



AWS Academy Cloud Architecting  
Module 14 Student Guide  
Version 3.0.2

200-ACACAD-30-EN-SG

© 2024, 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.

All trademarks are the property of their owners.

# Contents

[Module 14: Building Serverless Architectures and Microservices](#)

4



Welcome to the Building Serverless Architectures and Microservices module. This module introduces you to serverless architectures and microservices. You will also learn how to use AWS services for serverless architectures and microservices.



This introduction section describes the content of this module.

## Module objectives



This module prepares you to do the following:

- Define serverless architectures.
- Identify the characteristics of microservices.
- Architect a serverless solution with AWS Lambda.
- Define how containers are used in Amazon Web Services (AWS).
- Describe the types of workflows that AWS Step Functions supports.
- Describe a common architecture for Amazon API Gateway.
- Use the AWS Well-Architected Framework principles when building serverless architectures.

©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

3

## Module overview

### Presentation sections

- Thinking serverless
- Architecting serverless microservices
- Building serverless architectures with AWS Lambda
- Building microservice applications with AWS container services
- Orchestrating microservices with AWS Step Functions
- Extending serverless architectures with Amazon API Gateway
- Applying AWS Well-Architected Framework principles to microservices and serverless architectures



©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

4

### Demos

- Using AWS Lambda with Amazon S3
- Running a Container

### Activity

- Decomposing a Monolithic Application with AWS API Gateway

### Knowledge checks

- 10-question knowledge check
- Sample exam question

The objectives of this module are presented across multiple sections.

You will also participate in an activity to make service selections for a given microservices use case.

You will also view two demonstrations that show how AWS Lambda works with Amazon Simple Storage Service (Amazon S3) and how to run a container in AWS.

The module wraps up with a 10-question knowledge check delivered in the online course and a sample exam question to discuss in class.

The labs in this module are described on the next slide.

## Hands-on labs in this module

### Guided labs

- Implementing a Serverless Architecture on AWS
- Breaking a Monolithic Node.js Application into Microservices (Optional)

### Challenge (Café) lab

- Implementing a Serverless Architecture for the Café



©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

5

This module includes the hands-on labs that are listed. With guided labs, you are provided step-by-step instructions. With the café (challenge) lab, you work on updating the architecture for the café. Additional information about each lab is included in the student guide where the lab takes place, and detailed instructions are provided in the lab environment.



**As a cloud architect building microservices:**



- I need to recognize when to choose AWS serverless architectures and which services to choose for each use case so that I can build a performance- and cost-optimized solution.
- I need to apply event-driven architectures with microservices so that I can build scalable and resilient microservice architectures.
- I need to know when to use workflow orchestrations so that my microservice architectures can handle workflow failures and require minimal manual intervention.

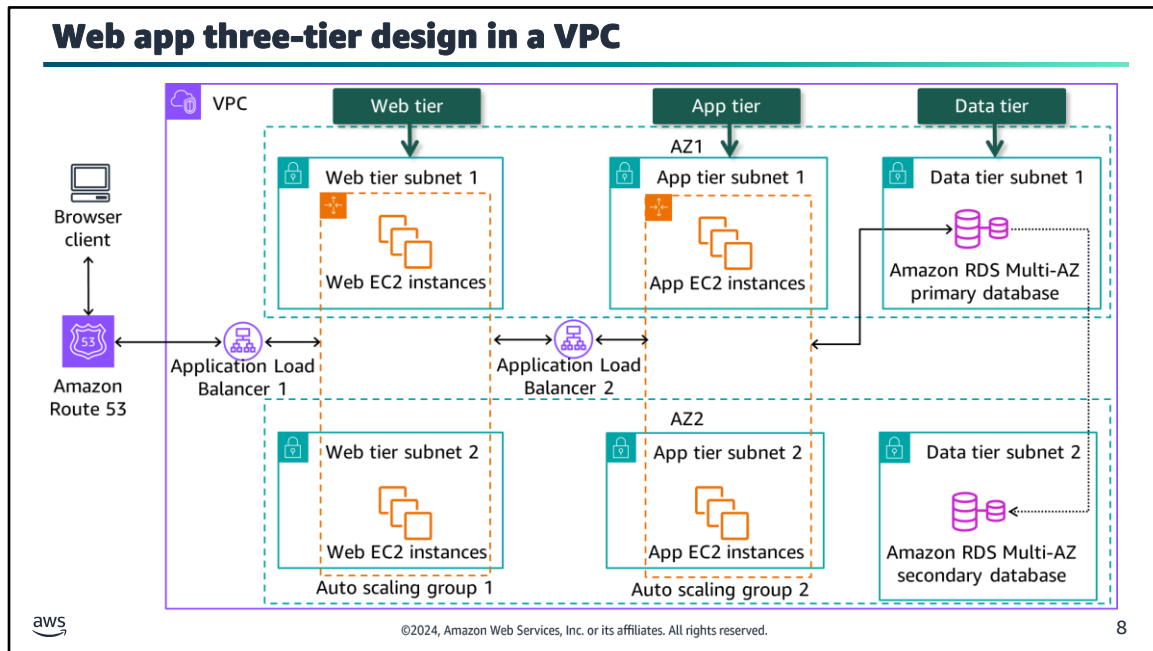
©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

6

This slide asks you to take the perspective of a cloud architect as you think about how to approach cloud network design. Keep these considerations in mind as you progress through this module, remembering that the cloud architect should work backwards from the business need to design the best architecture for a specific use case. As you progress through the module, consider the café scenario presented in the course as an example business need and think about how you would address these needs for the fictional café business.



This section looks at serverless architectures in AWS.



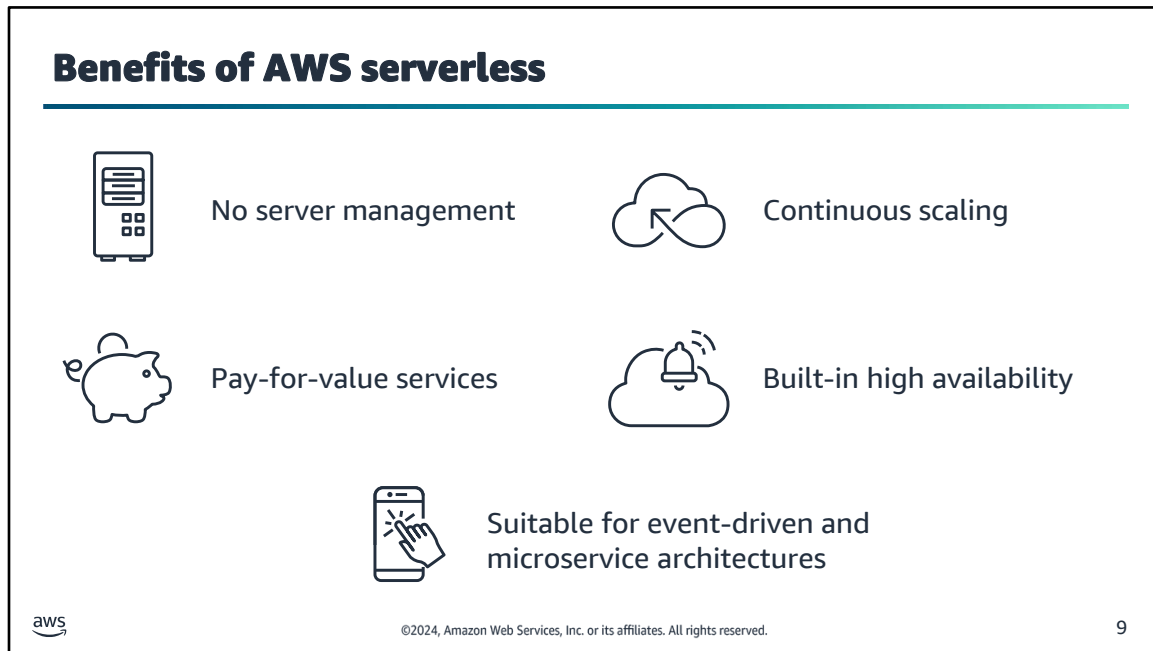
8

When you need to deploy a web application in a virtual private cloud (VPC) as in the diagram above, you will typically use a web domain pointing to an Application Load Balancer that points to a web tier layer. For high availability, the web tier is deployed on multiple Amazon Elastic Compute Cloud (Amazon EC2) instances across two availability zones (AZs) in an auto scaling group. This will ensure availability even when an Amazon EC2 instance fails or an AZ isn't available. The web tier layer is responsible for the presentation layer of the browser client.

The application tier is responsible for the business logic of the web app and is accessed from the web tier through another Application Load Balancer pointing to the app tier. Similar to the web tier, it has a high availability design across two AZs with an auto scaling group. The web tier and the app tier is managed by the AWS customer who must ensure that the instances security patches and software updates are regularly applied. The instances are monitored and alarms are configured in Amazon CloudWatch.

The app tier accesses the data tier for data requests. The data tier consists of an AWS managed Amazon Relational Database Service (Amazon RDS) Multi-AZ primary database in one AZ with synchronous replication to the Amazon RDS secondary database in the second AZ. RDS is a managed AWS service which means that AWS is responsible for managing the EC2 database instances and applying security patches and software updates. When the primary database fails, the secondary database is promoted to be the primary database.

The multi-tier architecture pattern provides a general framework to ensure that decoupled and independently scalable application components can be separately developed, managed, and maintained. The network acts as the boundary between tiers and often requires creating many undifferentiated application components. Components such as message queues and authentication components look the same for any application.



What if you are a single web developer for an application or a brand new startup that wants to minimize infrastructure cost? Or what if you don't have adequate staff to do maintenance and support of the EC2 instances? Or what if you want to break down the monolith tier deployments into smaller, agile chunks? AWS serverless services addresses these scenarios.

Back in 2013, AWS noticed that customers were using EC2 instances for really short periods. This began the process leading to AWS serverless services where the runtime is fully managed by AWS and the payment model is based on service usage. AWS serverless resources can be used for very short periods of time, such as double digit milliseconds, which improves resource efficiency and provides granular cost management.

An AWS service is deemed to be serverless when you can deploy an application without having to think about servers or the sizing of servers. AWS manages the runtime environment and sizing of the service. As a result, you can focus on your core product and make application deployments smaller and faster. For example, AWS Lambda functions can be deployed with one click in the AWS console and is generally ready within seconds.

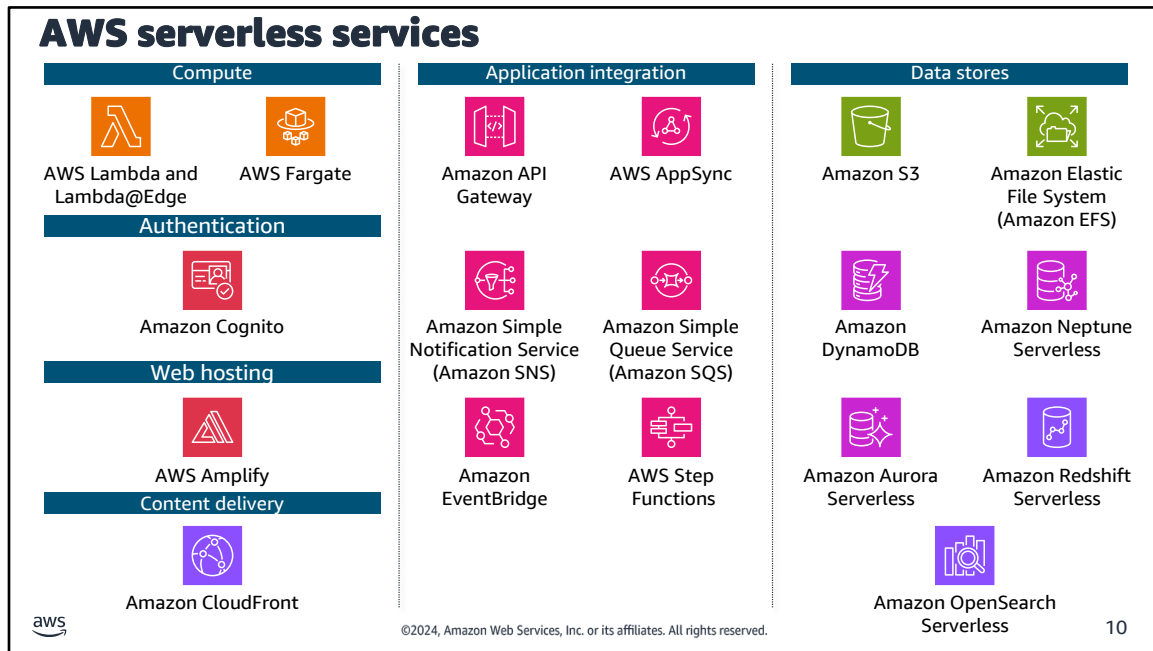
The service pricing model is typically based on a usage model where cost is based on how much the service is used and how much storage is used. This eliminates paying for services that aren't in use. This model is especially beneficial for startup companies when the business is still being grown to keep overhead costs low. For example, Amazon Simple Storage Service (Amazon S3) buckets have different usage tiers based on usage patterns each with its own price.

Many AWS serverless services have the option to automatically scale based on a specific metric. AWS will then scale the service resources based on the metric. As an example, Amazon DynamoDB can increase or decrease a table's provisioned read and write capacity throughput based on actual traffic patterns.

AWS serverless services come with fault tolerance and high availability built in. Serverless data stores operate over three AZs, and serverless compute resources, like AWS Fargate containers, run in an isolated environment that isn't affected by other container failures.

AWS serverless services are suitable to use for event-driven and microservice architectures. As you learned in

the previous module, microservices is an architectural style that structures an application as a collection of independent, loosely coupled services. Event-driven architecture (EDA) is a modern architecture pattern built from small, decoupled services that publish, consume, or route events. Events are messages sent between services. This architecture makes it easier to scale, update, and independently deploy separate components of a system. They can be invoked by events, they can emit events, and they have built-in capabilities for building with events. For example, an AWS Lambda function can handle queue message events from Amazon Simple Queue Service (Amazon SQS), or streaming events from Amazon Kinesis Data Streams or Amazon DynamoDB Streams, or object events from Amazon S3.



Modern applications are built serverless-first, a strategy that prioritizes the adoption of serverless services, so you can increase agility throughout your application stack. AWS developed serverless services for all three layers of an application stack: compute, application integration, and data stores. There are even services for specialized domains such as analytics.

AWS serverless compute options are AWS Lambda functions and AWS Fargate containers. AWS Lambda is an event-driven compute service that lets you run code functions without provisioning or managing servers. You can deploy Lambda functions in an AWS Region or at an edge location with Lambda@Edge. AWS Fargate is a serverless container solution that can be used with Amazon Elastic Container Service (Amazon ECS) or Amazon Elastic Kubernetes Service (Amazon EKS).

Serverless application integration can be subdivided into API publishers, messaging applications, and orchestration and events categories. Amazon API Gateway and AWS AppSync are two AWS serverless options to publish and manage APIs. Amazon API Gateway allows you to create, publish, and maintain restful and HTTP APIs. AWS AppSync lets you deploy and manage GraphQL APIs. One request can collect data from different data stores for a seamless user experience. AWS serverless messaging applications include Amazon Simple Notification Service (Amazon SNS) and Amazon SQS. Amazon SNS follows a publish and subscribe model for application-to-application and application-to-person communication.

Amazon SQS is a message queuing service used to decouple software application components. Serverless orchestration and events are supplied by AWS Step Functions and Amazon EventBridge. AWS Step Functions is a workflow orchestrator used to sequence multiple AWS services to perform complex transactions. AWS EventBridge is an event bus service that routes events to configured destinations while checking message schemas.

AWS serverless data stores include the following services:

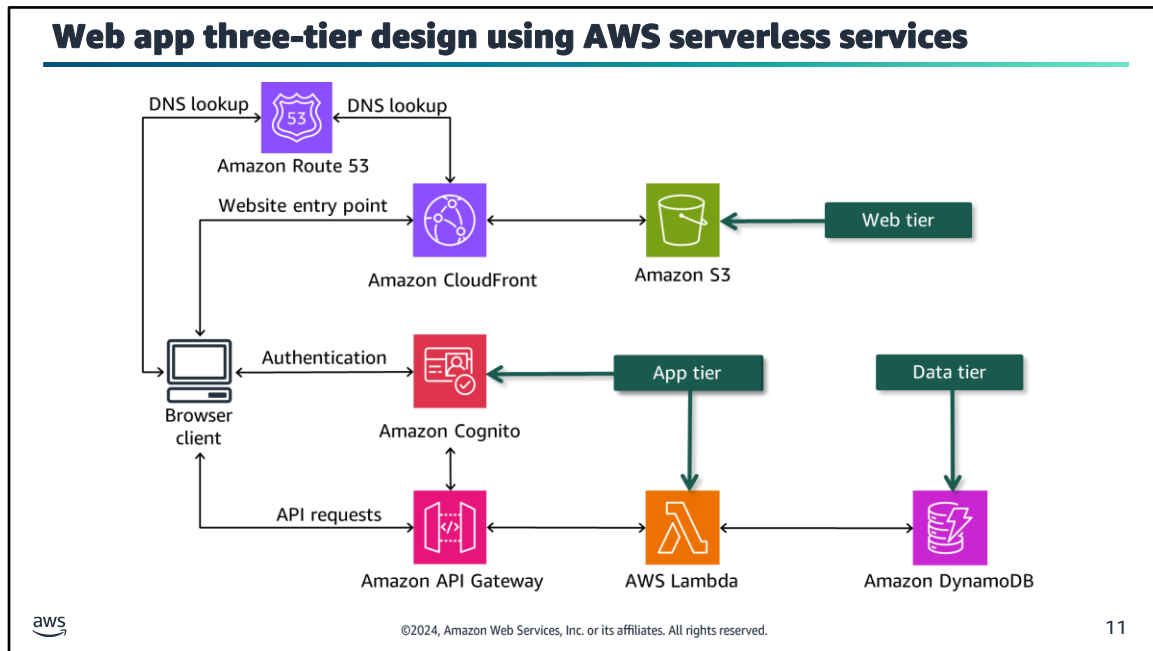
- Amazon S3 stores objects in buckets.
- Amazon Elastic File System (Amazon EFS) stores files available to EC2 instances, AWS Lambda, and Amazon ECS, Amazon EKS, and Fargate containers.
- Amazon DynamoDB is a key-value and document database service for single-digit millisecond response

times.

- Amazon OpenSearch Serverless provides data search and log analytics without provisioning resources.
- Amazon Neptune Serverless is a graph database that scales based on usage.
- Amazon Redshift Serverless is a columnar relational data warehouse with a pay-for-what-you-use pricing model.
- Amazon Aurora Serverless is a MySQL and PostgreSQL compatible relational database that scales based on usage.

This isn't an exhaustive list of serverless data storage options. AWS also offers specialized serverless databases such as Amazon Timestream, Amazon Quantum Ledger Database (Amazon QLDB), and Amazon Keyspaces (for Apache Cassandra).

For application authentication, use Amazon Cognito to manage application user pools and integrate with external identity providers. AWS Amplify is a solution to build and host web and mobile apps. Amazon CloudFront content delivery service can be used to distribute static web content from Amazon S3 acting as a web server.



11

Compare the three-tier VPC architecture with this diagram implementing the three architecture layers.

A commonly used serverless pattern for a three-tier web application design is depicted in the above example. Amazon CloudFront distributes the static web front-end from an Amazon S3 bucket to the browser client.

The app tier user authentication is handled by an Amazon Cognito user pool which provides an JSON web token (JWT) for authorization of API requests. API Gateway receives API requests and validates the JWT token with Amazon Cognito. If the token is valid the request is passed to an AWS Lambda function to run the application business logic.

To retrieve application data, the Lambda function requests data from Amazon DynamoDB used as the data tier.



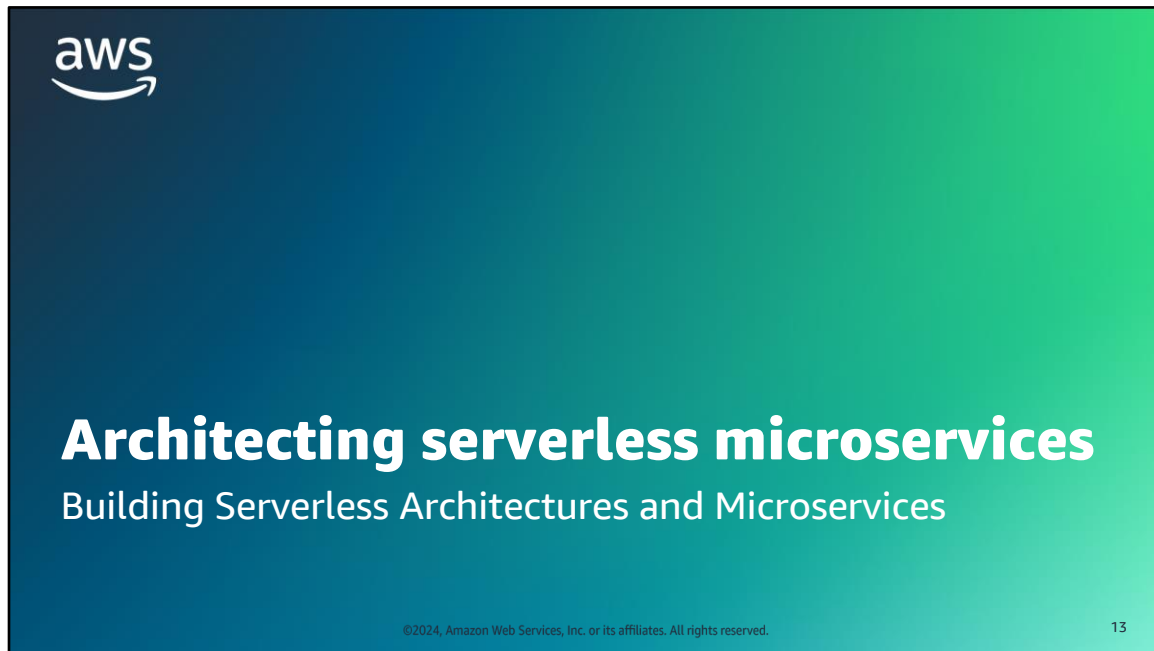
## Key takeaways: Thinking Serverless



- AWS serverless services provide you the following benefits:
  - No server management
  - Pay-for-value services
  - Continuous scaling
  - Built-in fault tolerance
- Serverless services are suitable for event-driven and microservice architectures.

