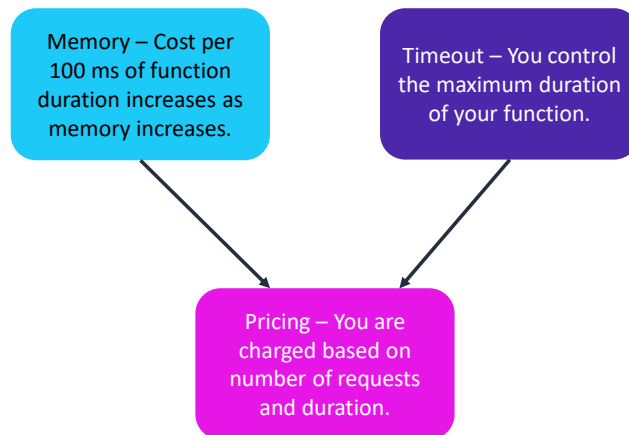
The *event object* provides information about the event that triggered the Lambda function. This event might be a pre-defined object that an AWS service generates, or a custom user-defined object in the form of a serializable string. An example of such a string might be a plain old Java object (POJO) or a 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.

The *context object* is generated by AWS and provides metadata about the runtime environment. The context object enables your function code to interact with the Lambda runtime 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 contains:

- *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 running of your function times out



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 run). 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. For example, say that you have a Lambda function with 256 MB of memory, and it runs for 110 milliseconds. This function will cost twice as much as a Lambda function with 128 MB of memory that runs for the same time.
- **Timeout** – You can control the maximum duration of your function by using the timeout configuration. You can set the timeout value for a function to any value up to 15 minutes. When the specified timeout is reached, AWS Lambda stops the running of your Lambda function. 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](#)
- [AWS Lambda pricing](#)

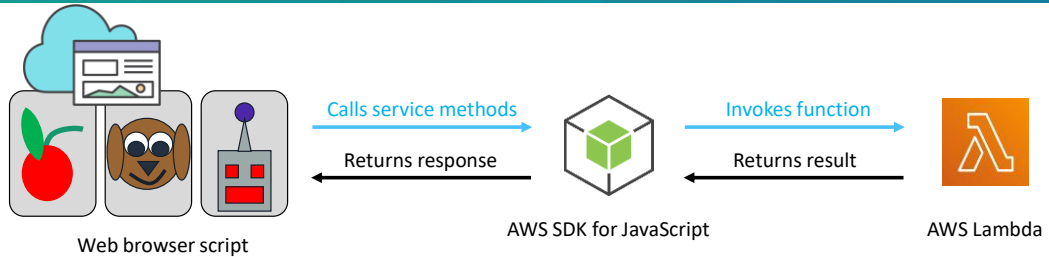
Demonstration: Creating an AWS Lambda function

42



Now, the educator might choose to demonstrate how to create an AWS Lambda function.

AWS Lambda example: Simulated slot machine browser game

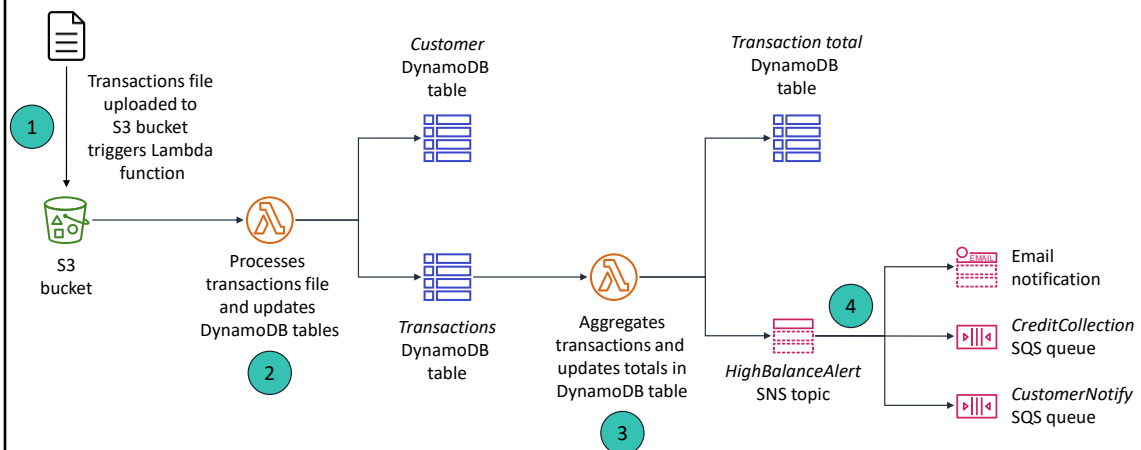


```
lambda.invoke(pullParams, function(error, data) {
  if (error) {
    prompt(error);
  } else {
    pullResults = JSON.parse(data.Payload);
  }
});
```

```
{
  iswinner: false,
  leftwheelImage : {S : 'cherry.png'},
  midwheelImage : {S : 'puppy.png'},
  rightwheelImage : {S : 'robot.png'}
}
```

You can create Lambda functions to perform various tasks. This example uses a browser-based game that simulates a slot machine. The game invokes a Lambda function that generates the random results of each slot pull. The function returns those results as the file names of images that are used to display the result. The images are stored in an Amazon S3 bucket that is configured to function as a static web host for the HTML, CSS, and other assets that are needed to present the application experience.

Event-based Lambda function example: Order processing



This example shows how Lambda can be used in a solution for order processing.

In this architecture:

1. A customer uploads a transactions file to an S3 bucket, which triggers the running of a Lambda function.
2. A Lambda function processes the transactions file and updates the *Customer* and *Transactions* DynamoDB tables.
3. Changes to the *Transactions* DynamoDB table trigger a second Lambda function to aggregate the transactions and update the totals in the *Transaction total* DynamoDB table. It also pushes a message to the *HighBalanceAlert* SNS topic.
4. The *HighBalanceAlert* SNS topic sends an email notification to the customer, and updates the *CreditCollection* and *CustomerNotify* SQS queues for payment processing.



- Enable functions to share code easily – You can upload a layer one time and reference it in any function
- Promote separation of responsibilities – Developers can iterate faster on writing business logic
- Enable you to keep your deployment packages small
- Limits –
 - Up to five layers
 - 250 MB

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

Previously, you packaged and deployed 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 Lambda layers, functions can share code. Developers use layers to upload code one time and reuse it multiple times. With layers, you can use libraries in your function without needing to include them in your deployment package.

Sharing code this way can help promote the separation of responsibilities. One person can be responsible for managing the core library. Another person can be responsible for using and building on top of the library code to build application logic.

Layers enable you to keep your deployment package small, which makes development easier.

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.

For more information about layers, see [AWS Lambda Layers](#).

Demonstration: Using AWS Lambda with Amazon S3

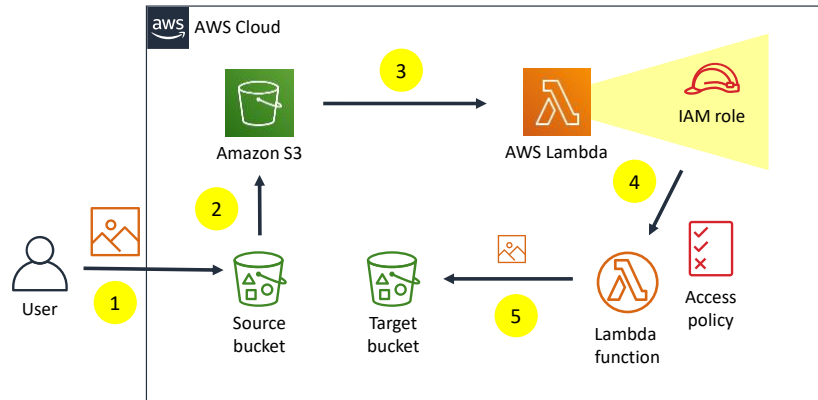
46



Now, the educator might choose to demonstrate how to configure an Amazon S3 event to trigger a Lambda function.

Demonstration diagram

1. User uploads image to source S3 bucket.
2. Amazon S3 detects the object-created event.
3. Amazon S3 publishes the event to Lambda.
4. Lambda runs the Lambda function.
5. The Lambda function resizes the original image and saves the thumbnail to the target S3 bucket.



This demonstration covers the following steps:

1. A user uploads an image to a source S3 bucket.
2. Amazon S3 detects the object-created event.
3. Amazon S3 publishes the event to Lambda by invoking the Lambda function and passing event data as a function parameter.
4. Lambda assumes an IAM role that allows it to run the function, and then runs the Lambda function.
5. From the event data it receives, the Lambda function knows the source bucket and the object key name. The Lambda function reads the object, creates a thumbnail of the original image, and saves the thumbnail to the target S3 bucket.

```

import boto3
import os
import sys
import uuid
from PIL import Image
import PIL.Image

s3_client = boto3.client('s3')

def resize_image(image_path, resized_path):
    with Image.open(image_path) as image:
        image.thumbnail((128, 128))
        image.save(resized_path)

def handler(event, context):
    for record in event['Records']:
        bucket = record['s3']['bucket']['name']
        key = record['s3']['object']['key']
        download_path = '/tmp/{}'.format(uuid.uuid4(), key)
        upload_path = '/tmp/resized-{}'.format(key)

        s3_client.download_file(bucket, key, download_path)
        resize_image(download_path, upload_path)
        s3_client.upload_file(upload_path, '{}-resized'.format(bucket), key)

```

Receives S3 event and downloads the image to local storage

Resizes the image

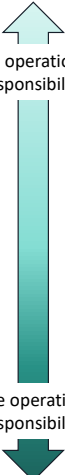
Uploads resized image to the *-resized* bucket

The Lambda function code performs the following steps:

- Receives an S3 event, which contains the name of the incoming object (Bucket, Key)
- Downloads the image to local storage
- Resizes the image using the *Pillow* library
- Uploads the resized image to the *-resized* S3 bucket

Comparison of operational responsibility for container and serverless architectures



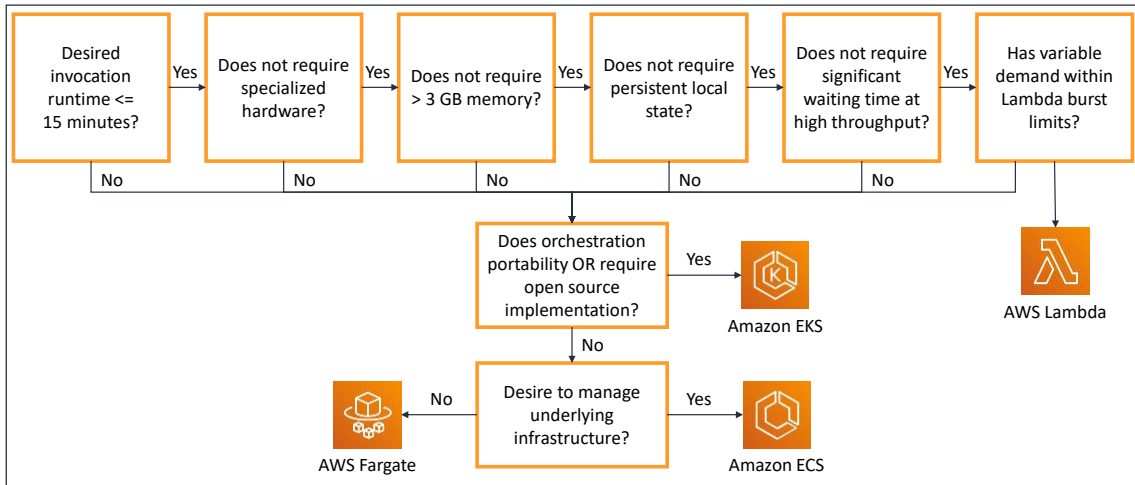
		AWS Manages	Customer Manages
 Less operational responsibility	AWS Lambda Serverless Functions	<ul style="list-style-type: none">• Data source integrations• Physical hardware, software, networking, and facilities• Provisioning	<ul style="list-style-type: none">• Application code
	AWS Fargate Serverless Containers	<ul style="list-style-type: none">• Container orchestration and provisioning• Cluster scaling• Physical hardware, host OS/kernel, networking, and facilities	<ul style="list-style-type: none">• Application code• Data source integrations• Security configuration and updates, network configuration, and management tasks
	Amazon ECS and Amazon EKS Container Management as a Service	<ul style="list-style-type: none">• Container orchestration control plane• Physical hardware software, networking, and facilities	<ul style="list-style-type: none">• Application code• Data source integrations• Work clusters• Security configuration and updates, network configuration, firewall, and management tasks
More operational responsibility			

© 2020, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

49

Which compute service you use to build your application depends on the level of control that you want. You share a spectrum of operational responsibility with AWS over your compute options for container and serverless architectures.