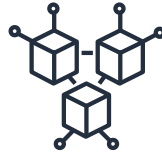
©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

12



This section introduces the concept of microservices.

## Characteristics of a microservice



### Autonomous

- Can be developed and deployed without affecting other microservices
- Scales independently
- Does not share code with other microservices
- Communicates over well-defined APIs

### Specialized

- Performs a single business function solving a specific problem
- Is owned by a small development team that chooses development tools
- Is stateless
- Has its own data store



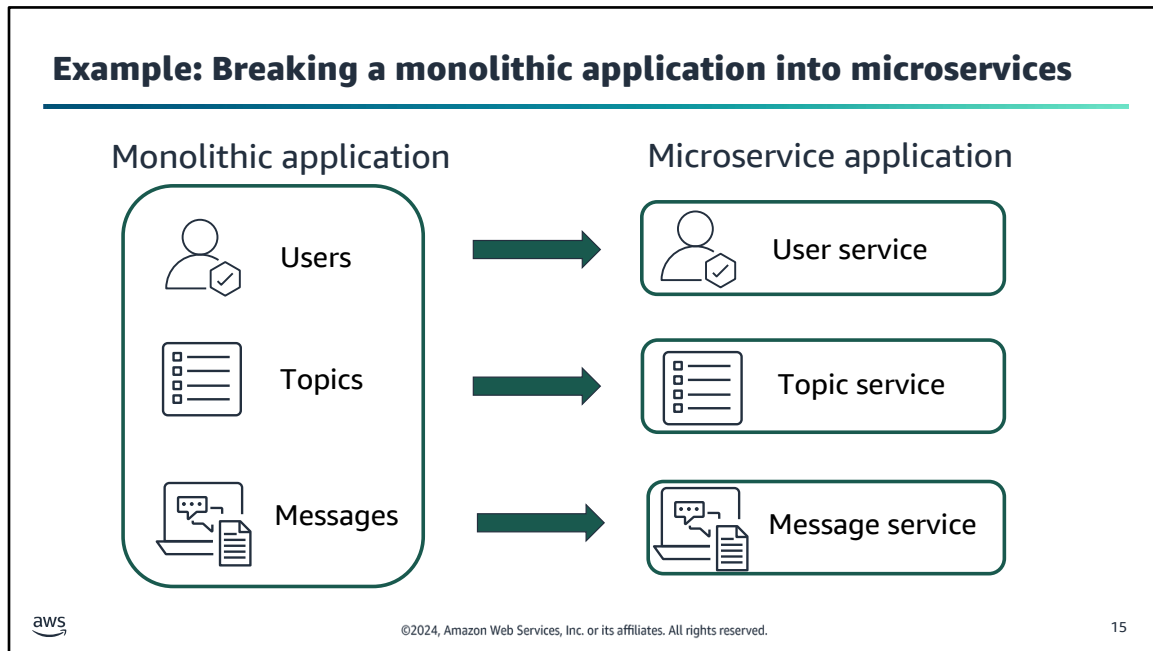
©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

14

With a microservices architecture, an application is built as independent components that run each application process as a service. These services communicate through a well-defined interface using lightweight APIs. Services are built for business capabilities and each service performs a single function. Because they are independently run, each service can be updated, deployed, and scaled to meet demand for specific functions of an application.

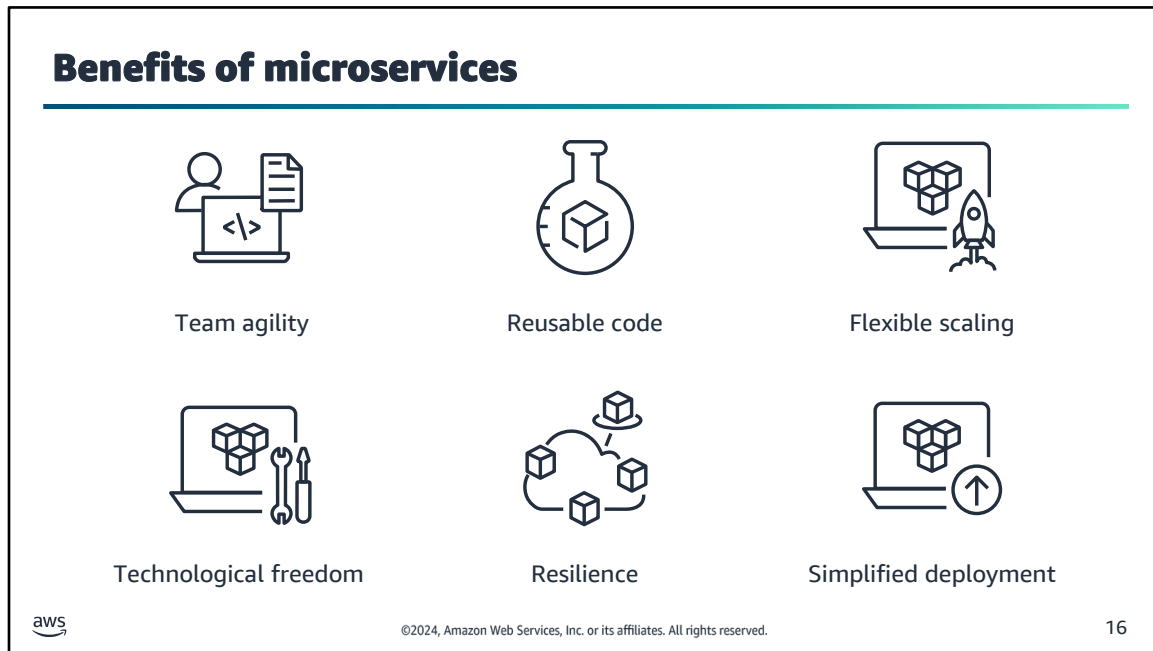
A microservice is autonomous, meaning that it can be operated and scaled without affecting the functioning of other services. The microservice doesn't share any of its code or implementation with other services. Any communication between individual microservices happens through well-defined, lightweight APIs.

A microservice is specialized, meaning that it's designed for a specific set of capabilities. It focuses on solving a specific problem for a single business function. For example, logging into a website can be developed as one microservice. Typically a microservice is owned by a small development team who decides on the most suitable runtime and coding language to use. A microservice should also be stateless to ensure fast instantiation and scaling. Caching state is ideal for microservices to recover quickly from hardware failure. Microservices own their own data stores to facilitate atomic, consistent, isolated, and durable (ACID) transactions if needed. If developers contribute more code to a microservice over time and the microservice becomes complex, it can be split into separate microservices.



In this example, a forum application is a monolithic, tightly coupled set of processes for users, topics, and messages. The processes 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 microservice. The microservices 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.



Microservices improve development team agility. Microservices foster an organization of small, independent teams that take ownership of their services. Teams act within a small and well understood context and are empowered to work more independently and more quickly. This shortens development into quick cycles.

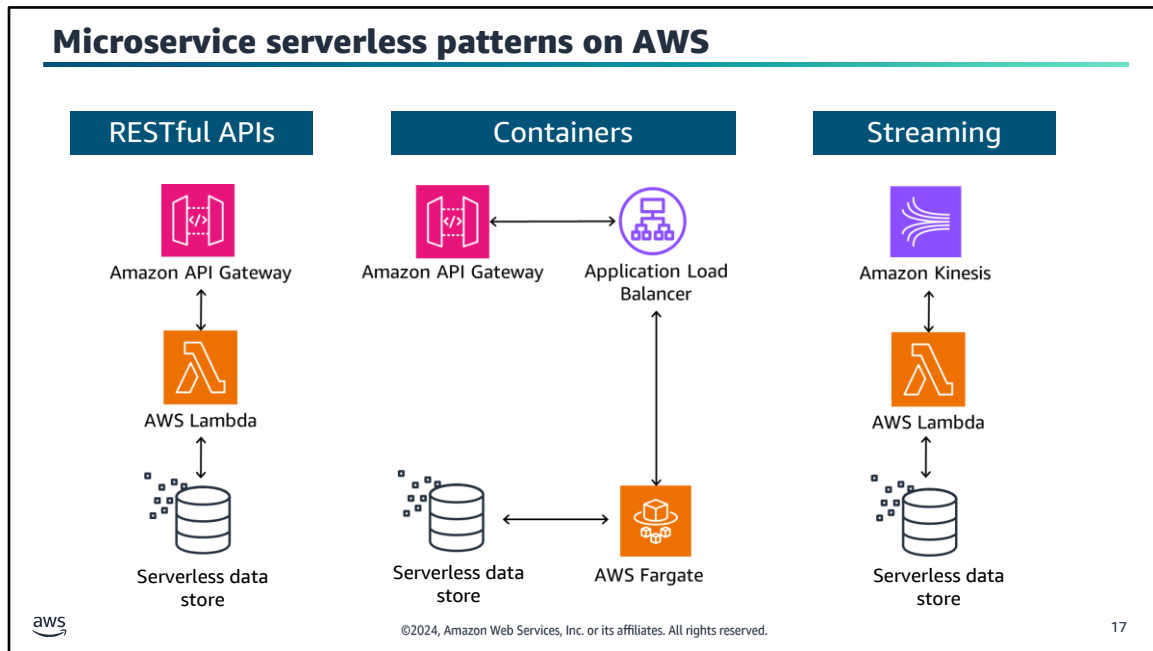
Dividing software into small, well-defined modules enables teams to use functions for multiple purposes. A service written for a certain function can be reused as a building block for another feature. This allows developers to combine existing microservices to create new capabilities without writing code from scratch.

Microservices allow flexible scaling where each microservice is independently scaled to meet demand for the application feature that it supports. This enables teams to right-size infrastructure needs, accurately measure the cost of a feature, and maintain availability if a microservice experiences a spike in demand.

Microservices allow development teams the technological freedom to make technical decisions autonomously. They understand that architectures don't follow a "one size fits all" approach. Teams have the freedom to choose the best tool to solve their specific problems. As a consequence, teams building microservices can choose the best tool for each job. The team is responsible for creating the microservice as well as deploying and maintaining it in a production environment.

With microservices, applications handle total service failure by degrading functionality and not crashing the entire application as is the case with monolithic applications.

Microservices simplify deployment by using continuous integration and continuous delivery pipelines. This simplifies trying out new ideas and rolling back if something doesn't work. The low cost of failure enables experimentation, helps to update code, and accelerates time-to-market for new features.



17

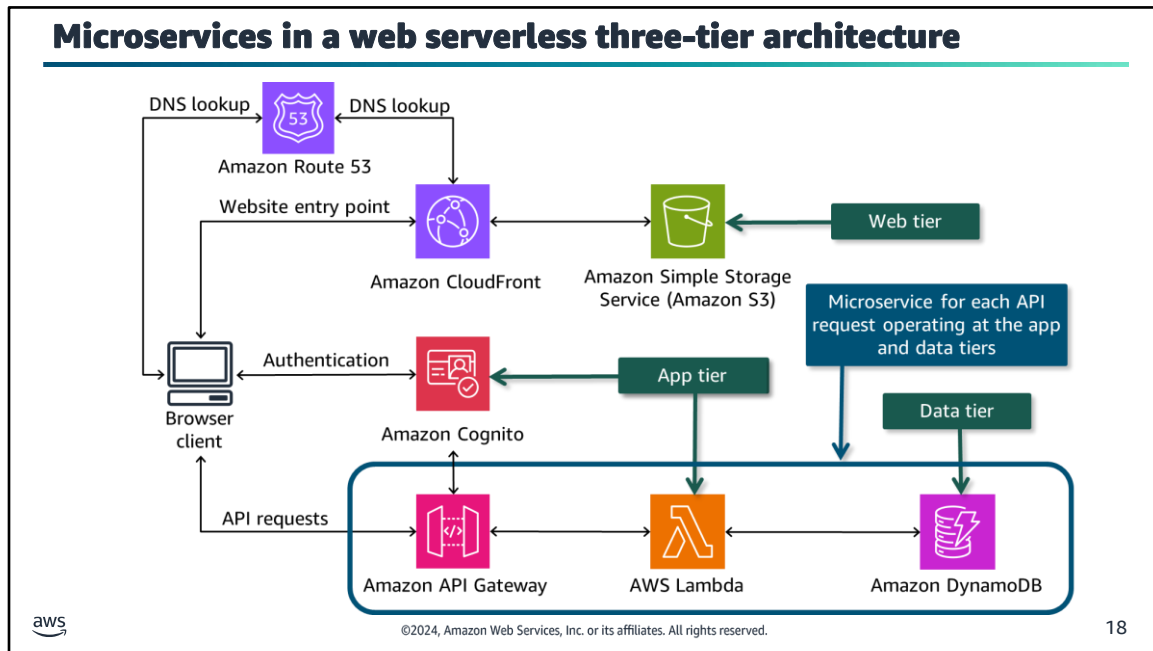
There are many microservices architecture patterns on AWS to choose from. Here are a few common serverless compute patterns:

**RESTful APIs:** Representational State Transfer (REST) is an architectural style user for communication between services that require compute to be stateless. Data and functions are resources with the state of the source at a specific time called the resource representation. RESTful architectures can be implemented with Amazon API Gateway and AWS Lambda as a serverless solution. REST aligns with the AWS Lambda design principle of stateless functions. The AWS Lambda service runs your code for a set amount of time before timing out with a maximum duration of 15 minutes.

**Containers:** If a microservice requires more than 15 minutes to successfully complete, containers can be used as the compute part of the service. For serverless microservice containers, AWS Fargate can be deployed behind an Application Load Balancer getting requests from Amazon API Gateway. If serverless is not a hard requirement, Amazon Elastic Container Service (Amazon ECS) or Amazon Elastic Kubernetes Service (Amazon EKS) can be used instead of AWS Fargate.

**Streaming:** Microservices can also be invoked by a streaming service. AWS Lambda is integrated with Amazon Kinesis and can scale with Kinesis to provide a streaming microservice solution.

For more information on these architecture patterns as well as deployable templates, see the Serverless pattern collection link in the resource list.



18

Microservices operate at the app and data tier of a web three-tier architecture. Each API request fulfills a business capability. Combining Amazon API Gateway as the API, AWS Lambda for compute, and Amazon DynamoDB as the data store results in a serverless microservice pattern. AWS Lambda functions are ideal for the compute component of microservices as each function has a maximum of 15 minutes to run.

So from the example above, you can see that microservices form only a part of a three-tier architecture solution and should not be considered a stand alone full three-tier architecture solution.

## Key takeaways: Architecting serverless microservices

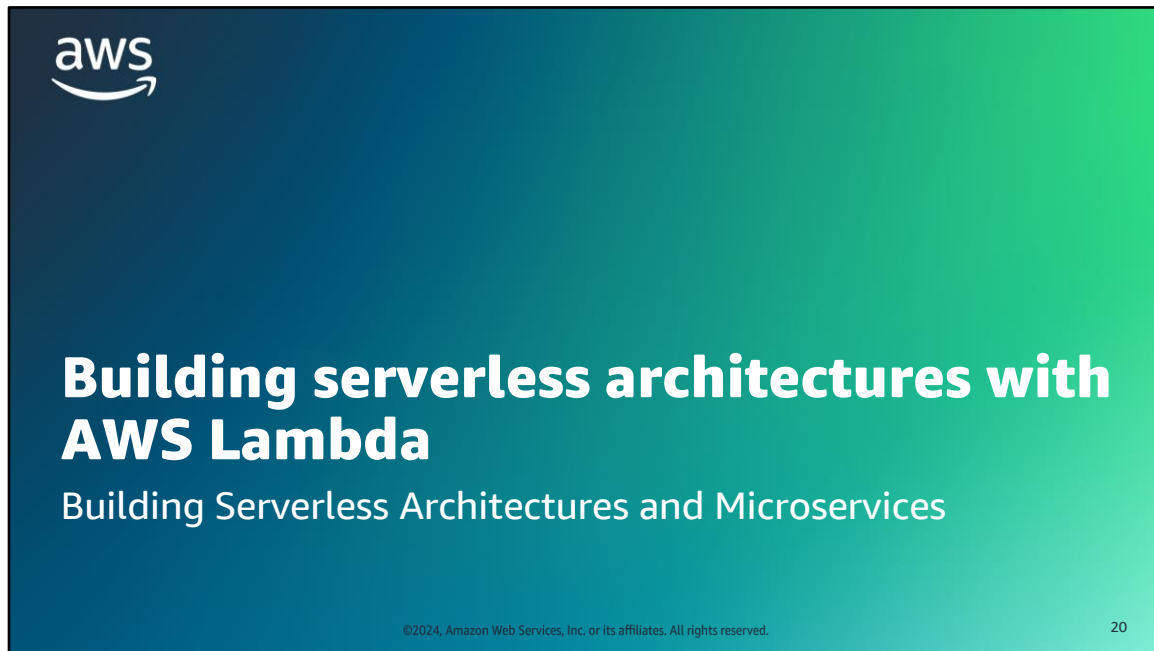


- Microservice applications are composed of autonomous, specialized services.
- Microservices have the following benefits:
  - Team agility
  - Reusable code
  - Flexible scaling
  - Technological freedom
  - Resilience
  - Simplified deployment
- Microservices can form part of a three-tier architecture operating in the app and data tiers.

©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

19





This section reviews serverless architectures in AWS.

## Operational task comparison

Operational tasks	Server in a VPC	Serverless
Configure an instance	Yes	No
Update operating system (OS)	Yes	No
Install application platform	Yes	No
Build and deploy apps	Yes	Yes
Configure automatic scaling and load balancing	Yes	No
Continuously secure and monitor instances	Yes	No
Monitor and maintain apps	Yes	Yes




©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

21

One of the major benefits of cloud computing is its ability to hide the infrastructure layer. This ability eliminates the need to manually manage the underlying physical hardware. In a serverless environment, this abstraction allows you to focus on the code for your applications without spending time building and maintaining the underlying infrastructure. With serverless applications, there are never instances, operating systems, or servers to manage. AWS handles everything required to run and scale your application. By building serverless applications, your developers can focus on the code that makes your business unique.

The chart above compares the operational tasks in a virtual private cloud (VPC) server environment to those in a serverless environment. The serverless approach to development reduces overhead and lets you focus, experiment, and innovate faster. The only serverless compute operational tasks you have to take care of in a serverless environment are to build and deploy applications and then monitor and maintain those applications in a production environment.

## AWS Lambda



Lambda

- AWS Lambda lets you run code functions without provisioning or managing servers.
- Lambda functions are configurable with runtime language, amount of memory, and timeout duration.
- A function has a maximum duration of 15 minutes.
- Functions can be deployed as .zip files or container images.

aws

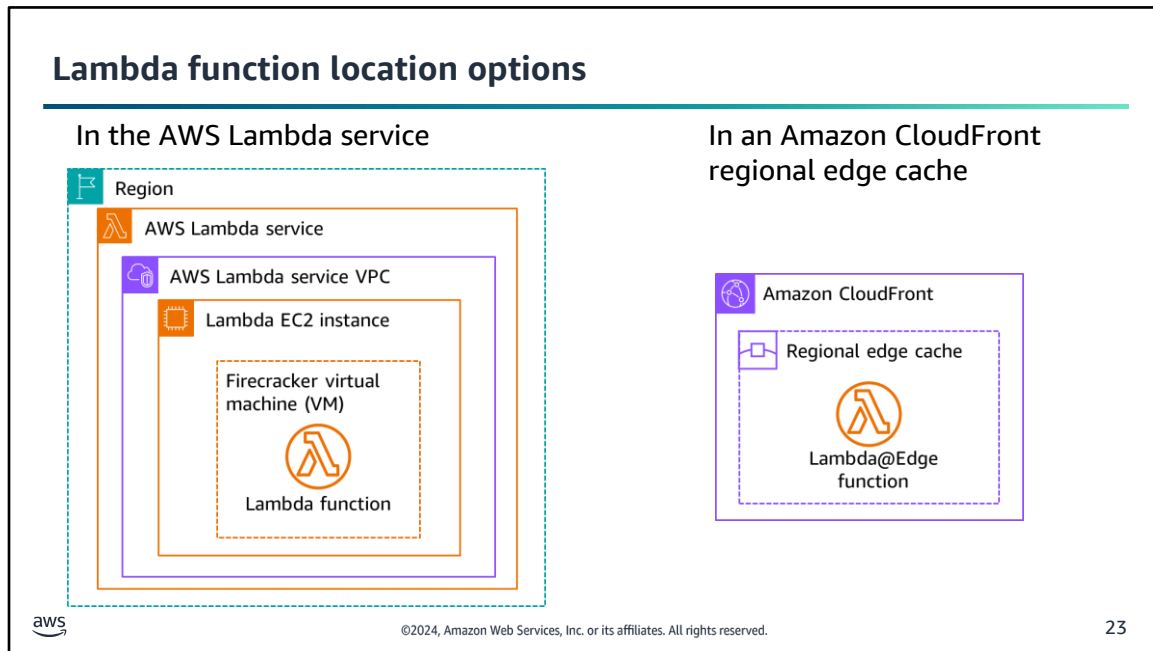
©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

22

AWS Lambda is a compute service that lets you run code without provisioning or managing servers. Lambda runs your code on a high-availability compute infrastructure and performs all of the administration of the compute resources, including server and operating system maintenance, capacity provisioning, automatic scaling, and logging.

You configure your Lambda function with the runtime language you prefer, the amount of memory that the function needs, and the maximum length of function timeout. The amount of memory determines the amount of virtual CPU and network bandwidth. Currently, the minimum and maximum amount of memory that can be allocated is 128MB and 10,240MB respectively. A Lambda function can't exceed 15 minutes in duration, so 15 minutes is the maximum timeout setting. This is an AWS hard limit and can't be changed.

You create code for the function and upload the code using a deployment package. Lambda supports two types of deployment packages: container images and .zip file archives. The Lambda service invokes the function when an event occurs. Lambda runs multiple instances of your function in parallel, governed by concurrency and scaling limits. You only pay for the compute time that you consume—there is no charge when your code isn't running.



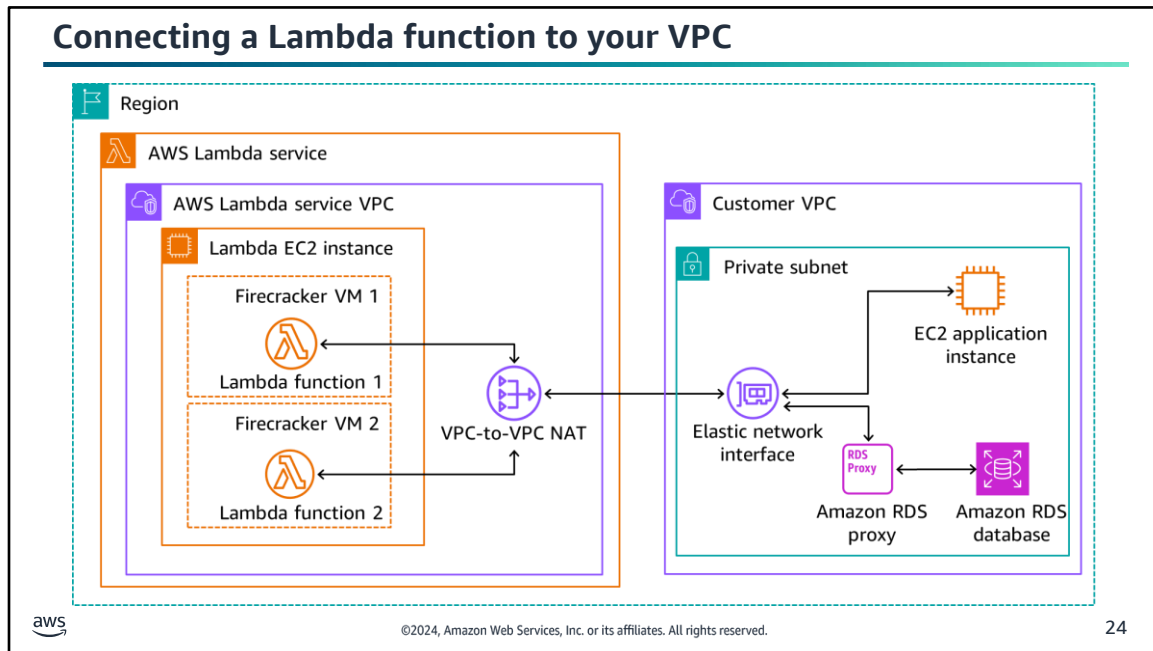
23

A Lambda function can run inside a VPC owned by the AWS Lambda service or in an Amazon CloudFront regional cache.

When you create a Lambda function, you deploy it to the AWS Region where you want your Lambda function to run. When a Lambda function is invoked, the Lambda service instantiates an isolated Firecracker virtual machine (VM) on an Amazon Elastic Compute Cloud (Amazon EC2) instance in the Lambda service VPC. Firecracker was developed at Amazon Web Services (AWS) and provides the ability to create a microVM in under a second. After the function runs, the VM is available for the next request to run. The Lambda service will instantiate as many runtime VM environments as the function configuration allows. The Lambda service applies network access and security rules to the Lambda service VPC, and Lambda maintains and monitors the VPC automatically.

Lambda@Edge is an extension of AWS Lambda, a compute service that lets you run functions that customize the content that CloudFront delivers. You can author Node.js or Python functions in one Region, US East (N. Virginia), and then run them in AWS regional edge locations globally that are closer to the viewer. Processing requests at AWS locations closer to the viewer instead of on origin servers significantly reduces latency and improves the user experience. An example of a Lambda@Edge use case is a retail website that sells clothing. If you use cookies to indicate which color a user chose for a jacket, a Lambda function can change the request so that CloudFront returns the image of a jacket in the selected color.

To extend functions to CloudFront edge locations, you can use CloudFront functions. Please note that CloudFront functions have no network access and have significantly smaller maximum execution time and total package size limits.



24

Sometimes you might have a requirement to implement an architecture that has serverless components and components running in your own VPC. When designing this type of architecture, pay close attention to scaling as some components can cause a bottleneck.

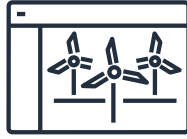


By default, a Lambda function isn't connected to VPCs in your account. If your Lambda function needs to access the resources in your account VPC, you can configure the function to connect to your VPC. The Lambda service provides managed resources named Hyperplane elastic network interfaces (ENIs) which are created when the function is configured to connect to a VPC. When invoked, the Lambda function in the Lambda VPC connects to an ENI in your account VPC. Hyperplane ENIs provide NAT capabilities from the Lambda VPC to your account VPC using VPC-to-VPC NAT (V2N). V2N provides connectivity from the Lambda VPC to your account VPC, but it doesn't in the other direction.

When you connect a function to a VPC in your account, the function can't access the internet unless your VPC provides access. To give your function access to the internet, route outbound traffic to a NAT gateway in a public subnet. The NAT gateway has a public IP address and can connect to the internet through the VPC's internet gateway.


In the example above, the database and the Amazon EC2 application instance can cause bottlenecks if the Lambda functions aggressively scale. Lambda functions 1 and 2 connect to an EC2 application instance and an Amazon Relational Database Service (Amazon RDS) proxy deployed in a private subnet in the customer's VPC using the VPC-to-VPC NAT and the ENI. To scale the EC2 instance, deploy it behind an application load balancer in an Amazon EC2 Auto Scaling group.

To scale the database, you can use Amazon RDS proxy that manages a connection pool to the Amazon RDS database. Because Lambda functions can scale rapidly, the connections to an Amazon RDS database can be saturated. Lambda functions that rapidly open and close database connections can cause the database to fall behind. When no more connections are available the function will produce an error. When using MySQL and Aurora Amazon RDS databases, you can solve this challenge with RDS proxy. The Lambda functions connect to RDS proxy, which will have an open connection to the database ready to be used.

## Identifying Lambda serverless scenarios



Synchronous processing	Asynchronous processing	Streaming processing
<ul style="list-style-type: none"><li>• Web apps</li><li>• Web services</li><li>• Microservices</li><li>• Machine learning inferences</li></ul>	<ul style="list-style-type: none"><li>• Scheduled events</li><li>• Queued messages</li><li>• Image or video transformation</li><li>• AWS service triggers</li></ul>	Streaming applications

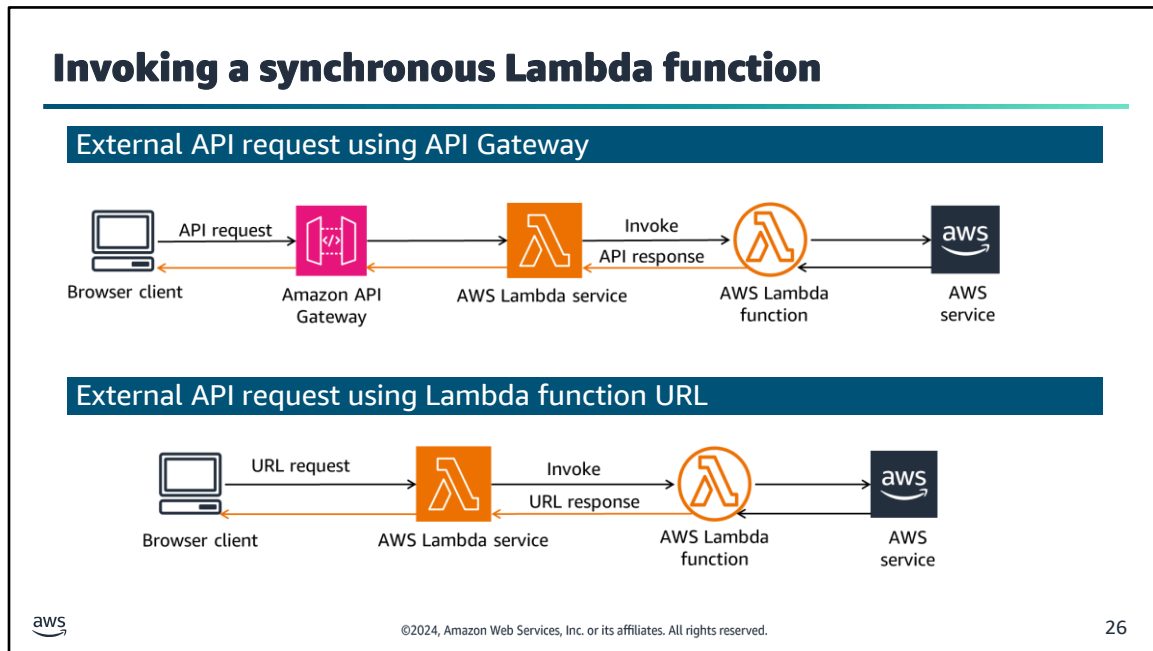
 ©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved. 25

Lambda functions are suitable to use for synchronous, asynchronous, and streaming processing.

Synchronous processing is when a requestor makes a request and expects a response within a certain time period. One of the most common Lambda synchronous patterns are website and mobile application microservices. For a website, you can invoke your Lambda function using Amazon API Gateway. For mobile applications, you can process clicks within your application by invoking a Lambda function URL. An example of a synchronous process is to add an item to your shopping cart while browsing on an ecommerce website. The Lambda function can update the shopping cart with the item selected.

Asynchronous processing is when a requestor makes a request to offload the process at a later time. Lambda functions can be used asynchronously as it has built in capability to handle inputs from queues and event schedulers. An example of an asynchronous process would be delivering a package to an online shopper. Lambda functions can be used to update the status of the package by processing messages in a queue. Other examples are file processing and image identification.

Streaming processing is required when data arrives in a continuous stream. The Lambda service has built in capability to batch the stream data and invoke Lambda functions to process a batch. Lambda functions are often used for data analytics to format data into a different format or aggregate raw data. An example of a streaming process is Amazon DynamoDB Streams to record changes in a DynamoDB table and forwarding it to a Lambda function to aggregate.



When you invoke a function synchronously, the Lambda service runs the function and waits for a response. In this example, an API request is received from a browser client to Amazon API Gateway. API Gateway invokes the Lambda function by calling the AWS Lambda service. While the function runs, it can optionally call other AWS services. When the function completes, the Lambda service returns the response from the function's code with additional data, such as the version of the function that was invoked. If the function encounters an error, the error is returned as the response. The Lambda service returns the response to API Gateway which in turn sends the response to the browser client.

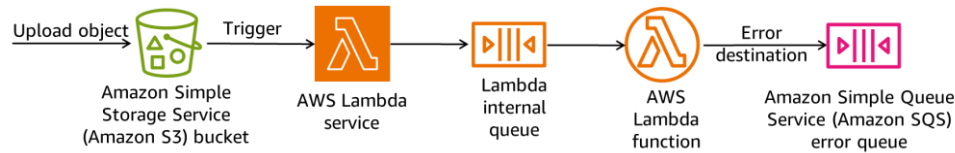
Another option for a synchronous call to Lambda is to use a function URL. A function URL is a dedicated HTTP(S) endpoint for your Lambda function. In the example above, the URL request is sent directly to the Lambda service from a browser client. The Lambda service invokes the Lambda function which returns a URL response. The response is returned to the browser client.

When you create a function URL, Lambda automatically generates a unique URL endpoint for you. After you create a function URL, its URL endpoint never changes. Lambda function URLs use resource-based policies for security and access control. Function URLs also support cross-origin resource sharing (CORS) configuration options. CORS defines a way for client web applications that are loaded in one domain to interact with resources in a different domain.

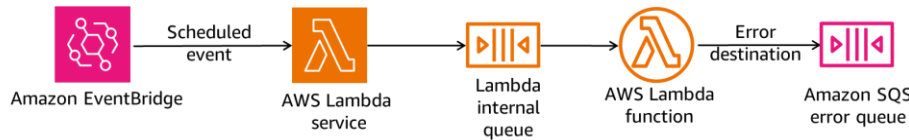
While function URLs are simple and easy to set up, the browser application will require an update if the URL changes. When a function changes behind Amazon API Gateway, no changes are required to the browser application.

## Invoking an asynchronous Lambda function

### AWS service trigger event



### Scheduled event



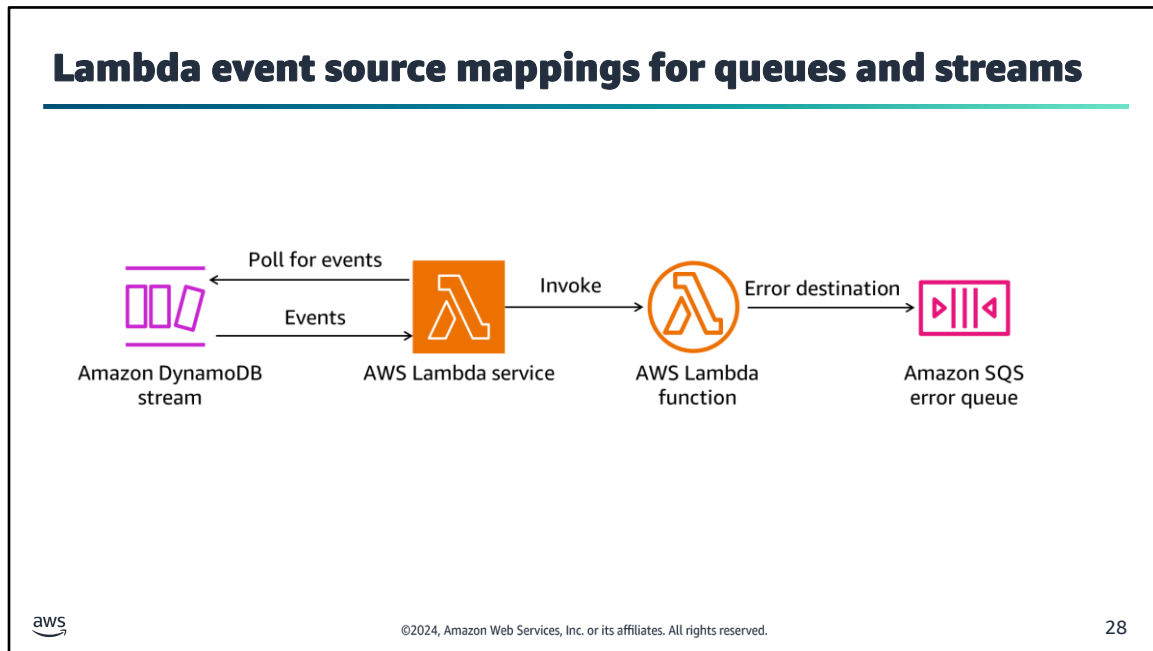
©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

27

When you invoke a function asynchronously, you don't wait for a response from the function code. You hand off the event to Lambda, and Lambda handles the rest. The Lambda service places the event in a queue and returns a success response without additional information. A separate process reads events from the queue and sends them to the function. You can configure how Lambda handles errors and can send invocation records to a downstream resource such as Amazon Simple Queue Service (Amazon SQS) or Amazon EventBridge to chain together components of your application.

Several AWS services, such as Amazon Simple Storage Service (Amazon S3) and Amazon Simple Notification Service (Amazon SNS), invoke functions asynchronously to process events. If scheduled events are required, you can use EventBridge as a scheduler to invoke Lambda functions. If an AWS service doesn't have a direct integration with the Lambda service, EventBridge can serve as the event bus to route the request. An example of a scheduled event is daily reporting or any recurring process that should be activated.





An event source mapping is a Lambda resource that reads from an event source and invokes a Lambda function. You can use event source mappings to process items from a stream or queue in services that don't invoke Lambda functions directly. The Lambda service can poll AWS services like Amazon DynamoDB, Amazon Kinesis, Amazon SQS, and Amazon DocumentDB (with MongoDB compatibility).

In the above diagram, the target event source is a DynamoDB stream. When the Lambda service polls the DynamoDB stream for events, a number of events are returned to the Lambda service. The Lambda service batches records together in a single payload and invokes the Lambda function with the payload. You can configure the Lambda function to specify the maximum size of the batch window and the payload. The payload can't exceed the Lambda function input limit of 6 MB.

An example scenario would be if a customer order changed status to delivery in progress in the DynamoDB orders table, then the Lambda function is invoked to send a notification to the customer and do some financial processing.

## Python Lambda function handler

```
1  import json
2  def lambda_handler(event, context):
3      length=event['length']
4      width=event['width']
5      area = calculate_area(length, width)
6      data = {"area": area}
7      return json.dumps(data)
8
9  def calculate_area(length, width):
10     return length*width
```

Line 2: Event object is a JSON document containing input data and invoking service data

Line 2: Context object provides methods and properties about function runtime and invocation

Line 7: json.dumps(data) returns result as a JSON document

Lines 8 and 9: Business logic method

aws

©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

29

The Lambda function handler is the entry point in your function code that processes events. When your function is invoked, Lambda runs the handler method. Your function runs until the handler returns a response, exits, or times out. The Lambda function handler name specified at the time that you create a Lambda function is derived from the name of the file where the Lambda handler function is located plus the name of the Python handler function. A function handler can be any name, however, the default name in the Lambda console is `lambda_function.lambda_handler`. This function handler name reflects the function name (`lambda_handler`) and the file where the handler code is stored (`lambda_function.py`).

When Lambda invokes your function handler, the Lambda runtime passes two arguments to the function handler: the event and the context objects. The first argument is the event object. An event is a JSON-formatted document that contains data for a Lambda function to process. The Lambda runtime converts the event to an object and passes it to your function code. The event object contains information from the invoking service. The second argument is the context object. A context object is passed to your function by Lambda at runtime. This object provides methods and properties that provide information about the invocation, function, and runtime environment.

In the above example, the code runs in the following way:

Line 1: The Python `json` library module is imported for the function to use.

Line 2: This is the entry point method for the Lambda function and receives the event and context objects parameters from the Lambda service.

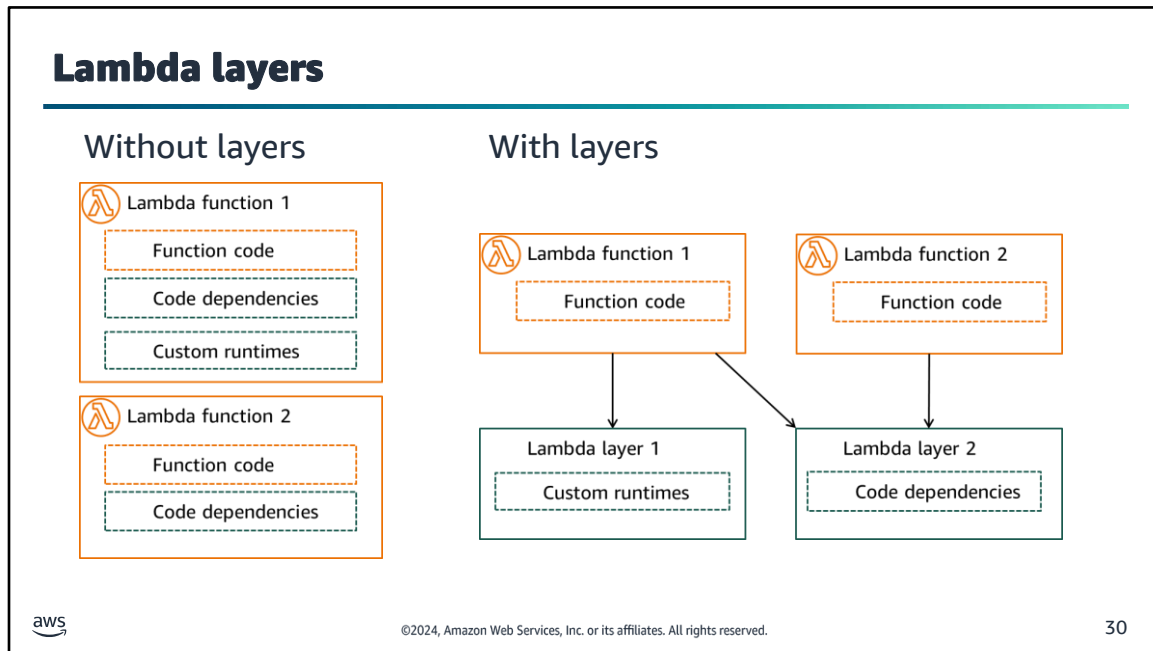
Lines 3 and 4: The `length` and `width` local function variables are extracted from the event object.