Choosing a compute platform: Containers versus AWS Lambda



This slide displays a rough guideline for selecting your compute platform. The recommendation is based on [AWS Lambda limits](#) (memory limitation, time limitation).

Section 5 key takeaways



51

- **Lambda** is a serverless compute service that provides built-in fault tolerance and automatic scaling
- A **Lambda function** is custom code that you write that processes events
- A Lambda function is invoked by a **handler**, which takes an **event object** and **context object** as parameters
- An **event source** is an AWS service or developer-created application that triggers a Lambda function to run
- **Lambda layers** enable functions to share code and keep deployment packages small

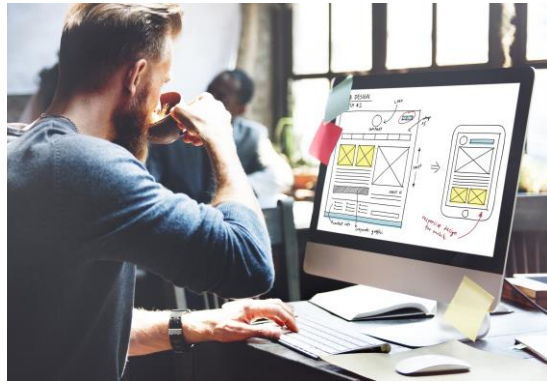
© 2020, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

Some key takeaways from this section of the module include:

- Lambda is a serverless compute service that provides built-in fault tolerance and automatic scaling.
- A Lambda function is custom code that you write that processes events.
- A Lambda function is invoked by a handler, which takes an event object and context object as parameters.
- An event source is an AWS service or developer-created application that triggers a Lambda function to run.
- Lambda layers enable functions to share code and keep deployment packages small.

Module 13 – Guided Lab 2: Implementing a Serverless Architecture on AWS

52



© 2020, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

You will now complete Module 13 – Guided Lab 2: Implementing a Serverless Architecture on AWS.

Guided lab 2: Tasks

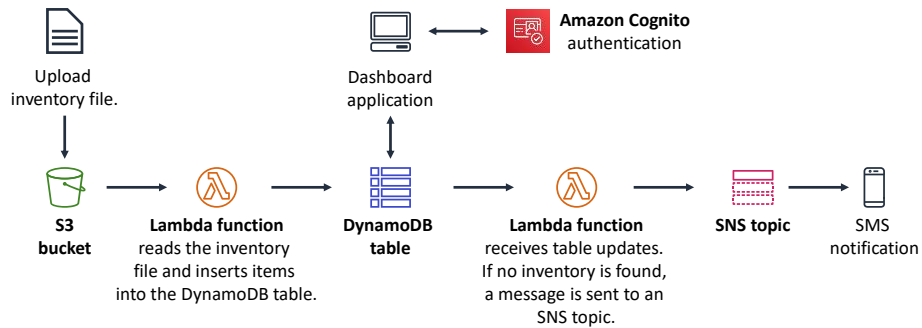


1. Create a Lambda function to load data
2. Configure an Amazon S3 event
3. Test the loading process
4. Configure notifications
5. Create a Lambda function to send notifications
6. Test the system

In this guided lab, you will complete the following tasks:

1. Create a Lambda function to load data
2. Configure an Amazon S3 event
3. Test the loading process
4. Configure notifications
5. Create a Lambda function to send notifications
6. Test the system

Guided lab 2: Final product



The diagram summarizes what you will have built after you complete the lab.



~ 40 minutes



Begin Module 13 – Guided Lab 2: Implementing a Serverless Architecture on AWS

It is now time to start the guided lab.

Guided lab 2 debrief: Key takeaways



Your educator might choose to lead a conversation about the key takeaways from this guided lab after you have completed it.

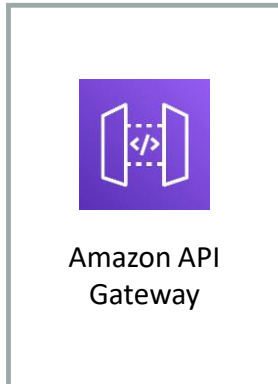
Module 13: Building Microservices and Serverless Architectures

Section 6: Extending serverless architectures with Amazon API Gateway

© 2020, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



Introducing Section 6: Extending serverless architectures with Amazon API Gateway.

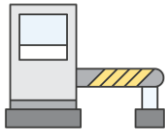


- Enables you to create, publish, maintain, monitor, and secure APIs that act as entry points to backend resources for your applications
- Handles up to hundreds of thousands of concurrent API calls
- Can handle workloads that run on –
 - Amazon EC2
 - Lambda
 - Any web application
 - Real-time communication applications
- Can host and use multiple versions and stages of your APIs

Amazon API Gateway is a fully managed service that enables you to create, publish, maintain, monitor, and secure APIs at any scale. You can use it to create Representational State Transfer (RESTful) and WebSocket APIs that act as an entry point for applications so they can access backend resources. Applications can then access data, business logic, or functionality from your backend services. Such services include applications that run on Amazon EC2, code that runs on Lambda, any web application, or real-time communication applications.