Line 5: The business logic method `calculate_area` is called with the `length` and `width` parameters. It's a best practice to keep the handler function small and put the business logic in separate methods. This reduces the handler function load times.

Lines 8 and 9: The `calculate_area` method runs returning the area result.

Lines 6 and 7: The area variable is converted to a JSON string and returned with the JSON library `dumps` method.

To get on demand code recommendations, use Amazon Q Developer in the AWS Management Console or as a plug-in for your preferred integrated development environment (IDE).



30

**Without layers image description:** Two Lambda functions (function 1 and function 2) each with their own function code and dependencies. Function 1 also includes a custom run time. **End description.**

**With layers image description:** Lambda function 1 and Lambda function 2 include only function code. Lambda function one uses layer 1 for its custom runtime and layer 2 for its code dependencies. Lambda function 2 uses layer 2 for its code dependencies. **End description.**

A Lambda layer is a .zip file archive that contains supplementary code or data. Layers usually contain library dependencies, a custom runtime, or configuration files. You can also package your own custom runtime in a Lambda layer if you prefer a different runtime from those provided by the Lambda service.

There are multiple reasons why you might consider using layers:


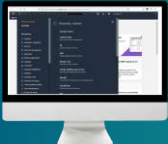
**Reduce the size of your deployment packages:** Instead of including all of your function dependencies along with your function code in your deployment package, put them in a layer. This keeps deployment packages small and organized.

**Separate core function logic from dependencies:** With layers, you can update your function dependencies independent of your function code, and vice versa. This promotes separation of concerns and helps you focus on your function logic.

**Share dependencies across multiple functions:** After you create a layer, you can apply it to any number of functions in your account. Without layers, you need to include the same dependencies in each individual deployment package.

**Use the Lambda console code editor:** The code editor is a useful tool for testing minor function code updates quickly. However, you can't use the editor if your deployment package size is too large. Using layers reduces your package size and can unlock usage of the code editor.

## Demo: Using AWS Lambda with Amazon S3



- This demonstration uses AWS Lambda and Amazon S3.
- In this demonstration, you will see how to do the following:
  - Use AWS Lambda with Amazon S3 to upload an object.
  - Create a thumbnail for the object.

©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

31

Find this recorded demo in your online course as part of this module.

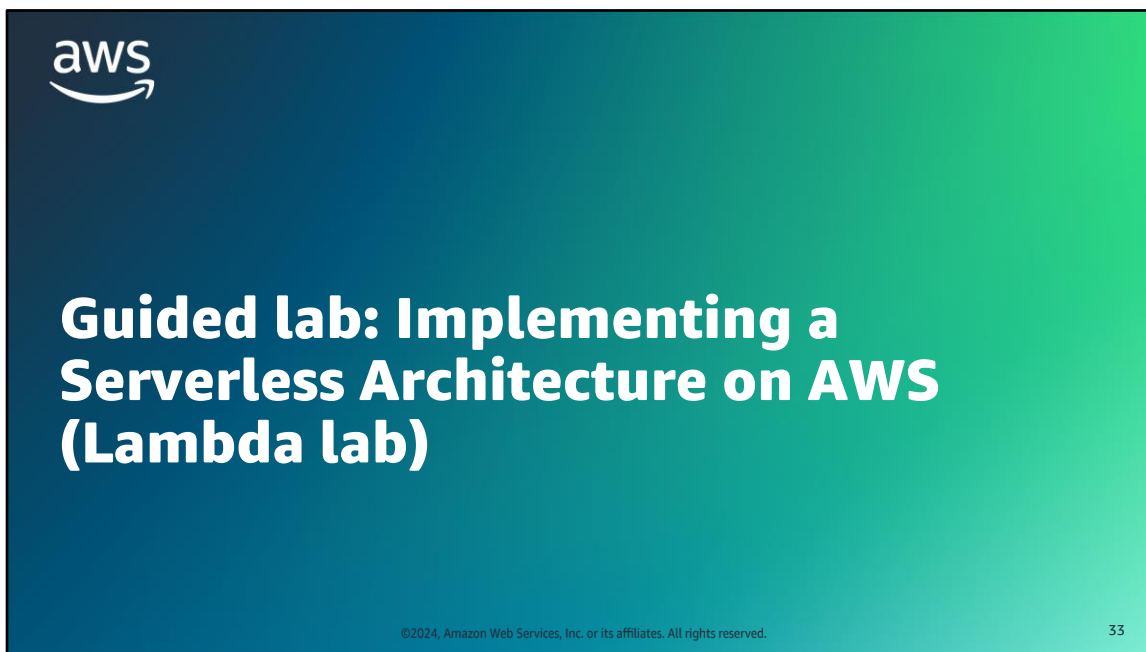
## Key takeaways: Building serverless architectures with AWS Lambda



- AWS Lambda is a service to run code functions without provisioning or managing servers.
- A Lambda function can run inside a VPC owned by the AWS Lambda service or as Lambda@Edge in an Amazon CloudFront regional cache.
- A Lambda function can be configured to connect to your VPC to access AWS services inside the VPC.
- A Lambda function can be invoked synchronously, asynchronously, and with event source mappings for queues and streams.
- Use Lambda layers to package code dependencies or custom runtimes to be re-used by all Lambda functions in the Region.



©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

32



This is an optional lab. The next slides summarize what you will do in the lab, and you will find the detailed instructions in the lab environment.

## Lambda lab tasks:



- In this lab, you will do the following:
  - Implement a serverless architecture on AWS.
  - Trigger Lambda functions from Amazon S3 and Amazon DynamoDB.
  - Configure Amazon Simple Notification Service (Amazon SNS) to send notifications.
- Open your lab environment to start the lab and find additional details about the tasks that you will perform.

©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

34

Access the lab environment through your online course to get additional details and complete the lab.

## Debrief: Lambda lab

---

- When you created your Load-Inventory function, you gave the function a role. What does a role do for a function?
- In this lab, you created multiple Lambda functions. What is the benefit of using multiple functions rather than combining multiple tasks into a single function?








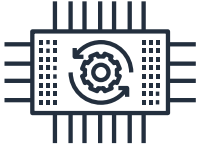
This section looks at building microservice applications with AWS container services.

## Choosing containers over AWS Lambda functions




**Longer than 15 minutes**

If a service runs longer than 15 minutes, it isn't suitable for Lambda functions.



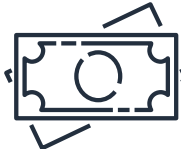
**Memory intensive applications**

Workloads that use more than 10 GB of memory are not suitable for Lambda functions.




**Migrating legacy containers**

Containers can help migrate legacy applications running on-premises or on EC2 instances without refactoring or rearchitecting them.



**Cost**

- Containers can be continuously run at a fixed cost.
- The price to run a Lambda function increases with the number of invocations, duration, and memory allocated.



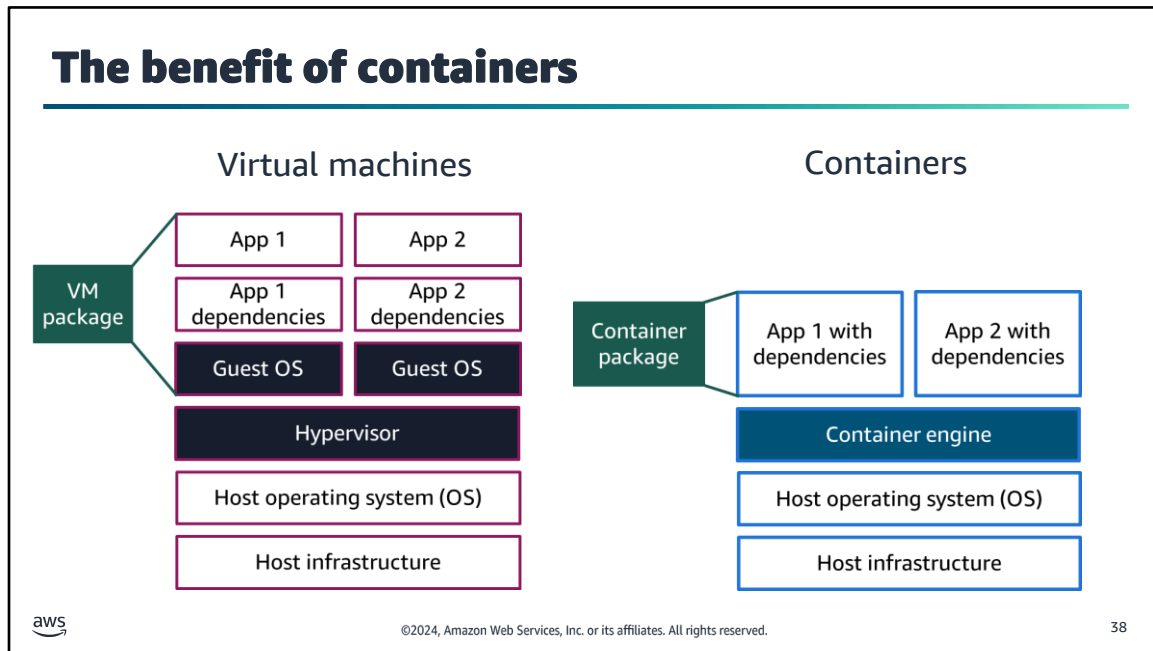
©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

37

Although AWS Lambda is an excellent solution for microservices, not all microservice applications are suited to be built using Lambda functions. If a microservice process takes longer than 15 minutes to run or uses more than 10 GB of memory, then an alternative to Lambda functions must be found. Containers can be sized to accommodate applications exceeding the Lambda function limitations. If you need a different runtime than Lambda provides, running it in a container could be a simpler solution than porting it to comply with the Lambda runtime.

Legacy applications, where refactoring the code into microservices is not an option due to time or practical constraints, are usually good candidates for containers.

AWS Lambda functions can become pricey if a function runs for an extended period and uses a lot of memory. For example, a Lambda function with 3 million requests a month running on average for 120 ms with 1536 MB of memory on an Arm processor would come to \$7.80 per month if no free tier savings were applied. If the memory and run period were increased to 5 GB and 5000 ms respectively, the cost increases to \$1000.60 per month with no free tier savings applied. In contrast, running five AWS Fargate containers continuously with 5 GB memory costs \$180.65. If an application receives requests continuously and has very little idle time, there is a number of transactions per second price point where containers would be more cost effective than Lambda functions. Please note that these price amounts are illustrative and might not match current prices.

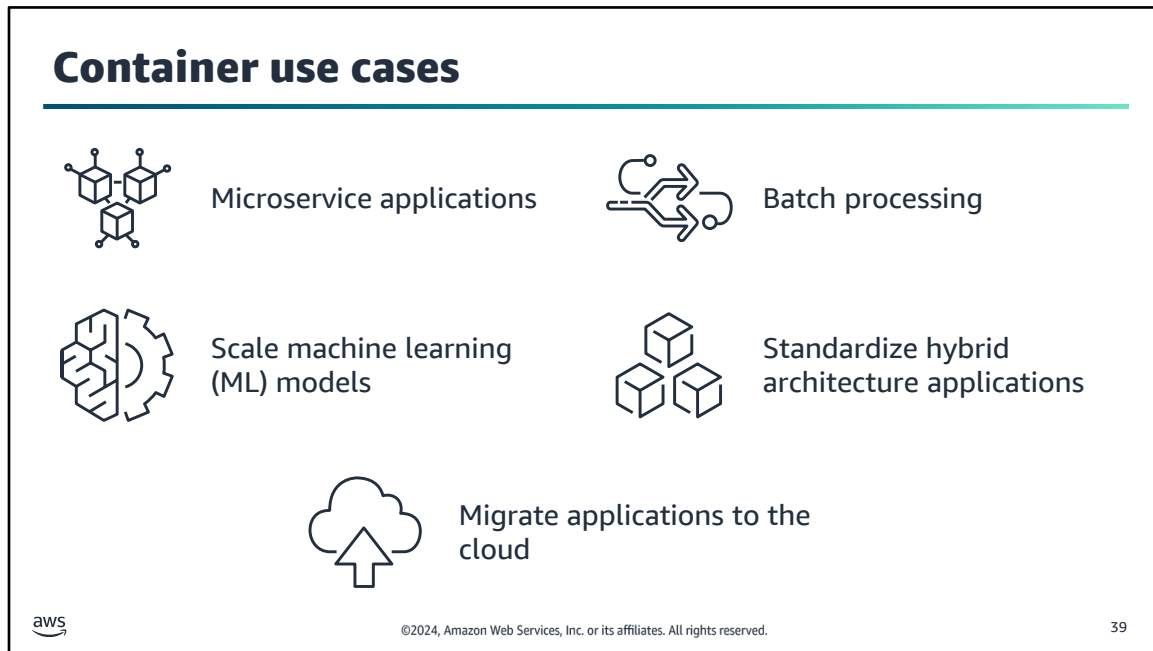


Virtual machine (VM) architecture was designed to accommodate multiple applications running on different guest operating systems (OS). Each VM package consists of the app code, app dependencies, and a guest OS layer, which means that they are relatively large. VM packages are deployed on the hypervisor that allocates host resources to each application.

Containers are standardized application code packages that contain the code and the code dependencies. They run on top of the container engine and only have read access to parts of the host OS. Container packages are notably smaller than VM packages. As a result, containers are lightweight and can rapidly start running and can scale quickly to accommodate changing workloads. Containers are self contained and will run consistently regardless of the OS and hardware platform.

In essence, the container engine replaces the hypervisor and guest OS layers of a VM. Containers use less host system resources because they do not include the guest OS. This means you can run many more containers on a same sized host than VM applications.

Some of the most widely used container engines are Docker Engine, containerd, and Red Hat OpenShift.

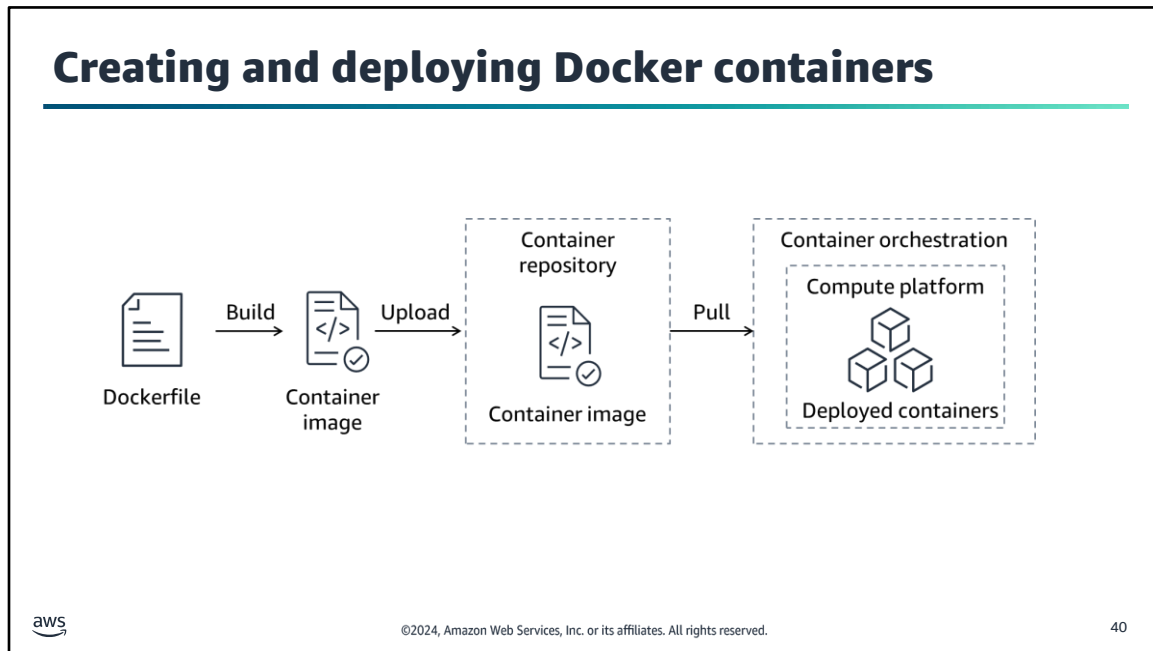


So what type of use cases are very well suited to containers? You can package a microservice as a container application which will run in an isolated process without affecting other microservices. Containers are also suitable for batch jobs that require prolonged processing. For example, an extract, transform, and load (ETL) job can run as long as the transformation takes and can be scaled dynamically to respond to demand changes.

AWS makes use of containers for machine learning (ML) services, such as Amazon SageMaker, to deploy packaged algorithms to train and deploy ML models. AWS customers can also use containers to scale and run ML models to run close to the large datasets needed to train the models.

If you have architecture that spans the AWS cloud and on-premises, you can use containers to standardize the application code and deployment packages to lighten the maintenance and administration load for the separate environments.

If you require applications to rapidly move from on-premises to the cloud, containers can be used to package applications and deploy them in the cloud without making any code changes.

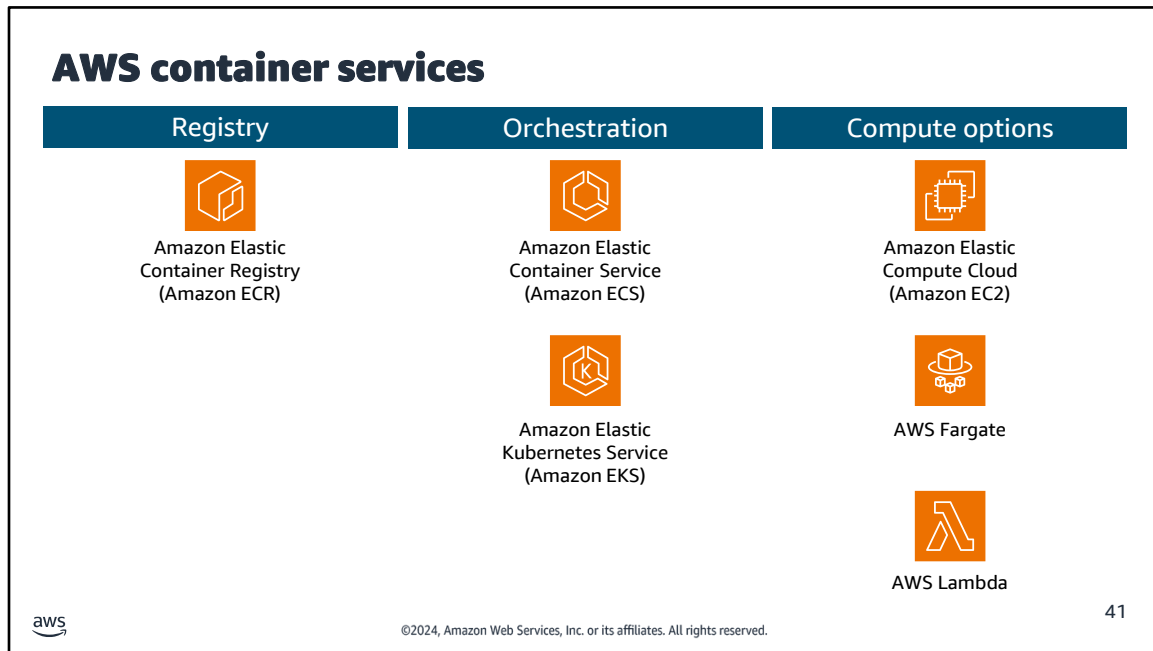


Docker Engine is widely used in AWS as a container engine. The first step in creating and deploying a Docker container is to create a Dockerfile. The Dockerfile is a text file that contains build instructions on how to build the container image. It specifies the container environment and code that runs inside the environment. The next step is to build the container image by running a container engine build command.

After a successful build and local host testing and verification, the container image is uploaded in a container repository. The container image is now ready for deployment. Typically, multiple containers will be deployed from a single image.

Although you can manually deploy containers, companies prefer to use container orchestration services to manage, deploy, and scale hundreds or thousands of containers. The orchestration service pulls the required container image from the container repository and deploys the number of required containers on the compute platform. Container orchestration services can use one compute platform or multiple types of compute platforms to deploy and manage containers. The orchestration service will keep track of the health of the compute platform and containers.

Container specifications are governed by the Open Container Initiative (OCI) which provides the industry standard for container image, runtime, and distribution specifications. This means that containers can be ported to different platforms and environments without changes.



AWS has a number of services that run and support containers. At the core of these are services to manage container registry, orchestration, and compute options.

The Amazon Elastic Container Registry (Amazon ECR) is an AWS managed container image registry service that is secure, scalable, and reliable. Amazon ECR supports private and public repositories. Private repositories have resource-based permissions using AWS Identity and Access Management (IAM) to allow only specific users or Amazon Elastic Compute Cloud (Amazon EC2) instances access to your container images.

AWS has two container orchestrations options to manage container clusters. Container orchestrations provide a simplified interface to create and maintain container clusters. Provisioning the cluster is usually done with a single API call and configuration files.