API Gateway handles all the tasks that are involved in accepting and processing up to hundreds of thousands of concurrent API calls. Such calls might include traffic management, authorization and access control, monitoring, and API version management. API Gateway has no minimum fees or startup costs. You pay only for the API calls you receive and the amount of data that is transferred out. With the API Gateway tiered-pricing model, you can reduce your cost as your API usage scales.

You can use API Gateway to host multiple versions and stages of your APIs.



Require
authorization



Apply resource
policies



Throttling
settings



Protection from
Distributed Denial of
Service (DDoS) and
injection attacks

When you make your APIs publicly available, you are exposed to attackers that try to exploit your services. With Amazon API Gateway, you can protect your APIs in several ways.

With Amazon API Gateway, you can optionally set your API methods to require authorization. When you set up a method to require authorization, you can use AWS Signature Version 4 or Lambda authorizers to support your own bearer token authentication strategy. *AWS Signature Version 4* is the process to add authentication information to AWS requests sent through HTTP. For security, most requests to AWS must be signed with an access key, which consists of an access key ID and secret access key. You use these AWS credentials to sign requests to your service and authorize access, like other AWS services. You can retrieve temporary credentials that are associated with a role in your AWS account by using Amazon Cognito. A *Lambda authorizer* is a Lambda function that authorizes access to APIs by using a bearer token authentication strategy like OAuth.

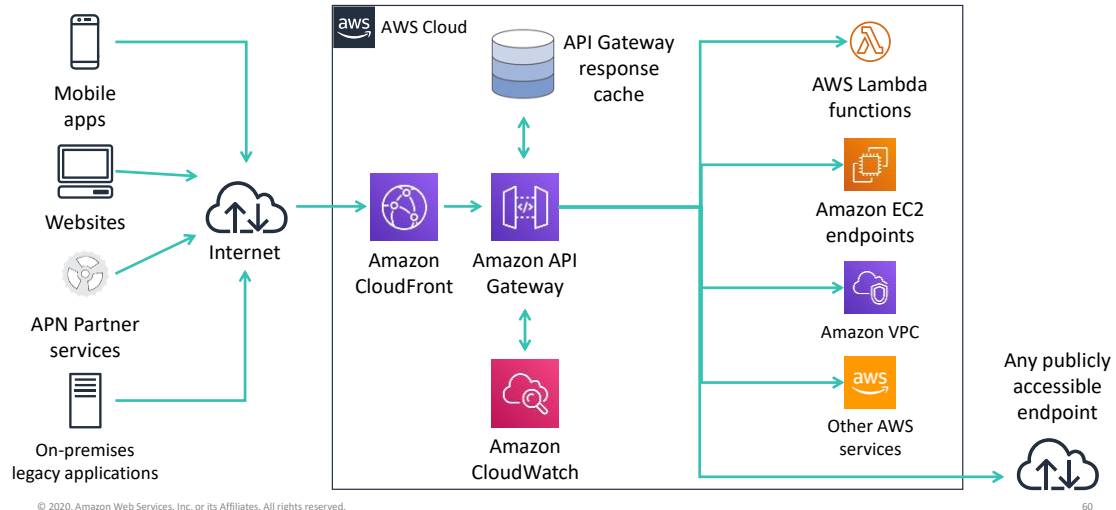
You can also apply a resource policy to an API to restrict access to a specific Amazon VPC or VPC endpoint. You can give an Amazon VPC or VPC endpoint from a different account access to the private API by using a resource policy.

Amazon API Gateway supports throttling settings for each method or route in your APIs. You can set a standard rate limit and a burst rate limit per second for each method in your REST APIs and each route in WebSocket APIs.

Additionally, you can use AWS WAF to secure your API Gateway APIs. [AWS WAF](#) is a web application firewall that helps protect your web applications from common web exploits that could affect availability, compromise security, or consume excessive resources.

For more information about how to secure your APIs with Amazon API Gateway, see the [Security & Authorization section](#) of Amazon API Gateway FAQs.

Amazon API Gateway: Common architecture example



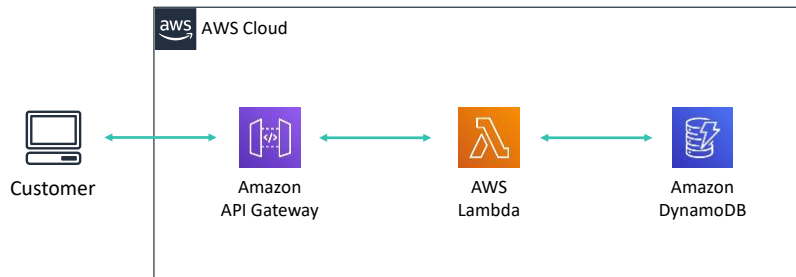
You can use Amazon API Gateway to provide the API layer for your applications. Here is an example of a common architecture with Amazon API Gateway.

In this example, front-end client applications and application services send traffic to API Gateway over the internet. Often, Amazon CloudFront is used to cache static content. API Gateway abstracts and exposes APIs that can call various backend applications. These applications include Lambda functions, Docker containers running on EC2 instances, virtual private clouds (VPCs), or any publicly accessible endpoint. If necessary, API Gateway can cache the response. Finally, all the API calls can be monitored with Amazon CloudWatch.

You can use API Gateway with other AWS managed services to build serverless backends for your applications. For example, API Gateway can proxy requests to Lambda functions that run your code and generate responses. You can even create APIs that proxy requests to other AWS services, such as Amazon Simple Storage Service (Amazon S3), without having to write any code. You can get production-scale backends running more quickly because you have less code to write, and you also offloaded operational burden to the AWS services.

For an example of how to use API Gateway with AWS Lambda, see this [AWS blog post](#).