Amazon Elastic Container Service (Amazon ECS) was developed by AWS and comes with AWS configuration and operational best practices built-in. It's integrated with both AWS and third-party tools, such as Amazon ECR and Docker. This integration accelerates the development time of teams to focus on building applications and not the environment.





The other orchestration option is to use Amazon Elastic Kubernetes Service (Amazon EKS) which uses the Kubernetes system to manage containers. Kubernetes is an open-source system that automates the management, scaling, and deployment of containerized applications. Amazon EKS is suitable for AWS customers who run Kubernetes clusters on-premises that want the same orchestration tools in the cloud.


AWS customers can also opt to run different container orchestration services by using Amazon EC2 instances and installing and managing the orchestration service in the same way as they would on-premises.

Both Amazon ECS and Amazon EKS clusters can create compute nodes using Amazon EC2 instances or use AWS Fargate nodes. Alternatively, a cluster can have both Amazon EC2 instance nodes and Fargate nodes. While EC2 instance nodes run inside the customer VPC, Fargate nodes are managed by AWS in the AWS Fargate virtual private cloud (VPC). Amazon ECS and Amazon EKS each have an Anywhere feature to deploy clusters in an on-premises environment.

AWS Lambda can also be used to launch containers using up to 10 GB of memory. AWS Lambda functions aren't managed by a container orchestration service. Be sure to do a cost comparison to select the most cost effective solution when choosing between AWS Lambda, Amazon ECS, AWS Fargate, or Amazon EKS.

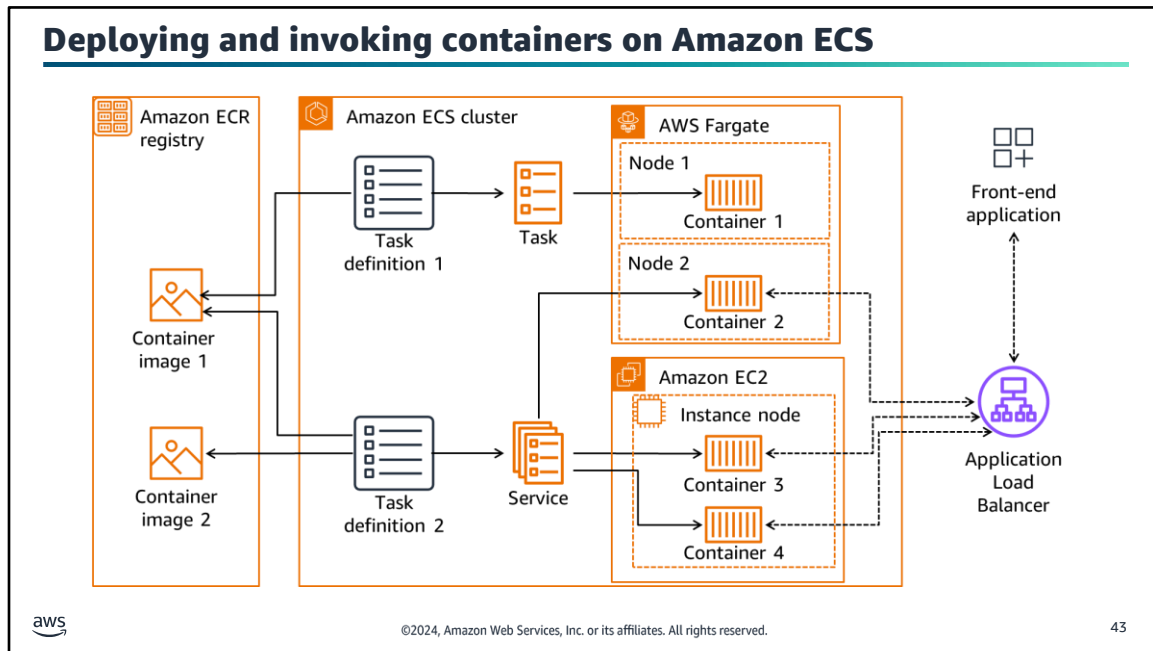
## Benefits of AWS Fargate

			
<b>No cluster or server management</b> <ul style="list-style-type: none"><li>• No server provisioning or maintenance.</li><li>• No cluster packing optimization.</li></ul>	<b>Per second billing</b> <p>Only pay for what you use.</p>	<b>Automatic scaling</b> <p>Dynamically scale tasks based on CPU, memory, or other metrics.</p>	<b>Suitable for teams new to containers</b> <p>AWS Fargate usage doesn't require comprehensive knowledge of container technology.</p>

 ©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved. 42

AWS Fargate lets you run containers without the overhead of managing servers or container clusters. With AWS Fargate, you don't have to provision, configure, or scale clusters of virtual machines to run containers. This removes the need to choose server types, decide when to scale your clusters, or optimize cluster packing for efficiency. AWS Fargate provides per second billing so that you can use compute as you need it. This ensures that no unused memory or CPU resources are consumed. Automatic scaling is the ability to increase or decrease the desired count of tasks in your Amazon ECS service automatically. You can control AWS Fargate scaling with Amazon ECS capacity providers. The capacity provider decides how tasks are spread across the cluster's capacity providers. AWS Fargate has scaling and security built in, which makes it suitable for teams that are just getting started with container technology and are unfamiliar with cluster processes.





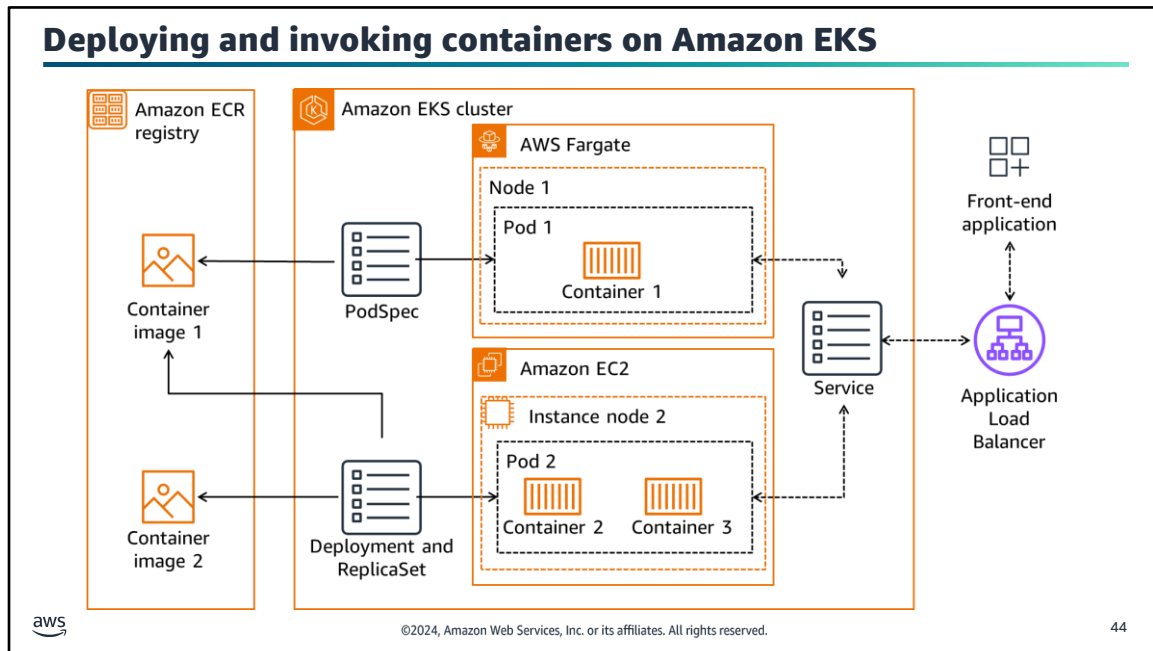
Before you can deploy a container using Amazon ECS, you have to create an ECS cluster. An ECS cluster consists of compute nodes on a compute platform such as AWS Fargate and Amazon EC2.

To deploy containers, you create an ECS task definition as a blueprint for your application. Tasks are the smallest unit of compute in Amazon ECS and provide a set of containers that you want to place together, their properties, and how they might be linked. Tasks include all the information that Amazon ECS needs to make the placement decision of which host to use to deploy containers. To launch a single container, your task definition should only include one container definition. It's a text file in JSON format that describes the parameters and one or more containers that form your application. Some of the parameters that you can specify in a task definition are the container image for each container in the task, the task or container CPU and memory, and the task launch type to determine the compute service.

After you create a task definition, you can run the task definition as a task or as a service. When a task is run, it results in a number of deployed containers within a cluster. Amazon ECS can manage multiple tasks as a service. An Amazon ECS service runs and maintains your desired number of tasks simultaneously in an Amazon ECS cluster to ensure high availability. If any of your tasks fail or stop for any reason, the Amazon ECS service launches another instance based on your task definition to replace the failed task.

For example, when a front-end application wants to use the container service, an Application Load Balancer can be configured in a service to distribute requests to containers. An Application Load Balancer makes routing decisions at the application layer (HTTP/HTTPS), supports path-based routing, and can route requests to one or more ports on each container instance in your cluster.

In the example above, task definition 1 specifies container image 1, which will run as a task deploying Fargate container 1 on node 1. Task definition 2, on the other hand, specifies container image 1 and container image 2. It will be deployed as service deploying Fargate container 2 on Fargate node 2 and EC2 instance node containers 3 and 4 on the EC2 instance node. Service 2 also specified an Application Load Balancer to invoke the service containers from front-end applications based on the URL path. For example, if the Login container implemented a Login microservice, then the Application Load Balancer can have a listener rule forwarding the /api/login path to the Login microservice.



Before you can deploy a container using Amazon EKS, you have to create an Amazon EKS cluster. The cluster consists of a set of worker machines called nodes. You can create an Amazon EKS cluster with AWS Fargate and Amazon EC2 instance nodes.

Kubernetes orchestrations use Pods as the smallest unit of deployment and don't interact directly with the container. A Pod can be used to deploy a single container or multiple containers that are dependent on one another. A single Pod can be deployed with a PodSpec, which specifies which container image to use. To deploy multiple highly available Pods, you can create a Deployment, which owns a ReplicaSet. The Deployment is responsible for keeping the number of Pods running as specified in the ReplicaSet.

To enable Pods to communicate or provide other applications to access to Pods, you can create a Kubernetes Service. A Service uses Pod labels to route network requests to available Pods with the same label.

In the example above, the PodSpec specifying container image 1 is used to deploy Pod 1 containing container 1. The Deployment specifies container image 1 and container image 2 and uses the ReplicaSet to deploy Pod 2. Pod 2 contains container 2 and container 3. Both Pod 1 and Pod 2 use the Service to accept and send request and responses from the front-end application.

## Choosing between Amazon ECS and Amazon EKS

Topic	Amazon ECS	Amazon EKS
Complexity	Simplifies cluster creation and maintenance	Provides more control over cluster using a complex interface
Scaling	Automated scaling based on demand	Manual configuration of autoscaling groups
Toolset	Amazon ECS toolset	Kubernetes toolset
Team experience	New to container cluster architecture	Familiar with Kubernetes cluster architecture and control processes



©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.


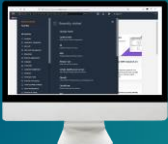
45

Amazon ECS delivers an AWS-opinionated solution for running containers at scale. It reduces the time it takes customers to build, deploy, or migrate their containerized applications successfully. Designed for simplicity from the start, using Amazon ECS decreases the number of decisions customers must make around compute, network, and security configurations, without sacrificing scale or features.

Amazon EKS provides the flexibility of Kubernetes with the security and resiliency of being an AWS managed service that's optimized for customers building highly available services. Amazon EKS provides a secure, reliable, scalable, and resilient Kubernetes environment for customers.

Choosing which service to use for container orchestration depends on the development and maintenance team requirements. If the team requires a simpler interface and less control over the cluster, then Amazon ECS would be the preferred service. It provides native AWS to AWS service integration and automated scaling based on demand. This benefits a team that's new to container cluster architecture and doesn't have Kubernetes experience.

## Demo: Running a Container



- This demonstration uses Amazon Elastic Container Service.
- In this demonstration, you will see how to do the following:
  - Run a sample application on an Amazon ECS cluster behind a load balancer.

©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

46

Find this recorded demo in your online course as part of this module.

## Key takeaways: Building microservice applications with AWS container services



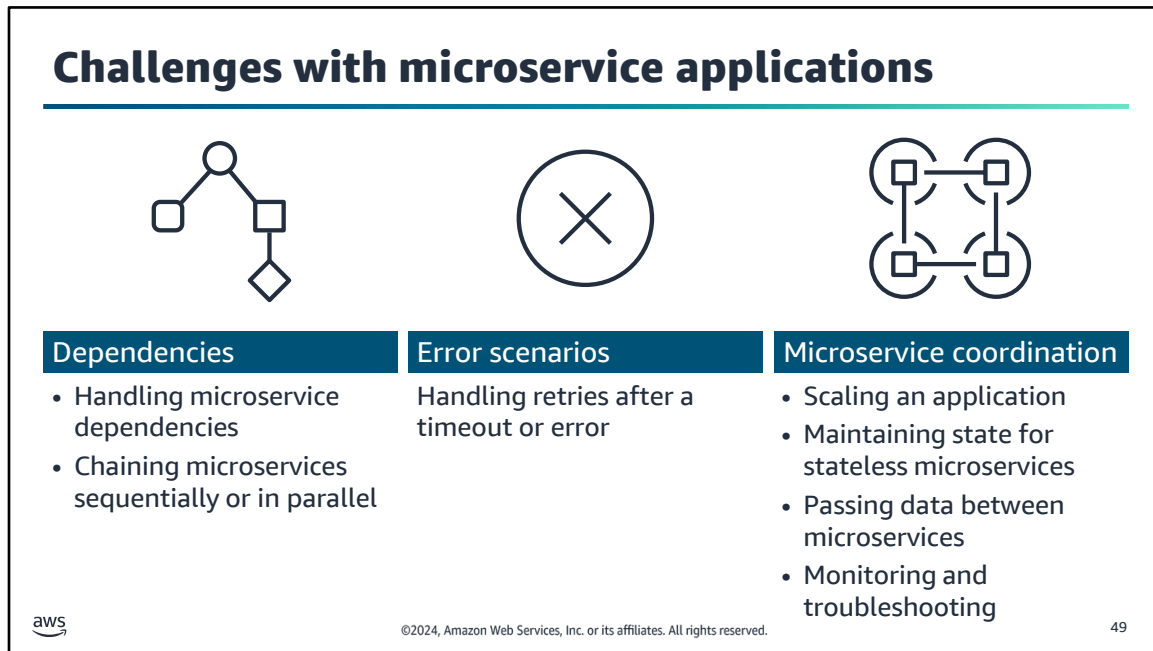
- Choose a container solution instead of Lambda functions when an application requires resources exceeding Lambda service limits.
- Amazon ECR provides a container image repository service.
- Amazon ECS is a container orchestration service with AWS toolkit.
- Amazon EKS is a container orchestration service with Kubernetes toolkit.
- AWS Fargate manages serverless cluster nodes and can be deployed in Amazon ECS and Amazon EKS clusters.

©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

47



This section looks at orchestrating services with AWS Step Functions.



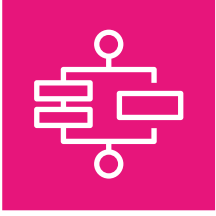
Implementing microservices are not without any challenges. Handling thousands of microservices that have dependencies on other microservices, handling error scenarios, and coordinating microservices can be difficult to do.

As your application grows in size, the number of components increases and many patterns of which tasks to run and in what order are possible. Consider a microservice application that's 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 application workloads and be able to 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's 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.

## AWS Step Functions



Step Functions



- Is a serverless orchestration service to manage workflows between multiple AWS services
- Has state machines (workflows) that contain a series of event-driven states (steps)
- Manages each workflow's state, checkpoints, and restarts
- Provides error handling capability
- Can transfer data between states
- States can filter and manipulate data

©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

50

AWS Step Functions is a serverless orchestration service that enables you to coordinate components of distributed applications and microservices by using visual workflows. With Step Functions, you can build an application's state machine (workflow) as a series of event-driven states (steps).

Step Functions provides a reliable way to coordinate components and go 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 efficiently. You can restart a workflow from the point of failure.

It is possible to pass input data into a state machine and get result data back as a synchronous response. State machines have the capability to pass data from one state to the next. A state can filter the data or add result data to the input data to pass on to the next state.

Step Functions state machines can be invoked from AWS services such as Amazon API Gateway, Amazon EventBridge, AWS Lambda, and other state machines running in Step Functions. You can use nested state machines to reduce the complexity of your main workflows and to reuse common processes.



## Standard or Express Workflow types

Topic	Standard Workflows	Express Workflows
Duration	Long-running	Short-running with no activities
Run metrics	Full history in the console	Results in CloudWatch Logs
Processing	Asynchronous	Synchronous and asynchronous
Run model	Exactly-once	Synchronous: At-least-once Asynchronous: At-most-once
State machine progress	Persisted on every state transition	No persisted state on every state transition
Pricing	Charged based on the number of state transitions	Charged based on the number of requests for workflow and its duration



©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

51

Step Functions has two workflow types. Standard Workflows have exactly-once workflow execution and can currently run for up to 1 year. This means that each step in a Standard Workflow will execute exactly once. Express Workflows, however, have at-least-once workflow execution for synchronous workflows and at-most-once for asynchronous workflows. Express Workflows can currently run for up to five minutes. This means that one or more steps in an Express Workflow can potentially run more than once.

Standard Workflows are ideal for long-running, auditable workflows because they show execution history and visual debugging. An example of a long-running workflow would be a customer order fulfillment which can take a number of days. Express Workflows are ideal for high-event-rate workloads, such as streaming data processing and IoT data ingestion.

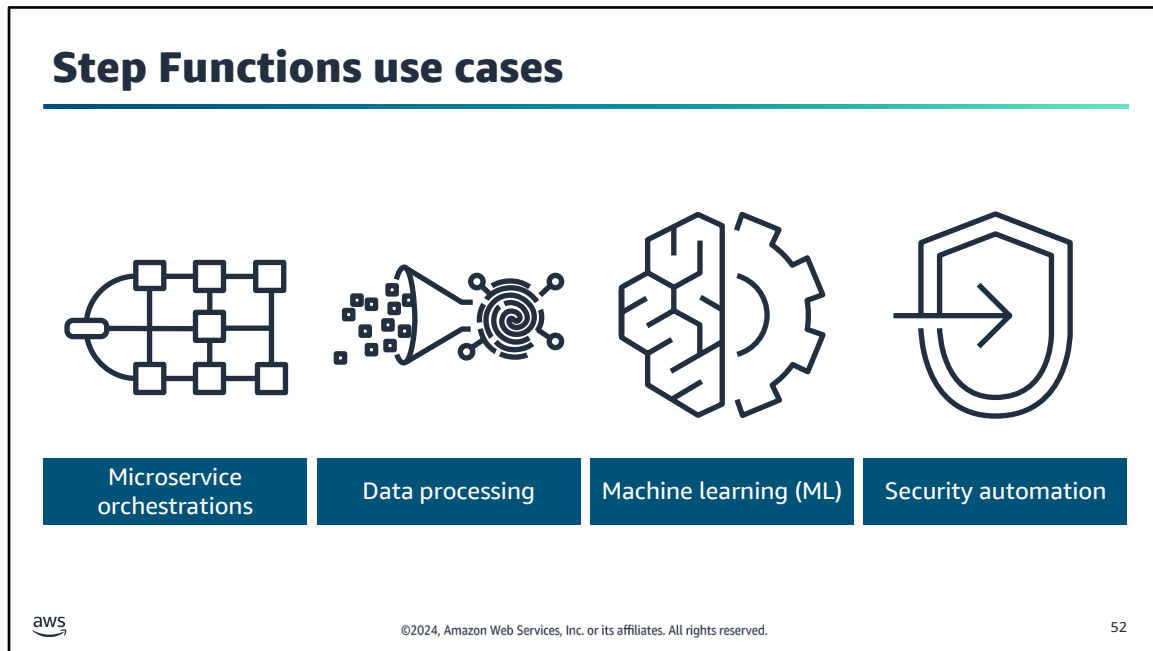
Asynchronous Express Workflows return confirmation that the workflow started, but don't wait for the workflow to complete. To get the result, you must poll the service's Amazon CloudWatch logs. You can use Asynchronous Express Workflows when you don't require immediate response output, such as messaging services or data processing that other services don't depend on.

Synchronous Express Workflows start a workflow, wait until it completes, and then return a result. This allows developers to quickly receive the workflow response without needing to poll additional services or write additional code. This is useful for high-volume microservice orchestration and fast compute tasks that communicate through HTTPS as used in most customer-facing applications. With Synchronous Express Workflows, you can develop applications without the need to develop additional code to handle errors, retries, or run parallel tasks.

Even when a production workflow might be best suited to Express Workflows, developers have found it helpful to debug the workflow using Standard Workflows. The reason is that Standard Workflows have a visual representation of errors and the execution history that provides easy-to-access details about each run or each step in the run. After you have the workflow working as desired, if the workload is more suitable to an Express Workflow, you can copy the Standard Workflow to an Express Workflow.

Standard Workflows are charged based on the number of state transitions that are required to execute your

application. Express Workflows are charged based on the number of requests for your workflow and its duration.



Step Functions state machines can be utilized whenever a workflow is needed. In AWS, we see customers using Step Functions frequently for microservice orchestrations, data processing, machine learning, and security automation.