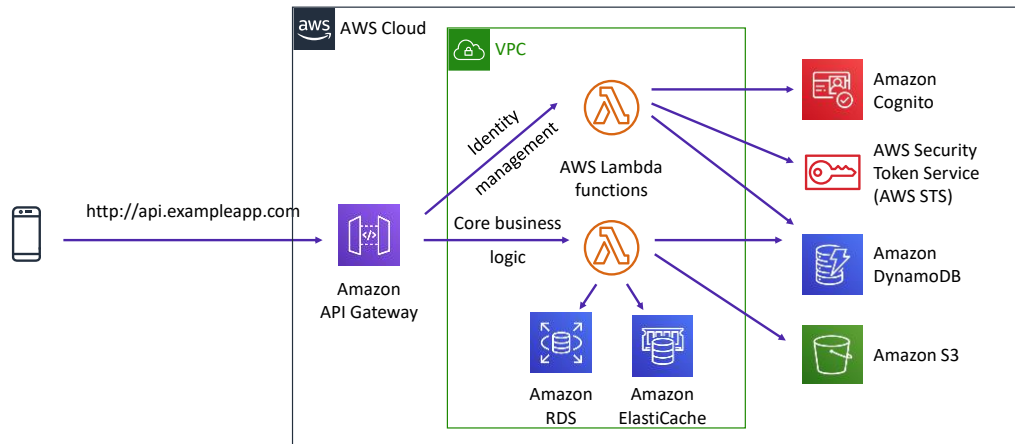
Example: RESTful microservices



Amazon API Gateway is deeply integrated with AWS Lambda. This example shows how the two services can be used together in an architecture for a RESTful microservice application.

In this example, customers can use your microservice by making HTTP API calls. Amazon API Gateway hosts RESTful HTTP requests from and responses to your customers. In this scenario, API Gateway provides built-in authorization, throttling, security, fault tolerance, request-response mapping, and performance optimizations. AWS Lambda contains the business logic to process incoming API calls and uses DynamoDB as a persistent storage. Amazon DynamoDB persistently stores microservice data and scales based on demand. Because microservices are designed to do one thing well, a schemaless NoSQL data store is regularly incorporated.

Example: Serverless mobile backend



Consider yet another example of how Amazon API Gateway can be used with Lambda in a serverless architecture as the gateway to the backend for a mobile application. Mobile apps are expected to have a fast, consistent, and feature-rich user experience, and they often have a global footprint. Further, mobile user patterns are dynamic, with unpredictable peak usage. Mobile apps require a rich set of mobile services that work together seamlessly without sacrificing control and flexibility of the backend infrastructure.

In this example, a mobile app sends a request to Amazon API Gateway, which forwards the request to a Lambda function that calls Amazon Cognito and AWS Security Token Service (AWS STS) to manage identity. Amazon Cognito authenticates users of your mobile app through external identity providers (IdPs) that support Security Assertion Markup Language (SAML) or OpenID Connect, social IdPs (for example, Facebook, Twitter, Amazon), and custom IdPs. After the identity of the mobile user is confirmed, another Lambda function runs the core business logic of the app.

Section 6 key takeaways



63

- Amazon API Gateway is a fully managed service that enables you to create, publish, maintain, monitor, and secure APIs at any scale.
- Amazon API Gateway acts as an **entry point to backend resources for your applications**. It abstracts and exposes APIs that can call various backend applications. These applications include Lambda functions, Docker containers that run on EC2 instances, VPCs, or any publicly accessible endpoint.
- Amazon API Gateway is **deeply integrated with Lambda**.

© 2020, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

Some key takeaways from this section of the module include:

- Amazon API Gateway is a fully managed service that enables you to create, publish, maintain, monitor, and secure APIs at any scale.
- Amazon API Gateway acts as an entry point to backend resources for your applications. It abstracts and exposes APIs that can call various backend applications. These applications include Lambda functions, Docker containers that run on EC2 instances, VPCs, or any publicly accessible endpoint.
- Amazon API Gateway is deeply integrated with Lambda.

Module 13: Building Microservices and Serverless Architectures

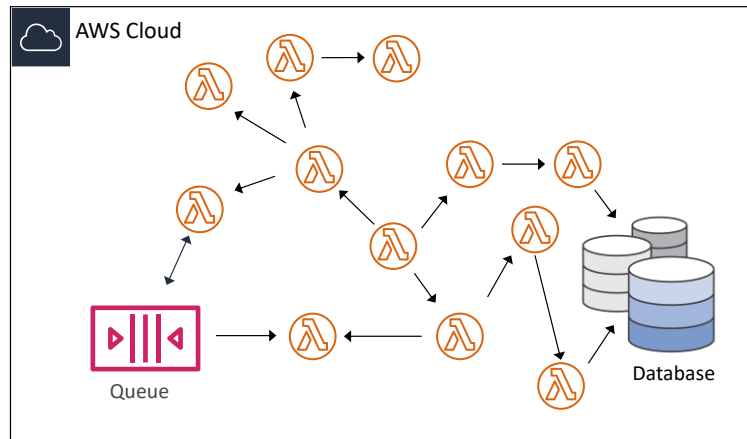
Section 7: Orchestrating microservices with AWS Step Functions

© 2020, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



Introducing Section 7: Orchestrating microservices with AWS Step Functions.

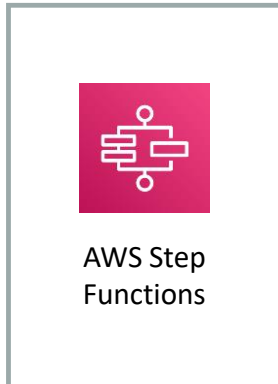
Challenges with microservice applications



As your application grows in size, the number of components increases, and many patterns of which tasks to run and in which order are possible. Consider a microservice application that is built by using Lambda functions, for example. You might want to invoke a Lambda function immediately after another function, and only if the first function runs successfully. You might want two functions to be invoked in parallel and then feed the combined results into a third function. Or you might want to choose which of two functions is invoked based on the output of another function.

Function invocation can result in an error for several reasons. Your code might raise an exception, time out, or run out of memory. The runtime that runs your code might encounter an error and stop. When an error occurs, your code might have run completely, partially, or not at all. In most cases, the client or service that invokes your function retries if it encounters an error, so your code must be able to process the same event repeatedly without unwanted effects. If your function manages resources or writes to a database, you must handle cases where the same request is made several times.