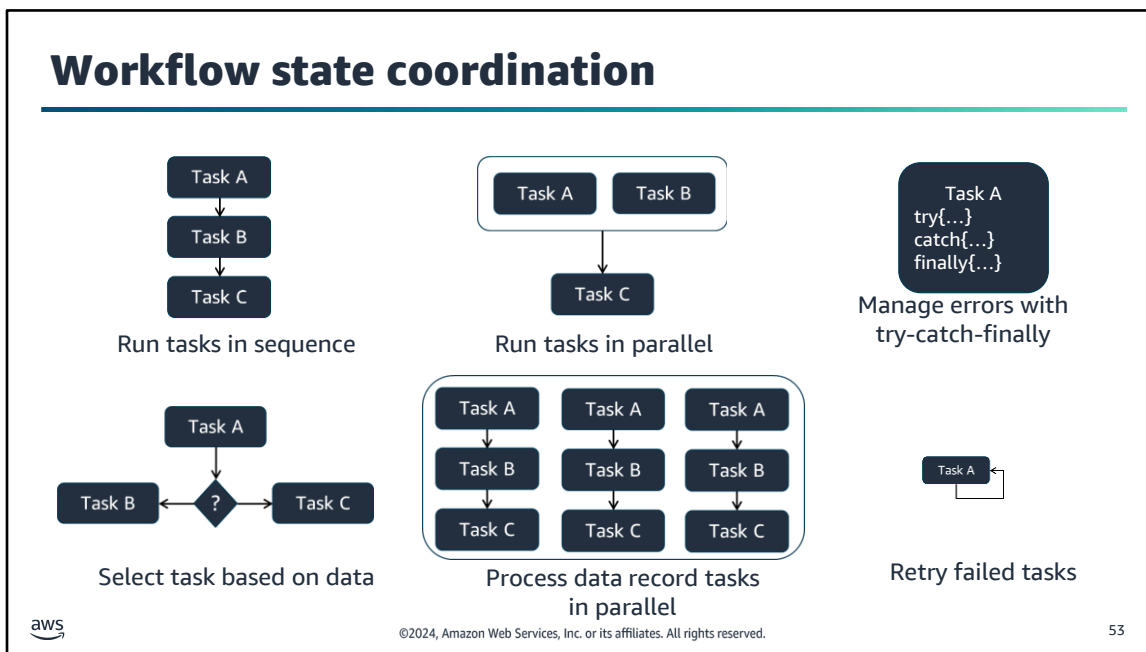
Step Functions gives you several ways to manage your microservice workflows. For long-running workflows, you can use Standard Workflows with the AWS Fargate integration to orchestrate applications running in containers. For short-duration, high-volume workflows that require an immediate response, Synchronous Express Workflows are ideal. These can be used for web-based or mobile applications, which often have workflows of short duration and require the completion of a series of steps before they return a response. You can directly initiate Synchronous Express Workflows from Amazon API Gateway, and the connection is held open until the workflow completes or timeouts. For short duration workflows that don't require an immediate response, Step Functions provides Asynchronous Express Workflows.

Step Functions provides the scalability, reliability, and availability needed to successfully manage your data processing workflows. You can manage millions of concurrent executions with Step Functions because it scales horizontally and provides fault-tolerant workflows. Process Step Functions data faster using parallel executions, like its Parallel state type, or dynamic parallelism using its Map state type. As part of your workflow, you can use the Map state to iterate over objects in a static data store like an Amazon Simple Storage Service (Amazon S3) bucket. Step Functions also lets you retry failed executions or choose a specific way to handle errors without the need to manage a complex process.

Step Functions lets you to orchestrate end-to-end machine learning (ML) workflows on Amazon SageMaker. These workflows can include data preprocessing including data enrichment, post-processing, feature engineering, data validation, and model evaluation.

Security automation can help manage increasingly complex and time-consuming operations, such as upgrading and patching software, deploying security updates to address vulnerabilities, selecting infrastructure, synchronizing data, routing support tickets, and more. The automation of repetitive and time-consuming tasks can allow your organization to complete routine operations quickly and consistently on a large scale. Step Functions allows you to create workflows that automatically scale to meet the needs of your business without

requiring manual intervention. In cases where an error occurs in your workflow, it often doesn't require manual intervention. Step Functions lets you automatically retry failed tasks and has an exponential backoff that can manage errors in your workflow.



AWS Step Functions manages the logic of your application for you. It implements coordinations, 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. 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.

State machine state (step) types		
Work states	Transition states	Stop states
<ul style="list-style-type: none"> <li>• <b>Task:</b> Integrates with AWS services</li> <li>• <b>Activity:</b> Performs a task hosted anywhere</li> <li>• <b>Pass:</b> Passes or filters input data to next state</li> <li>• <b>Wait:</b> Delays workflow for a specified time</li> <li>• <b>State has wait for callback state option:</b> Pauses workflow and waits for callback</li> </ul>	<ul style="list-style-type: none"> <li>• <b>Choice:</b> Adds conditions to control flow to next state</li> <li>• <b>Parallel:</b> Adds branches of nested state machines inside a state machine</li> <li>• <b>Map:</b> Separates workflow for each data record in data set that runs in parallel</li> </ul>	<ul style="list-style-type: none"> <li>• <b>Success:</b> Stops state machine and marks execution as successful</li> <li>• <b>Fail:</b> Stops state machine and marks execution as failed</li> <li>• <b>State has the end parameter:</b> Stops state machine</li> </ul>



©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

54

State machine state (step) types can be classified as work, transition, or stop states.

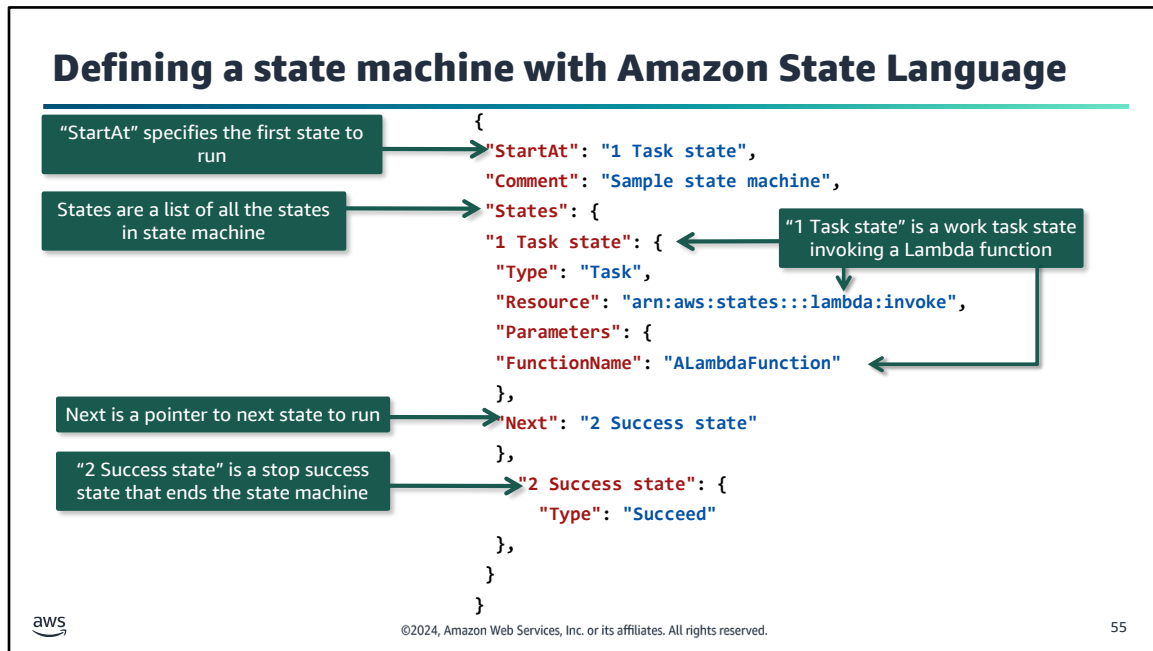
Work states include task, activity, pass, and wait states. A task state represents a unit of work that another AWS service, such as AWS Lambda, performs. Alternatively, a task can call an HTTP endpoint. A task state handles AWS service integrations and can call any AWS service or API supported by Step Functions first-class AWS service integrations and the AWS SDK. The task can integrate synchronously with a request response invocation type, run a job, and wait for the job to complete. A task can optionally call a service with a task token and wait for a callback that includes the token and a payload.

Activities are tasks in your state machine where the work is performed by a worker that can be hosted anywhere. An activity worker can be an application running on an Amazon EC2 instance, an AWS Lambda function, a mobile device, or any application that can make an HTTP connection. Similar to a task, an activity can optionally call a service or HTTP endpoint with a task token and wait for a callback that includes the token and a payload.

A pass state passes its input to its output without performing work. Pass states are useful when constructing and debugging state machines. You can also use a pass state to transform JSON state input using filters, and then pass the transformed data to the next state in your workflow. A wait state delays the state machine from continuing for a specified time. You can choose either a relative time, specified in seconds from when the state begins, or an absolute end time, specified as a timestamp.

Transition states control the flow of the state machine determining which state to run next. A choice state adds conditional logic to a state machine. It contains an array of choice rules that determine which state the state machine transitions to next. You use a comparison operator in a choice rule to compare an input variable with a specific value. For example, using choice rules, you can compare whether an input variable is greater than or less than 50. The parallel state can be used to add separate branches of nested state machines inside a state machine. Use the map state to run a set of workflow steps for each item in a dataset. The map state's iterations run in parallel, which makes it possible to process a dataset quickly. Map states can use a variety of input types, including a JSON array, a list of Amazon S3 objects, or a CSV file.

Stop states ends the state machine run. A state can contain the end parameter which ends the run, or the success and fail states can be used when recording whether the run was successful or not.



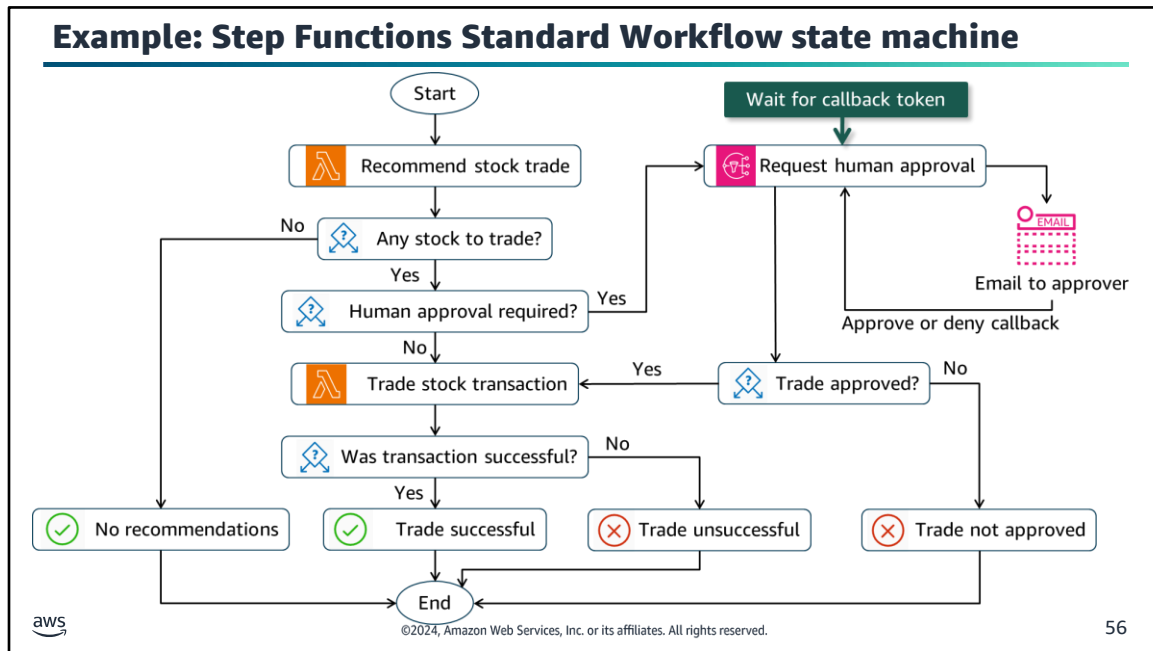
A Step Function state machine is a collection of states defined in the Amazon States Language. Amazon States Language is a JSON-based, structured language used to define the state machine and the collection of states to run. A state machine uses key-value pairs to specify the fields and values of the state machine and always specifies which state to run first.

A state machine always contains a **StartAt** field with the name of the state to point to the first state to run. The **States** field contains a collection of nested JSON documents specifying the states in the state machine. Each state has a name used as an identifier and contains a state type in the **Type** field. Each state type has a set of mandatory and optional fields.

A state should include the **Next** field if control is to be passed to another state. Choice states has multiple **Next** fields for each choice rule.

If a state is the last state to be run in the state machine, it should include the **End** field. Alternatively, control can be passed to a success state or a failed state to end the state machine run.





56

This is an example of a stock trade Step Functions state machine. The first task to run is a Lambda function that returns the stock company identifier, the number of stocks, current stock price, and total trade amount. It's passed to the first choice state which checks whether there's a recommended stock trade. If not, the state machine transitions to the success state and ends.

If there is a recommended trade, then control is passed to the Human approval required choice state. The choice state rule can check whether the total trade amount exceeds a limit, like \$100.00.

If yes, it continues to the Request human approval task, which invokes Amazon Simple Notification Service (Amazon SNS) to request approval by sending an email to the approver with a task token. When the approver approves or denies the request, a callback is the token, and the response is returned to the Request human approval task state. It passes the result to the Trade approved choice state. If not approved, control is passed to the Trade not approved fail state and the run ends.

If the trade was approved, then control is passed to the Trade stock transaction Lambda task state. It returns the transaction result and passes control to the Was transaction successful choice state. If not, then control is passed to the Trade unsuccessful fail state and the run ends. If the transaction was successful, then control is passed to the Trade successful state and the run ends.

## Key takeaways: Orchestrating microservice workflows with AWS Step Functions



- AWS Step Functions is a serverless orchestration service to manage workflows between multiple AWS services.
- A state machine (workflow) is a series of event-driven states (steps).
- A Step Functions state machine is a collection of states defined in the Amazon States Language.
- States can be grouped into work, transition, and stop states.
- A task state can invoke an AWS service or request an activity hosted on any compute service with an HTTP connection.



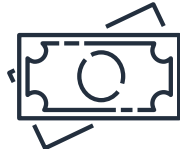
©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.


57



This section looks at extending serverless architectures with Amazon API Gateway and decomposing a monolithic application into microservices.

## Benefits of APIs

		
<b>Standardize app communication</b>	<b>Protect microservices</b>	<b>Monetize API and track statistics</b>
<ul style="list-style-type: none"><li>• Connect software apps written in different languages in a standardized way.</li><li>• Hide implementation complexity.</li></ul>	<ul style="list-style-type: none"><li>• Choose whether to require authorization.</li><li>• Check request formats.</li><li>• Throttle the number of requests.</li><li>• Restrict resource access.</li></ul>	<ul style="list-style-type: none"><li>• Track client usage for billing purposes.</li><li>• Supply usage statistics per client.</li></ul>

 ©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved. 59

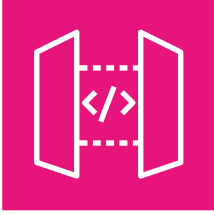
Earlier in this module, you learned that microservices communicate through well-defined APIs. So why are APIs considered to be a best practice for microservices?

APIs act as a bridge or intermediary to connect applications written in different languages with different data type formats. An API usually provides documentation on how to use the API and about the client application's needs to format the API request correctly. The benefit is that the client application making the request doesn't need to understand the complexity or technical details of the microservice implementation.

A microservice is designed to run and scale rapidly. Therefore, it's a good practice to protect a microservice from bad actors. An API can implement authorization mechanisms to ensure that only authenticated applications access the microservice. In addition, the API can check the client request format and return an error if the format is incorrect. The API can also be configured to throttle the number of API requests if there's a limitation on the microservice resources. An API can also restrict access to microservice resources to allow only limited actions on the resource.

An API is also a convenient place to collect client usage data points to analyze how the microservices are utilized. If required, the usage can translate into monetization because the client with preferential usage can pay for the prioritized usage.

## Amazon API Gateway



API Gateway

- Provides ability to create, publish, and maintain REST, HTTP, and WebSocket APIs
- Configurable traffic management, authorization, and resource access control
- Provides access to AWS services and publicly accessible endpoints
- Hosts multiple versions and stages of an application's API
- Establishes client usage plans to monetize and control APIs
- Can cache common responses

aws

©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

60

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. You can build RESTful APIs using both HTTP APIs and REST APIs in Amazon API Gateway. Applications can then access data, business logic, or functionality from your backend services. Backend services can be applications that run on Amazon Elastic Compute Cloud (Amazon EC2), code that runs on AWS Lambda or Amazon Elastic Container Service (Amazon ECS), private applications in a virtual private cloud (VPC) or on-premises, and publicly accessible endpoints. You can also integrate API Gateway directly with some AWS services, such as sending data directly to an Amazon Kinesis stream, AWS Step Functions workflows, or an Amazon DynamoDB database.

API Gateway handles all the tasks that are involved in accepting and processing up to hundreds of thousands of concurrent API requests. API Gateway has features to handle traffic management, authorization and access control, monitoring, and API version management. API Gateway is integrated with Amazon Cognito for authorization and allows Lambda function plug-ins for customized authorization solutions.

You can use API Gateway to host multiple versions and stages for different development and production environments of your APIs. Usage plans help you declare plans for third-party developers that restrict access only to certain APIs, define throttling and request quota limits, and associate them with API keys. You can also extract utilization data on a per-API key basis to analyze API usage and generate billing documents. For example, you can create a plan to only allow 1,000 requests per day and a maximum of 5 requests per second (RPS).

You can enable API caching in Amazon API Gateway to cache your endpoint's responses. With caching, you can reduce the number of calls made to your endpoint and also improve the latency of requests to your API.

Choosing an API type		
REST APIs	HTTP APIs	WebSocket APIs
<ul style="list-style-type: none"> <li>• Collection of routes and methods</li> <li>• For apps that require API management features, such as usage plans, payload validation, private API endpoints, and resource policies</li> <li>• Supports cross-origin resource sharing (CORS)</li> <li>• Stateless</li> </ul>	<ul style="list-style-type: none"> <li>• Collection of routes and methods</li> <li>• For microservices</li> <li>• Lower latency and lower cost than REST APIs</li> <li>• Supports CORS</li> <li>• Stateless</li> </ul>	<ul style="list-style-type: none"> <li>• Collection of WebSocket routes</li> <li>• For real-time applications</li> <li>• Establishes a session between client and backend services</li> <li>• API management features are validate payload schema and data transformations</li> <li>• Stateful</li> </ul>

aws

©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

61

Amazon API Gateway offers two options to create RESTful APIs, namely HTTP and REST APIs, and an option to create WebSocket APIs. Choosing which API type is most suitable for your application depends on the API behavior you require.

RESTful APIs are useful in cloud applications because their calls are stateless. If something fails, stateless components can smoothly redeploy and scale to accommodate load changes. RESTful APIs are made up of a collection of routes and methods for application resources. Methods are HTTP verbs such as GET, POST, PUT, PATCH, and DELETE that corresponds to data create, read, update, and delete (CRUD) actions. For example, a request with a POST method and path /class-roster with student information as parameters will add a student to a class-roster. A GET method with path /class-roster will return all the students registered for the class.

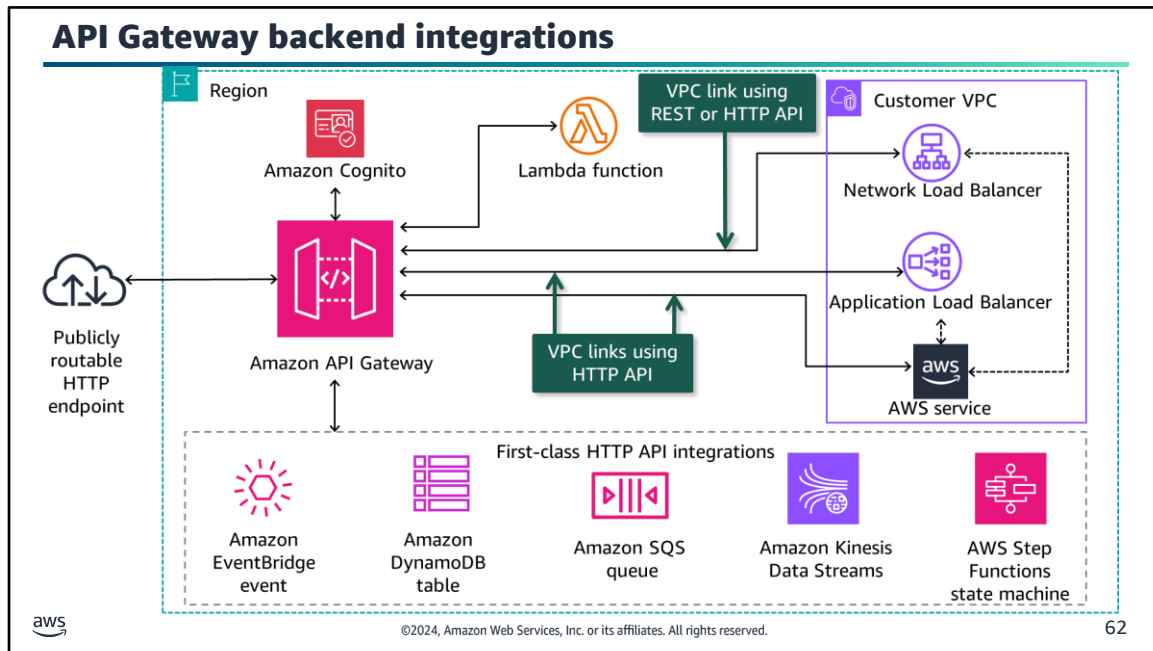
API Gateways REST APIs offer API proxy functionality and an extensive set of API management features in a single solution. REST APIs uses usage plans that allocate API keys to clients to track API usage for billing purposes and control usage limits with throttling. You can also validate incoming requests against a request payload schema and do limited data transformations to match backend data formats. REST APIs allows private API endpoints to API Gateway which is only accessible from a private subnet in a customer VPC. You can use API Gateway resource policies to allow your API to be securely invoked users from a specified AWS account, specified source IP address ranges or CIDR blocks, or specified VPCs in any account. So API Gateway REST APIs are particularly suitable when you need an extensive set of API management features.

HTTP APIs are optimized for building serverless workload APIs that proxy to AWS Lambda functions or HTTP backends. They are lightweight and provide improved performance over REST APIs with lower latency. HTTP APIs also provide a better price point than REST APIs because they do not offer API management functionality. This makes HTTP APIs suitable for microservice implementation using Lambda functions and Amazon ECS containers that need low latency responses.

WebSocket APIs maintain a persistent connection between connected clients to enable real-time message communication. In a WebSocket API, the client and the server can both send messages to each other at any time. Backend servers can easily push data to connected users and devices, avoiding the need to implement complex polling mechanisms. For example, you could build a serverless application using an API Gateway

WebSocket API and AWS Lambda to send and receive messages to and from individual users or groups of users in a chat room in the same domain.

Both REST APIs and HTTP APIs support cross-origin resource sharing (CORS). CORS is typically required to build web applications that access APIs hosted on a different domain or origin. You can enable CORS to allow requests to your API from a web application that's hosted on a different domain.



Because API Gateway is designed to act as a bridge between applications, it supports many backend integrations. As you learned in the microservices section of this module, API Gateway and AWS Lambda can function as a microservice making use of Amazon Cognito for microservice authorization.



An HTTP proxy integration enables you to connect an API route to a publicly routable HTTP endpoint. With this integration type, API Gateway passes the entire request and response between the frontend and the backend. To create an HTTP proxy integration, provide the URL of a publicly routable HTTP endpoint.

You can integrate your HTTP API with AWS services by using first-class integrations. A first-class integration connects an HTTP API route to an AWS service API. When a client invokes a route that's backed by a first-class integration, API Gateway invokes an AWS service API for you. For example, you can use first-class integrations to send a message to an Amazon Simple Queue Service (Amazon SQS) queue or to start an AWS Step Functions state machine.

If your application solution is using AWS services or load balancers in your VPC, Amazon API Gateway can be configured with a VPC link. For REST APIs, a VPC link provides access to a Network Load Balancer in a VPC. For HTTP APIs, a VPC link can provide access to a Network Load Balancer, an Application Load Balancer, and other AWS services in a VPC, such as Amazon EC2 instances and Amazon ECS containers.



**Activity:  
Decomposing a  
Monolithic  
Application with  
AWS API Gateway**



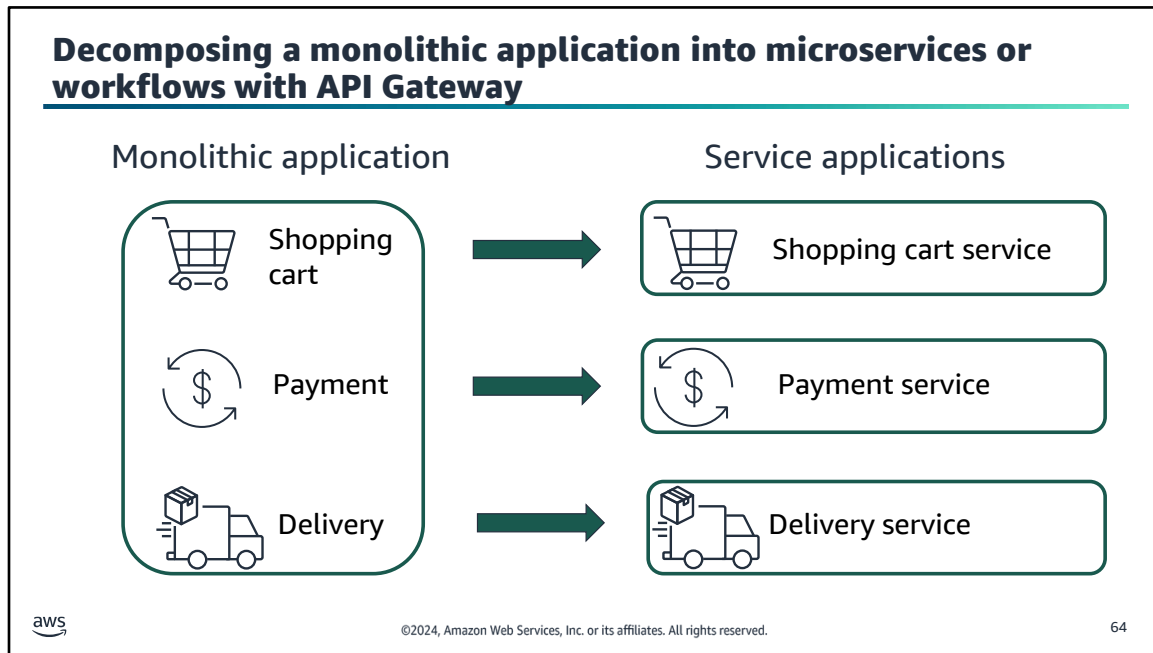
Choose the appropriate Amazon API Gateway API and AWS service or services to decompose an online shopping monolithic application into services. The following are the components of the monolithic shopping application:

- Shopping cart
- Payments
- Delivery

©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

63

This is an instructor-led activity to discuss various options for decomposing an online shopping monolithic application into microservices.



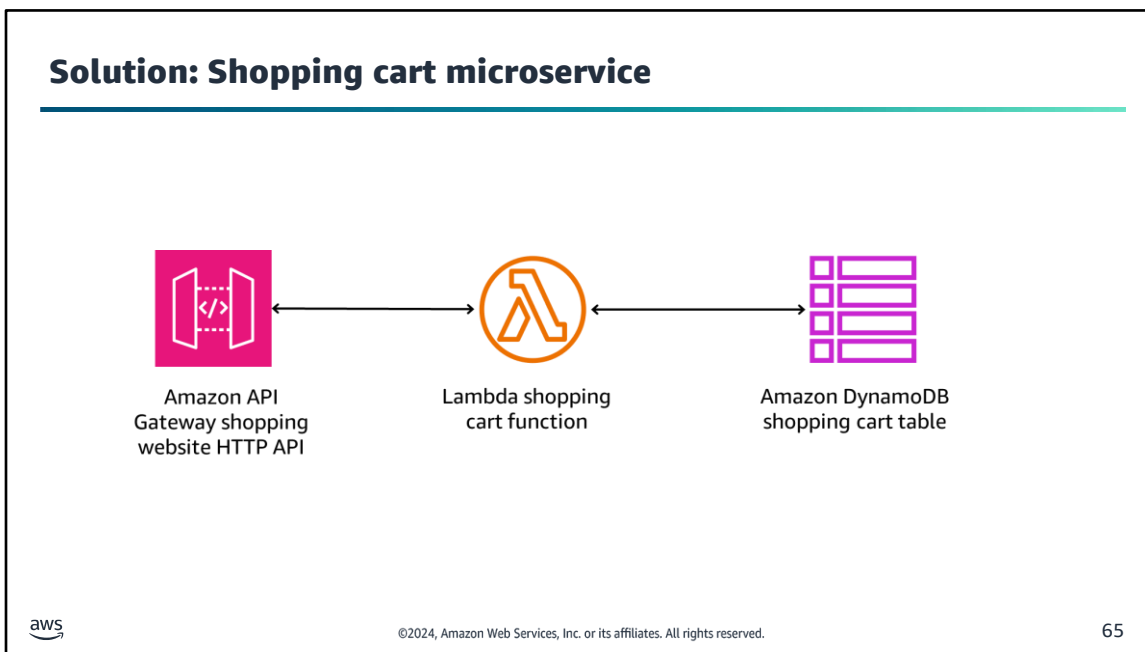
Monolithic application module functionality is as follows:

**Shopping cart:** Online customer adds, removes, or updates the number of items in the shopping cart with a required response time of single digit seconds.

**Payments:** Online customer finalizes shopping cart and pays for the number of items in the shopping cart. The payment system should be ported as-is from the on-premises solution with minimal code and configuration changes.

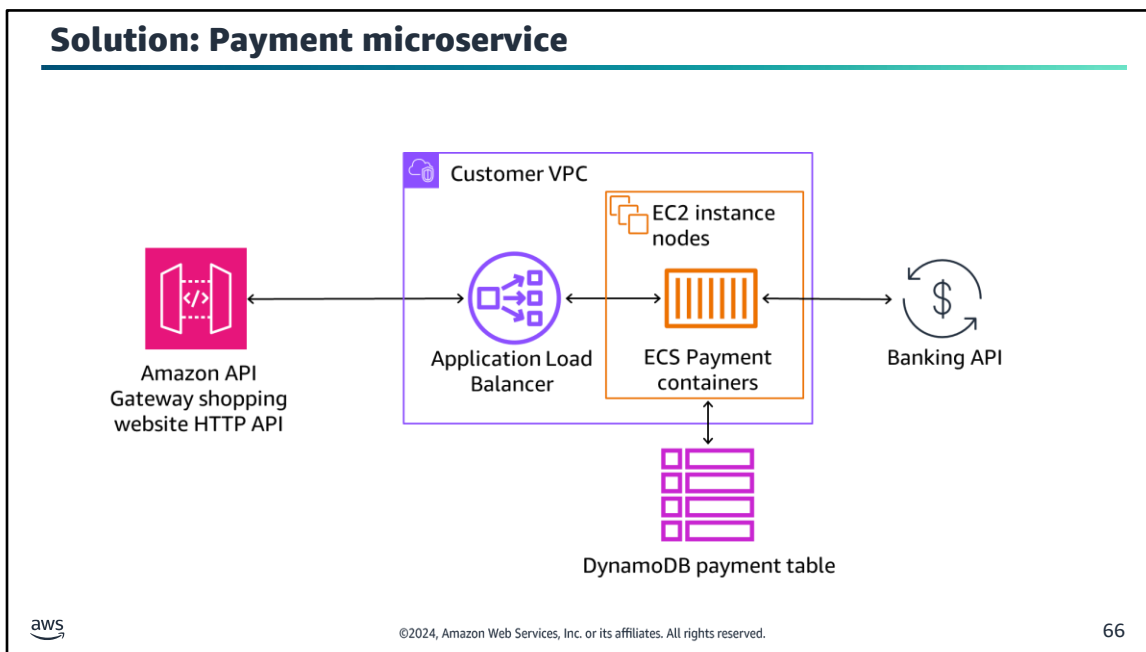
**Delivery:** When the payment is successful, the shopping cart items must be delivered to the customer. This process might take up to 5 days, and the customer should be notified when delivery is made.

Refer to the API Gateway backend integrations page to choose the appropriate API Gateway API and AWS services for each microservice or workflow.

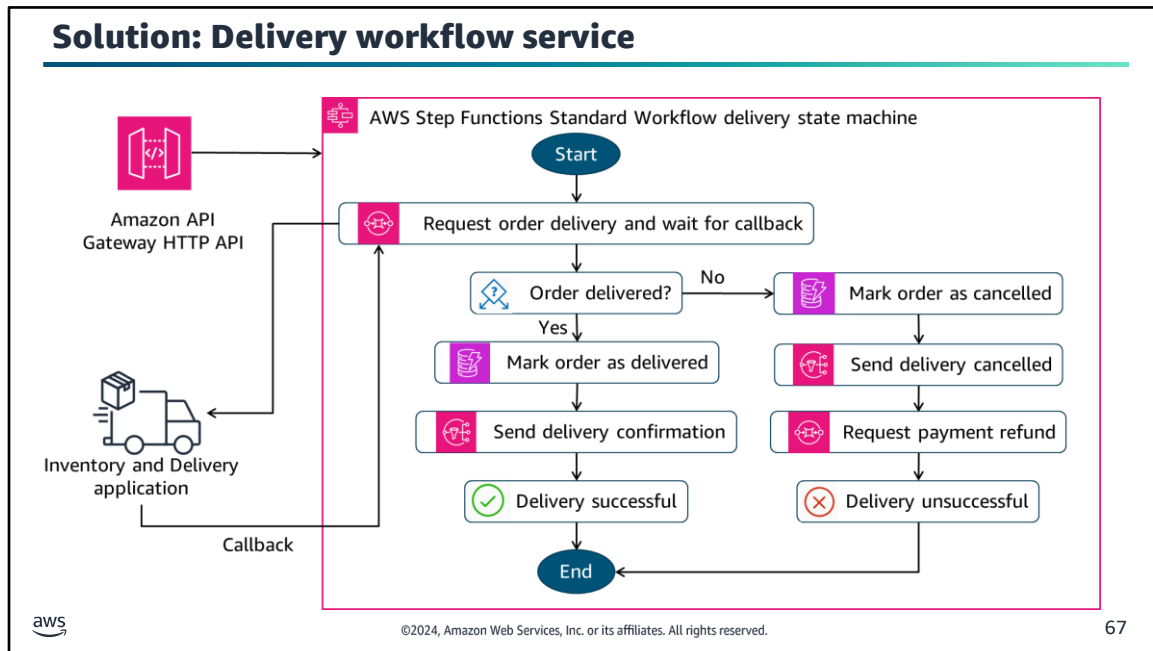


For this solution, an Amazon API Gateway HTTP API was chosen to provide AWS Lambda function and Amazon ECS container integration.

The shopping cart service consists of small, millisecond transactions. Use Amazon API Gateway shopping website HTTP API service to route shopping cart requests to the AWS Lambda shopping cart function. Shopping cart transactions are stored on the Amazon DynamoDB shopping cart table. This solution will satisfy the performance requirement of single digit second response times.



Because the payment system is ported from the on-premises environment with minimum code and configuration changes, porting it to ECS payment containers would be a suitable solution. To distribute the payment transactions, the Application Load Balancer receives the payment requests from API Gateway shopping website HTTP API. The Application Load Balancer distributes transactions to containers, which uses a publicly accessible banking API for payment authorization. The containers store the payments on the DynamoDB payment table.



67

Because delivery is a process that can take multiple days to complete, this is not a microservice, and it would be better suited as a longer running workflow service.

After successful payment, the Amazon API Gateway HTTP API receives a delivery call and forwards the request to the AWS Step Functions Standard Workflow delivery state machine. The state machine puts the delivery request on an Amazon SQS delivery queue and suspends the state machine. The inventory and delivery API polls the queue and picks up the delivery request. Upon delivery, the inventory and delivery API does a callback to the delivery state machine. Control is passed to the choice state Order delivered. If the order was delivered, the delivery state machine updates the DynamoDB delivery table with a delivered status and sends an email notification to the customer using Amazon Simple Notification Service (Amazon SNS). The state machine ends with execution marked as successful.

If the order was not delivered, the delivery state machine updates the DynamoDB delivery table with a cancelled status and sends an email cancellation to the customer using Amazon SNS. The next state puts a request payment refund on an Amazon SQS refund queue, and the state machine ends with execution marked as unsuccessful.

Please note, this is just a generic solution, and there are many different viable solutions depending on the actual scale and capacity that's required.

## Key takeaways: Extending serverless architectures with Amazon API Gateway



- Amazon API Gateway provides the ability to create, publish, and maintain application APIs.
- It provides access to AWS services and publicly accessible endpoints.
- Use REST APIs when full API management and control are required.
- Use HTTP APIs when lower latency and lower cost than REST APIs are required.
- Use WebSocket APIs for real-time applications that require an active session.



©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

68



This is an optional lab. The next slides summarize what you will do in the lab, and you will find the detailed instructions in the lab environment.

## Microservices lab tasks:



- In this lab, you will do the following:
- Migrate a monolithic Node.js application to run in a Docker container.
- Refactor a Node.js application from a monolithic design to a microservices architecture.
- Deploy a containerized Node.js microservices application to Amazon ECS.
- Open your lab environment to start the lab and find additional details about the tasks that you will perform.

©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

70

Access the lab environment through your online course to get additional details and complete the lab.



## Debrief: Microservices lab

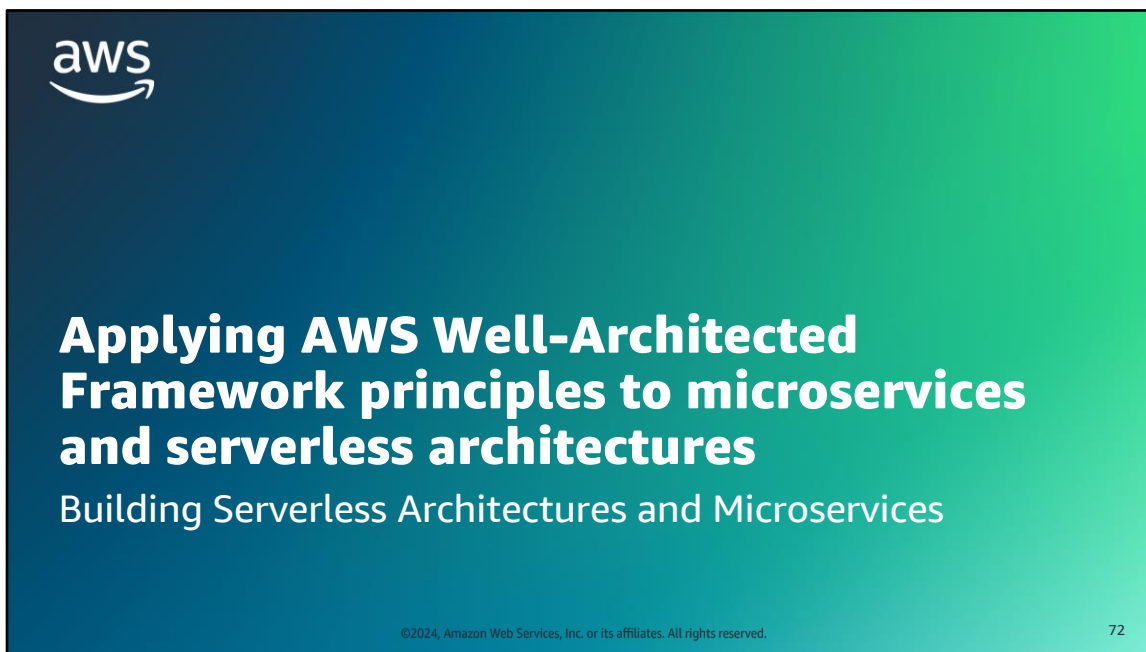
---

- How did you deploy the containerized monolithic application to an Amazon ECS runtime environment?
- What did you learn about task definitions in this lab?

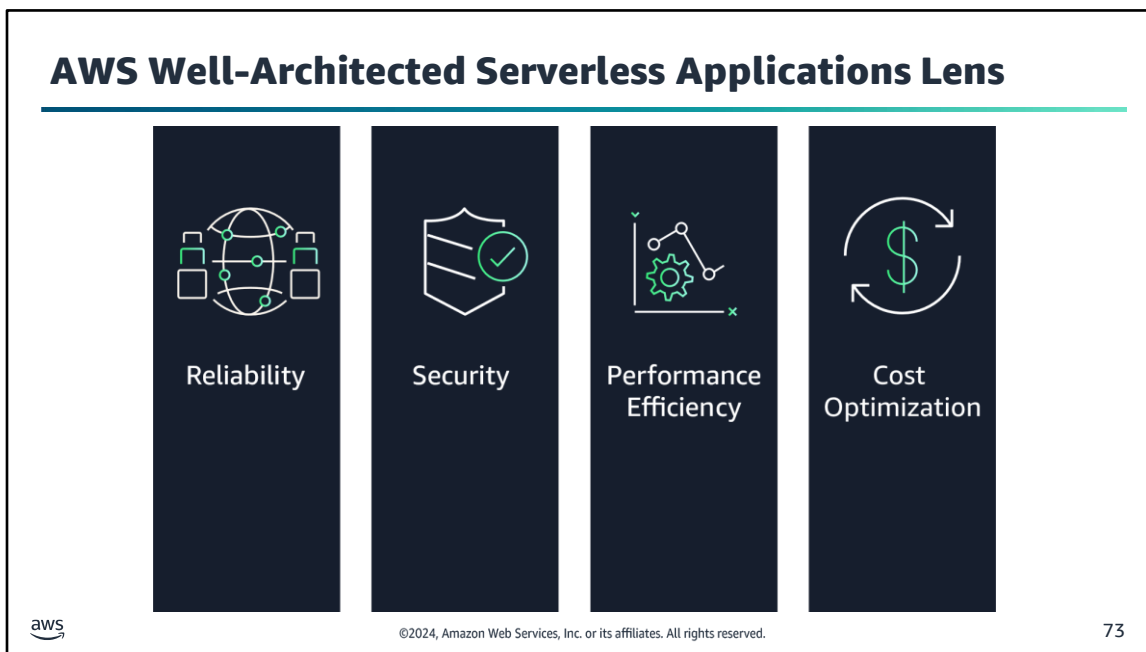


©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

71



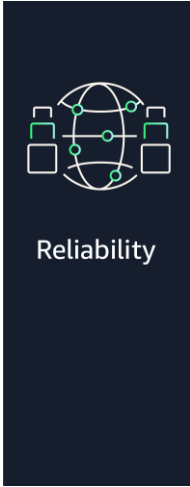
This section looks at how to apply the AWS Well-Architected Framework principles to serverless architectures, including microservices.