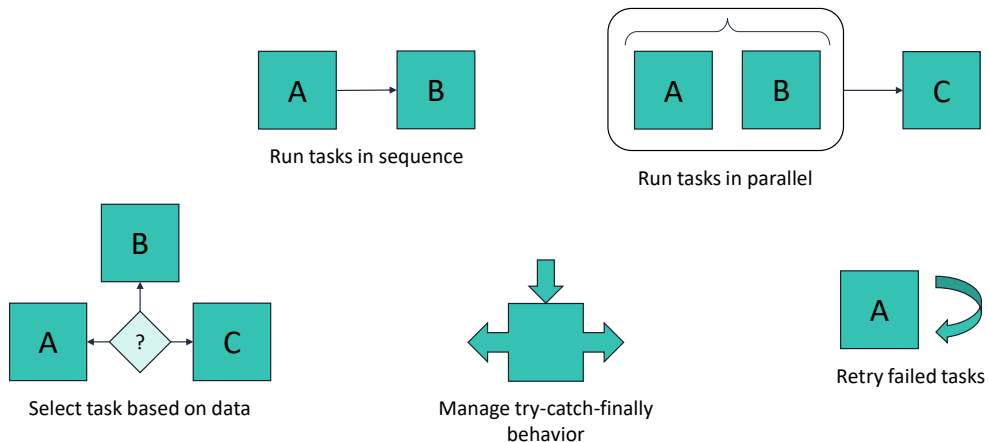
You need a way to coordinate the components of your application. This coordination layer must be able to scale automatically in response to changing workloads, and handle errors and timeouts. It must also maintain the state of your application while it is running, such as tracking which step is currently running and storing data that is moving between the steps of your workflow. These features help you build and operate your applications. You also want visibility into your application so that you can troubleshoot errors and track performance.



- Coordinates microservices by using visual workflows
- Enables you to step through the functions of your application
- Automatically triggers and tracks each step
- Provides simple error catching and logging if a step fails

[AWS Step Functions](#) is a web service that enables you to coordinate components of distributed applications and microservices by using visual workflows. Step Functions provides a reliable way to coordinate components and step through the functions of your application. Step Functions offers a graphical console so that you can visualize the components of your application as a series of steps. It automatically triggers and tracks each step, and it also retries when there are errors, so your application runs in order and as expected. Step Functions logs the state of each step, so you can diagnose and debug problems quickly.

Workflow coordination



AWS Step Functions manages the logic of your application for you. It implements basic primitives, such as running tasks sequentially or in parallel, branching, and timeouts. This technique removes extra code that might be repeated in your microservices and functions. AWS Step Functions automatically handles errors and exceptions with built-in try-catch and retry, whether the task takes seconds or months to complete. You can automatically retry failed or timed-out tasks. You can respond differently to different types of errors and recover gracefully by falling back to designated cleanup and recovery code.



A **state machine** is a collection of states that can do work.

Vending machine

Waiting for transaction



Soda selection

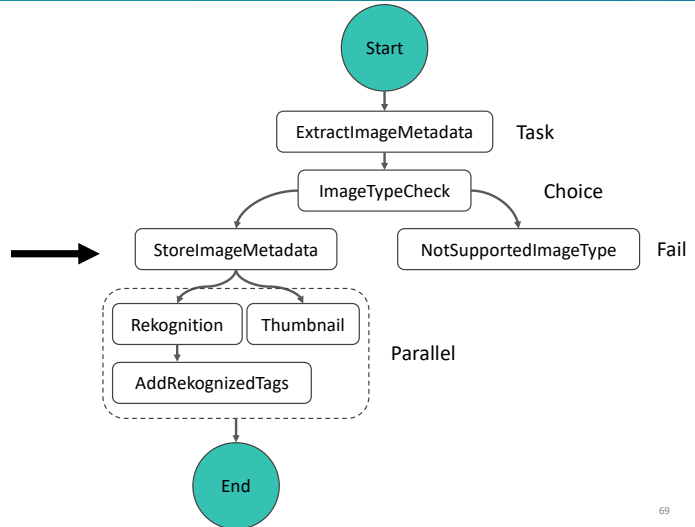
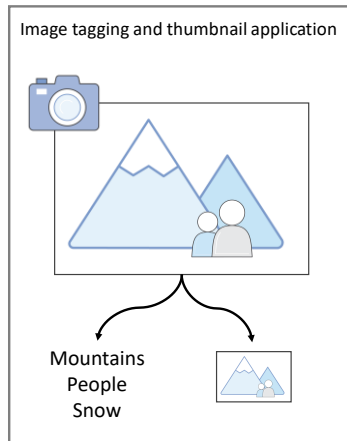


Vend soda

AWS Step Functions enables you to create and automate your own state machines within the AWS environment by using the JSON-based Amazon States Language. This language contains a structure that is made of various states, tasks, choices, error handling and more.

A *state machine* is collection of states that can do work.

A common example of a state machine is a soda vending machine. The machine starts in the operating state (waiting for a transaction), and then moves to soda selection when money is added. Next, it enters a vending state, where soda is deployed to the customer. After completion, the state returns back to operating.



A finite state machine can express an algorithm as a number of states, their relationships, and their input and output. *States* are elements in your state machine. Individual states can make decisions based on their input, perform actions, and pass output to other states.

State types

Task	A single unit of work performed by a state machine
Choice	Adds branching logic to a state machine
Fail	Stops a running state machine and marks it as a failure
Succeed	Stops a running state machine successfully
Pass	Passes its input to its output, without performing work
Wait	Delays from continuing for a specified time
Parallel	Creates parallel branches to run in your state machine
Map	Dynamically iterates steps

States can perform various functions in your state machine:

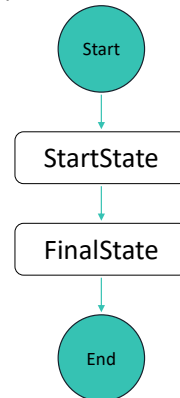
- Do some work in your state machine (a *Task* state)
- Make a choice between branches of state machines to run (a *Choice* state)
- Stop an running state machine with a failure or success (a *Fail* state or *Succeed* state)
- Pass its input to its output or inject some fixed data (a *Pass* state)
- Provide a delay for a certain amount of time or until a specified time and date (a *Wait* state)
- Begin parallel branches to run in your state machine (a *Parallel* state)
- Dynamically iterate steps (a *Map* state)

For more information about state types, see [States](#) in the AWS Documentation.

Define workflow in JSON
by using the Amazon States Language

```
{
  "Comment": "An example of the ASL.",
  "StartAt": "StartState",
  "States": {
    "StartState": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:us-east-1:123456789012:function:my-function",
      "Next": "FinalState"
    },
    "FinalState": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:us-east-1:123456789012:function:my-function",
      "End": true
    }
  }
}
```

Visualize workflow in the
Step Functions console

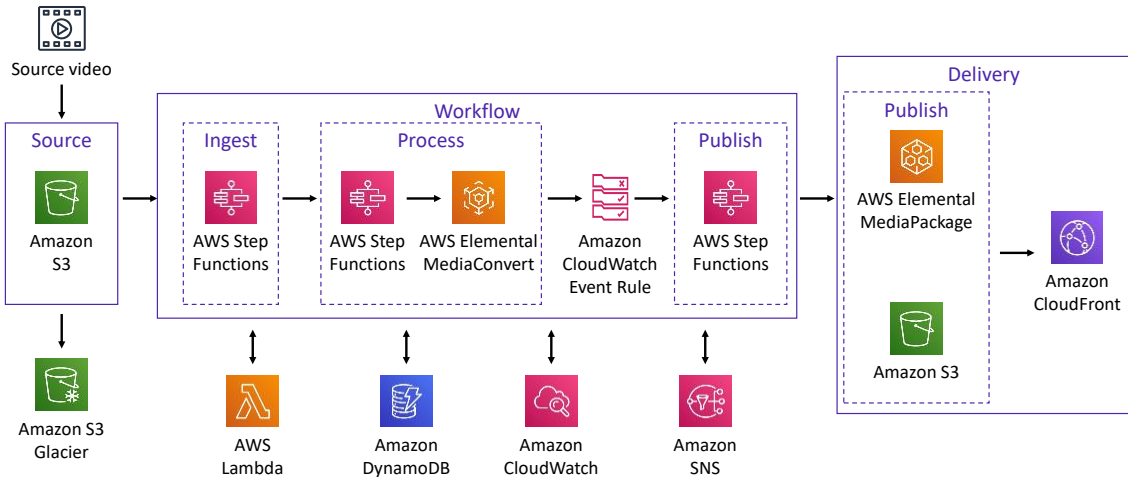


You define state machines by using the Amazon States Language. The Amazon States Language is a JSON-based, structured language. AWS Step Functions then represents the JSON structure in a real-time graphical view. In this way, you can visualize your state machine directly in the Step Functions console.

Here, you see an example of a state machine with two task state types.

For more information about the Amazon States Language, see the [AWS Documentation](#).

AWS Step Functions example: Video-on-demand (VOD) architecture



© 2020, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

72

This architecture diagram shows an example use case for AWS Step Functions in a video-on-demand (VOD) solution. Another key component in this architecture is AWS Elemental MediaConvert, which is a file-based video transcoding service with broadcast-grade features. It enables you to create VOD content for broadcast and multiscreen delivery at scale.

In this solution, source videos and metadata files are ingested and processed for playback on a wide range of devices.

- AWS Step Functions creates **Ingest**, **Process**, and **Publish** step functions.
- AWS Lambda functions perform the work of each step and process error messages.
- Amazon S3 buckets store source and destination media files.
- Amazon CloudWatch is used for logging.
- Amazon CloudWatch event rules for AWS Elemental MediaConvert notifications.
- An Amazon DynamoDB table stores data captured through the workflow.
- Amazon SNS topics send encoding, publishing, and error notifications.
- The transcoded media files are stored for on-demand delivery to users through Amazon CloudFront.

For more information about this architecture, see [Architecture Overview](#).

Section 7 key takeaways



73

- **AWS Step Functions** is a web service that enables you to coordinate components of distributed applications and microservices by using visual workflows
- AWS Step Functions enables you to **create and automate your own state machines** within the AWS environment
- AWS Step Functions **manages the logic of your application for you, and it implements basic primitives**, such as sequential or parallel branches, and timeouts
- You define state machines by using the **Amazon States Language**

© 2020, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

Some key takeaways from this section of the module include:

- AWS Step Functions is a web service that enables you to coordinate components of distributed applications and microservices by using visual workflows
- AWS Step Functions enables you to create and automate your own state machines within the AWS environment
- AWS Step Functions manages the logic of your application for you, and it implements basic primitives, such as sequential or parallel branches, and timeouts
- You define state machines by using the Amazon States Language

Module 13 – Challenge Lab: Implementing a Serverless Architecture for the Café

74



© 2020, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

You will now complete Module 13 – Challenge Lab: Implementing a Serverless Architecture for the Café.

The business need: A serverless environment



Frank and Martha want to get daily email reports about all the orders that were placed on the website. Olivia advises Sofía and Nikhil that non-business-critical reporting tasks should be kept separate from the production web server instance.

Sofía and Nikhil want to further decouple the architecture and move the cron job into a managed, serverless environment that will scale well and reduce costs.



© 2020, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

75

Frank and Martha want to get daily reports via email about all the orders that were placed on the website. Frank wants to anticipate demand so he can bake the correct number of desserts going forward (reducing waste). Martha wants to identify any patterns in the café's business (analytics). Currently, Sofía has set up a cron job on the web server instance that sends these daily order report email messages to Frank and Martha. However, the cron job is resource-intensive and reduces the performance of the web server.

Olivia advises Sofía and Nikhil that non-business-critical reporting tasks should be kept separate. Sofía and Nikhil want to further decouple the architecture and move the cron job into a managed, serverless environment that will scale well and reduce costs.

Challenge lab: Tasks

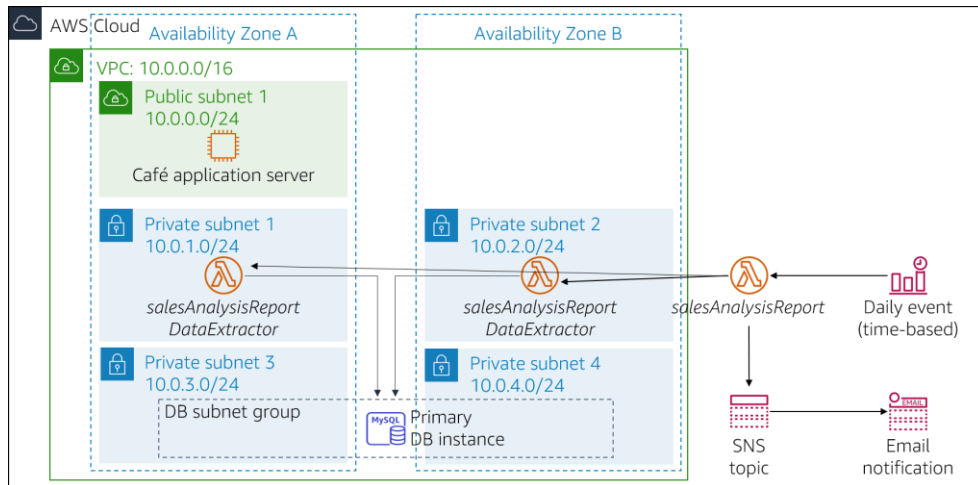


1. Downloading the source code
2. Creating the *DataExtractor* Lambda function in the VPC
3. Creating the *salesAnalysisReport* Lambda function
4. Creating an SNS topic
5. Creating an email subscription to the SNS topic
6. Testing the *salesAnalysisReport* Lambda function
7. Setting up an Amazon EventBridge event to trigger the Lambda function each day

In this challenge lab, you will complete the following tasks:

1. Downloading the source code
2. Creating the *DataExtractor* Lambda function in the VPC
3. Creating the *salesAnalysisReport* Lambda function
4. Creating an SNS topic
5. Creating an email subscription to the SNS topic
6. Testing the *salesAnalysisReport* Lambda function
7. Setting up an Amazon EventBridge event to trigger the Lambda function each day

Challenge lab: Final product



The diagram summarizes what you will have built after you complete the lab.



~ 90 minutes



Begin Module 13 – Challenge Lab: Implementing a Serverless Architecture for the Café

It is now time to start the challenge lab.

Challenge lab debrief: Key takeaways



Your educator might choose to lead a conversation about the key takeaways from this challenge lab after you have completed it.

Module 13: Building Microservices and Serverless Architectures

Module wrap-up

© 2020, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



It's now time to review the module and wrap up with a knowledge check and discussion of a practice certification exam question.

Module summary



In summary, in this module, you learned how to:

- Indicate the characteristics of microservices
- Refactor a monolithic application into microservices and use Amazon ECS to deploy the containerized microservices
- Explain serverless architecture
- Implement a serverless architecture with AWS Lambda
- Describe a common architecture for Amazon API Gateway
- Describe the types of workflows that AWS Step Functions supports

In summary, in this module, you learned how to:

- Indicate the characteristics of microservices
- Refactor a monolithic application into microservices and use Amazon ECS to deploy the containerized microservices
- Explain serverless architecture
- Implement a serverless architecture with AWS Lambda
- Describe a common architecture for Amazon API Gateway
- Describe the types of workflows that AWS Step Functions supports

Complete the knowledge check



It is now time to complete the knowledge check for this module.

Sample exam question

An organization hosts 10 microservices, each in an Auto Scaling group behind individual Classic Load Balancers. Each EC2 instance is running at optimal load.

Which of the following actions would enable the organization to reduce costs without impacting performance?

- A. Reduce the number of EC2 instances behind each Classic Load Balancer.
- B. Change instance types in the Auto Scaling group launch configuration.
- C. Change the maximum size but leave the desired capacity of the Auto Scaling groups.
- D. Replace the Classic Load Balancers with a single Application Load Balancer.

Look at the answer choices and rule them out based on the keywords that were previously highlighted.

The correct answer is D: “Replace the Classic Load Balancers with a single Application Load Balancer.” Choice A can be eliminated—Because the EC2 instances are running at optimal load, they will become overloaded if you reduce their number. Choice B can be eliminated—You will not reduce costs if you choose a larger instance size. Alternatively, if you decrease the instance size, the EC2 instances will become overloaded because they are already running at optimal load. Choice C can also be eliminated—Costs will remain the same or increase if you increase the maximum size and keep the desired capacity. By process of elimination, choice D is correct. You can also use a single Application Load Balancer to route requests to all the services for your application instead of using a single Classic Load Balancer for each microservice.

Additional resources



- [Break a Monolith Application into Microservices](#) project
- [Serverless Architectures with AWS Lambda](#) whitepaper
- [AWS Serverless Multi-Tier Architectures with Amazon API Gateway and AWS Lambda](#) whitepaper
- [AWS Well-Architected Framework: Serverless Application Lens](#) whitepaper
- [Creating and Using Lambda Functions](#) tutorial

If you want to learn more about the topics covered in this module, you might find the following additional resources helpful:

- [Break a Monolith Application into Microservices](#) project
- [Serverless Architectures with AWS Lambda](#) whitepaper
- [AWS Serverless Multi-Tier Architectures with Amazon API Gateway and AWS Lambda](#) whitepaper
- [AWS Well-Architected Framework: Serverless Application Lens](#) whitepaper
- [Creating and Using Lambda Functions](#) tutorial

Thank you

© 2020 Amazon Web Services, Inc. or its affiliates. All rights reserved. This work may not be reproduced or redistributed, in whole or in part, without prior written permission from Amazon Web Services, Inc. Commercial copying, lending, or selling is prohibited. Corrections or feedback on the course, please email us at: aws-course-feedback@amazon.com. For all other questions, contact us at: <https://aws.amazon.com/contact-us/aws-training/>. All trademarks are the property of their owners.



Thank you for completing this module.