The AWS Well-Architected Framework has six pillars, and each pillar includes best practices and a set of questions that you should consider when you architect cloud solutions. This section highlights a few best practices from the pillars that are most relevant to this module. Find the complete set of best practices by pillar on the Well-Architected Framework website. A link is provided in the content resources section of your online course.

AWS provides a specialized Serverless Applications Lens for the AWS Well-Architected Framework that covers common serverless applications scenarios and identifies key elements to ensure that your workloads are architected according to best practices. For more information, see the Serverless Applications Lens – AWS Well-Architected Framework link in the content resources section.

## Best practice approach: Failure management



Reliability

### Best practices

- Use a dead-letter queue mechanism to retain, investigate, and retry failed transactions.
- Roll back failed transactions.

aws

©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

74

Certain parts of a serverless application are dictated by asynchronous calls to various components in an event-driven fashion, such as by publish and subscribe and other patterns. When asynchronous calls fail, they should be captured and retried whenever possible. Otherwise, data loss can occur, resulting in a degraded customer experience.

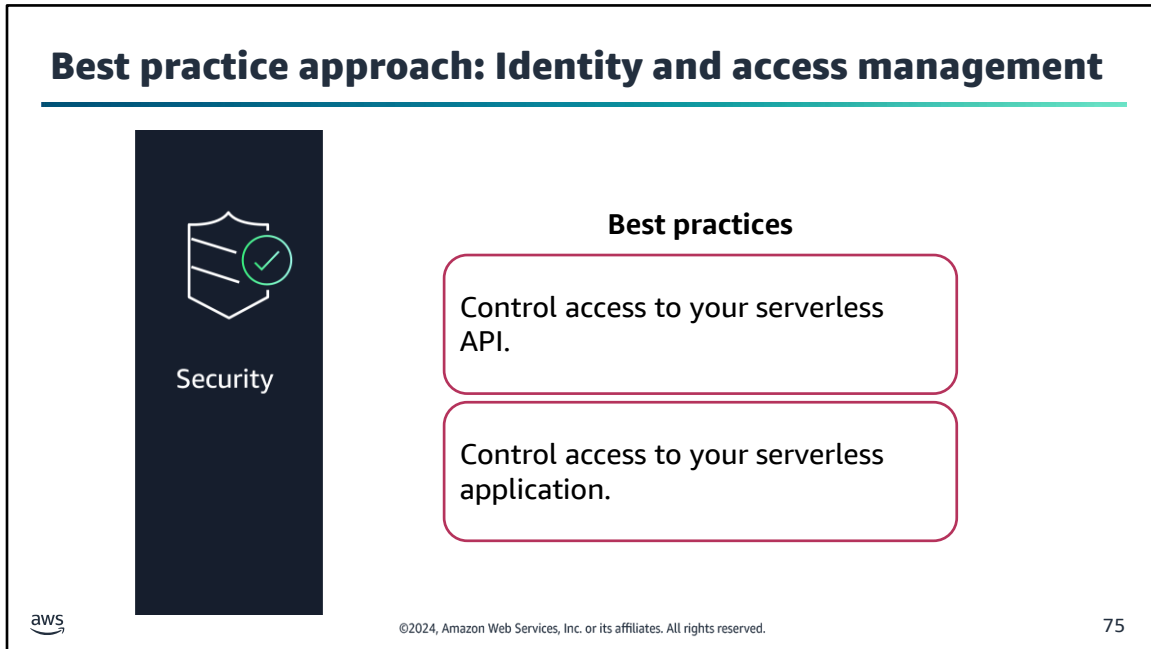
**Dead-letter queue mechanism:** Use a dead-letter queue mechanism to retain, investigate, and retry failed transactions. AWS Lambda allows failed transactions to be sent to a dedicated Amazon Simple Queue Service (Amazon SQS) dead-letter queue on a per function basis.

Amazon Kinesis Data Streams and Amazon DynamoDB Streams retry the entire batch of items. Repeated errors block processing of the affected shard until the error is resolved or the items expire. Within AWS Lambda, you can configure controls to retry while processing data records, and effectively remove poison-pill messages from the batch by sending its metadata to an Amazon SQS dead-letter queue for further analysis.

**Roll back failed transactions:** For synchronous parts that are transaction-based and depend on certain guarantees and requirements, rolling back failed transactions can be achieved by using AWS Step Functions state machines, which will decouple and simplify the logic of your application.

In this module, you learned about a number of AWS services that support these best practices, including AWS Lambda and Amazon API Gateway.

## Best practice approach: Identity and access management



**Security**

**Best practices**

- Control access to your serverless API.
- Control access to your serverless application.

aws

©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

75

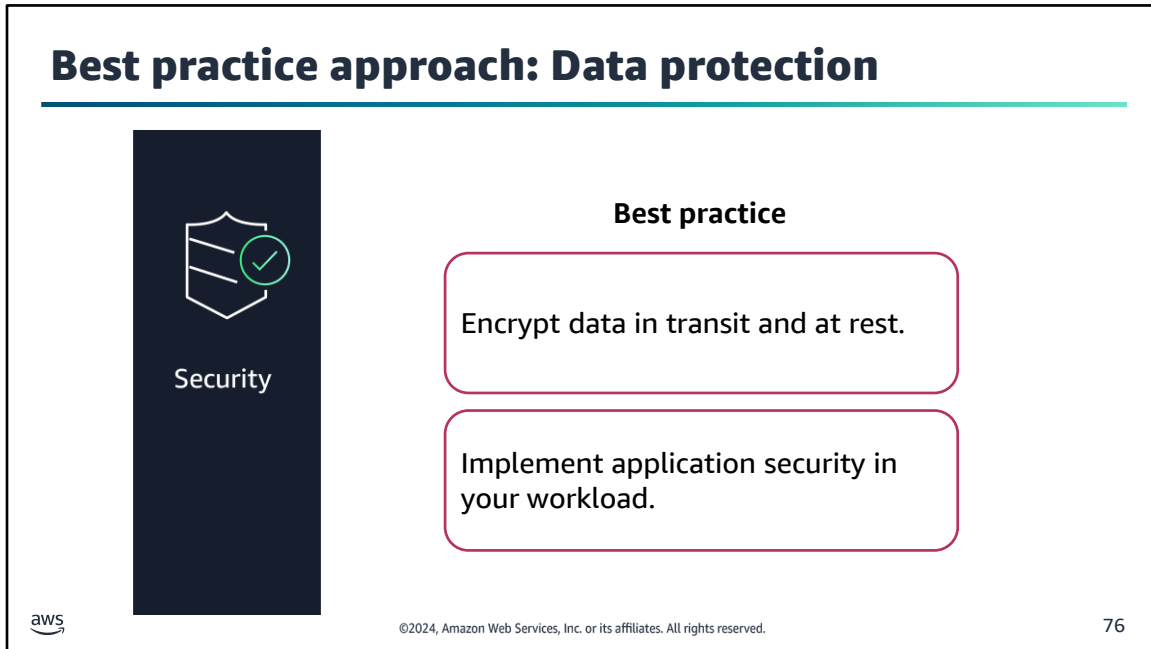
APIs are often targeted by attackers because of the operations that they can perform and the valuable data that they can obtain. There are various security best practices to defend against these attacks.

**Control access to your serverless API:** From an authentication and authorization perspective, there are mechanisms to authorize an API call within API Gateway, such as Amazon Cognito user pools, API Gateway Lambda authorizer, and API Gateway resource policies. It's important to understand whether and how any of these mechanisms are implemented.

**Manage security boundaries of your serverless application:** With Lambda functions, it's recommended that you follow least-privileged access and only allow the access needed to perform a given operation. Attaching a role with more permissions than necessary can open up your systems for abuse. With the security context, having smaller functions that perform scoped activities contribute to a more well-architected serverless application.

In this module, you learned about a number of AWS services that support these best practices, including AWS Lambda, Amazon Cognito, and API Gateway.

## Best practice approach: Data protection



**Best practice**

Encrypt data in transit and at rest.

Implement application security in your workload.

aws

©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

76

Malformed or intercepted input can be used as an attack vector which could gain access to a system or cause a malfunction. Sensitive data should be protected at all times in all layers possible, as discussed in detail in the AWS Well-Architected Framework. The recommendations in that whitepaper still apply here.

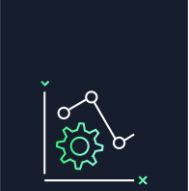
**Encrypt data in transit and at rest:** With regard to API Gateway, sensitive data should be either encrypted at the client-side before making its way as part of an HTTP request or sent as a payload as part of an HTTP POST request. That also includes encrypting any headers that might contain sensitive data prior to making a given request. Concerning Lambda functions or any integrations that API Gateway might be configured with, sensitive data should be encrypted before any processing or data manipulation. This will prevent data leakage if such data gets exposed in persistent storage or by standard output that's streamed and persisted by Amazon CloudWatch Logs.

Lambda functions should persist encrypted data in Amazon DynamoDB, Amazon OpenSearch Service, or Amazon Simple Storage Service (Amazon S3) along with encryption at rest. It's strongly advised to not send, log, and store unencrypted sensitive data, either as part of HTTP request path or query strings or in the standard output of a Lambda function.

**Implement application security in your workload:** Validate and sanitize inbound events and perform a security code review as you normally would for non-serverless applications. For API Gateway, set up basic request validation as a first step to ensure that the request adheres to the configured JSON-schema request model and any required parameters in the URL, query string, or headers. Application-specific deep validation should be implemented, whether that's as a separate Lambda function, library, framework, or service.

In this module, you learned about a number of AWS services that support these best practices, including AWS Lambda and API Gateway.

## Best practice approach: Selection



Performance Efficiency

### Best practice

Optimize the performance of your serverless application.

aws

©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

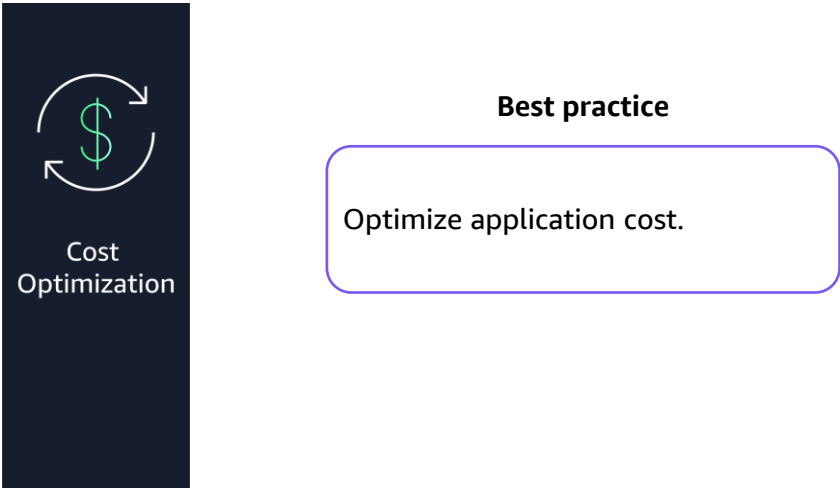
77

Because serverless application components scale at different rates, it's important to ensure optimum performance by testing the application in various manners. Run performance tests on your serverless application using steady and burst rates. Using the result, try tuning capacity units and the provisioning model, and load test after changes to help you select the best configuration.

**Optimize the performance of your serverless application:** With Amazon API Gateway, you can use edge endpoints for geographically dispersed customers. Use regional endpoints for regional customers and when using other AWS services within the same Region. With AWS Lambda, you can test different memory settings because CPU, network, and storage IOPS are allocated proportionally. With AWS Step Functions, you should test Standard and Express Workflows and look at the per second rates for execution start rate and state transition rate.

In this module, you learned about a number of AWS services that support this best practice, including AWS Lambda, API Gateway, and Step Functions.

## Best practice approach: Cost-effective resources



The diagram consists of two main parts. On the left, a dark blue vertical rectangle contains a white circular arrow icon with a green dollar sign in the center, and the text 'Cost Optimization' below it. On the right, a light blue rounded rectangle contains the text 'Best practice' above 'Optimize application cost.'.

Cost Optimization

Best practice

Optimize application cost.

aws

©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

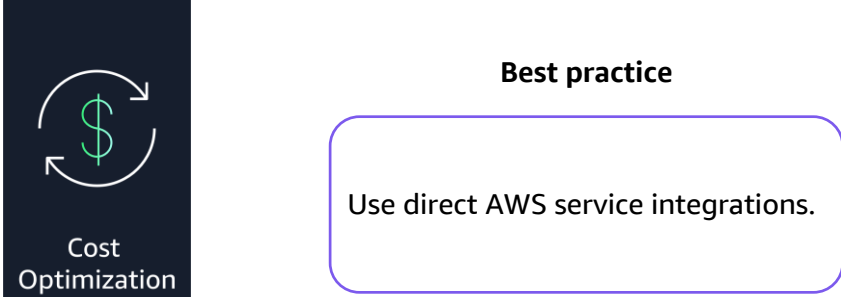
78

Serverless architectures are easier to manage in terms of correct resource allocation. Due to its pay-per-value pricing model and scale based on demand, serverless effectively reduces the capacity planning effort.

**Optimize application cost:** Because Lambda proportionally allocates CPU, network, and storage IOPS based on memory, the faster the initiation, the cheaper and more value your function produces due to 1-ms billing incremental dimension.



## Best practice approach: Optimizing over time



**Cost Optimization**

**Best practice**

Use direct AWS service integrations.

aws

©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

79

AWS services such as Amazon EventBridge, Amazon API Gateway, AWS Step Functions, and AWS Lambda can collectively integrate with almost all AWS services. Putting together a serverless architecture requires careful planning to ensure that you selected only the required services needed.

**Use direct AWS service integrations:** If your Lambda function isn't performing custom logic while integrating with other AWS services, chances are that it might be unnecessary. API Gateway, Step Functions, EventBridge, and Lambda destinations can directly integrate with a number of services and provide you more value and less operational overhead.

As an example, you can implement click stream ingestion data with a client using API Gateway to run a Lambda function which puts the request on an Amazon Kinesis Data Firehose stream to deliver it to an Amazon S3 bucket. A more cost effective solution would be to directly put the clickstream data on the Amazon Kinesis Data Firehose stream from the client. In this manner, API Gateway and the Lambda function costs are saved.

In this module, you learned about a number of AWS services that support this best practice, including API Gateway and Step Functions.

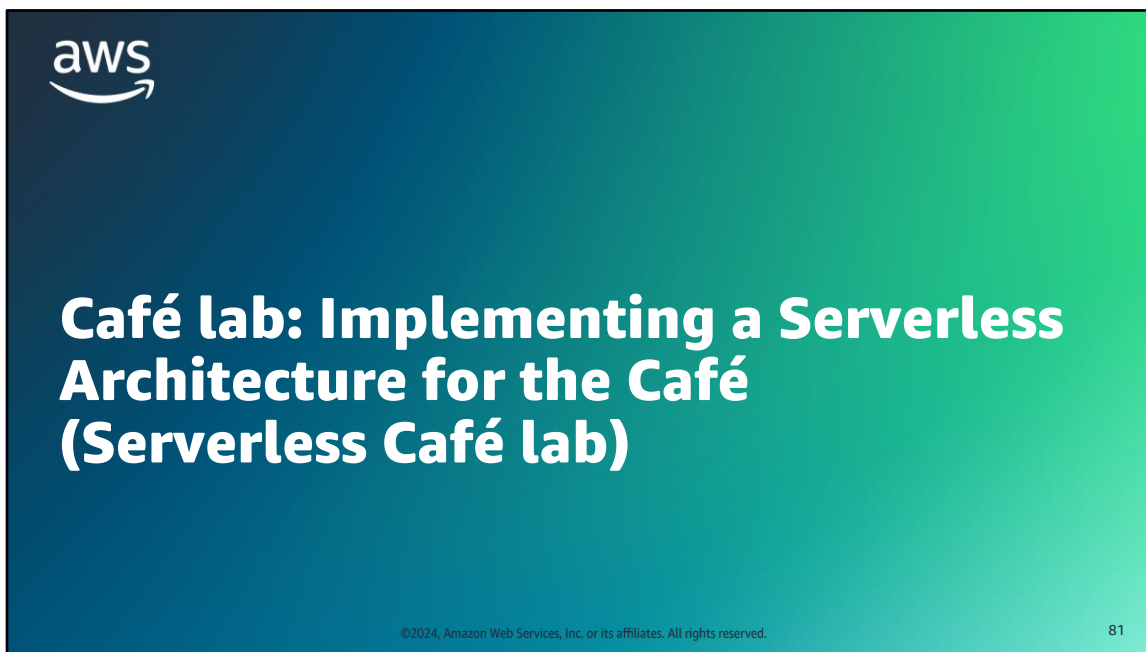
## Key takeaways: Applying AWS Well-Architected Framework to serverless architecture



- Use a dead-letter queue mechanism to retain, investigate, and retry failed transactions.
- Roll back failed transactions.
- Control access to your serverless API.
- Control access to your serverless application.
- Encrypt data in transit and at rest.
- Implement application security in your workload.
- Optimize the performance of your serverless application.
- Optimize application cost.
- Use direct AWS service integrations where available.

©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

80



You will now complete a lab. The next slides summarize what you will do in the lab, and you will find the detailed instructions in the lab environment.

## The evolving café architecture: version 7

Architecture Version	Business reason for update	Technical requirements/ architecture update
V1	Static website for small business.	Host the website on Amazon S3.
V2	Add online ordering.	Deploy web application and database on Amazon EC2.
V3	Reduce effort to maintain the database and secure its data.	Separate web and database layers. Migrate database to Amazon RDS on a private subnet.
V4	Enhance the security of the web application.	Use Amazon VPC features to configure and secure public and private subnets.
V5	Ensure the website can handle an expected increase in traffic and remain highly available and resilient to failure.	Add a load balancer, implement auto scaling on the EC2 instances and distribute compute and database instances across 2 availability zones.
V6	Consistently deploy, manage, and update café resources across Regions.	Build a version-controlled CloudFormation template to deploy the network and application layers. Deploy the CloudFormation stack to another Region.
V7	Improve performance and reduce costs while supporting reporting requirements.	Deploy Lambda functions that connect to the Amazon RDS database and generate a report based on a schedule.



©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.





82

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.

## Serverless Café lab tasks



- In this lab, you will do the following:
  - Implement a serverless architecture to generate a daily sales report.
  - Open your lab environment to start the lab and find additional details about the tasks that you will perform.

©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

83

Access the lab environment through your online course to get additional details and complete the lab.

## Debrief: Serverless Café lab

---

- When you created the DataExtractor Lambda function, how did you configure it to extract data from the Amazon RDS database?
- What did you do to configure the salesAnalysisReport function to run daily at a specified time?





This section summarizes what you have learned and brings the module to a close.

## Module summary

---

This module prepared you to do the following:

- Define serverless architectures.
- Identify the characteristics of microservices.
- Architect a serverless solution with AWS Lambda.
- Define how containers are used in AWS.
- Describe the types of workflows that AWS Step Functions supports.
- Describe a common architecture for Amazon API Gateway.
- Use the AWS Well-Architected Framework principles when building serverless architectures.



©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

86





## Considerations for the café

---

- Discuss how the café lab in this module answered the key questions and decisions that were presented at the start of this module for the café business.



## Module knowledge check



- The knowledge check is delivered online within your course.
- The knowledge check includes 10 questions based on that material that was presented on the slides and in the slide notes.
- You can retake the knowledge check as many times as you like.

©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

88

Use your online course to access the knowledge check for this module.

## Sample exam question

What is the most effective use of Amazon Elastic Container Service (Amazon ECS) when refactoring a monolithic application to use a microservice architecture?

Identify the key words and phrases before continuing.

The following are the key words and phrases:

- Amazon ECS
- refactoring a monolith
- microservice architecture



©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

89

The question asks you to consider the solution in terms of Amazon ECS and focuses on breaking down a monolith into a microservice architecture.

## Sample exam question: Response choices

What is the most effective use of Amazon Elastic Container Service (Amazon ECS) when refactoring a monolithic application to use a **microservice architecture**?

Choice	Response
A	Create services that each provide a distinct function of the application, and run multiple services in a single container that Amazon ECS manages.
B	Port the application to a new image, and run it in a container that Amazon ECS manages.
C	Refactor the application and centralize common functions to create a smaller code footprint.
D	Create services that each provide a distinct function of the application, and run each service in a separate container that Amazon ECS manages.



©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.


90

Use the key words that you identified on the previous slide, and review each of the possible responses to determine which one best addresses the question.

## Sample exam question: Answer

The answer is D.

Choice	Response
D	Create services that each provide a distinct function of the application, and run each service in a separate container that Amazon ECS manages.

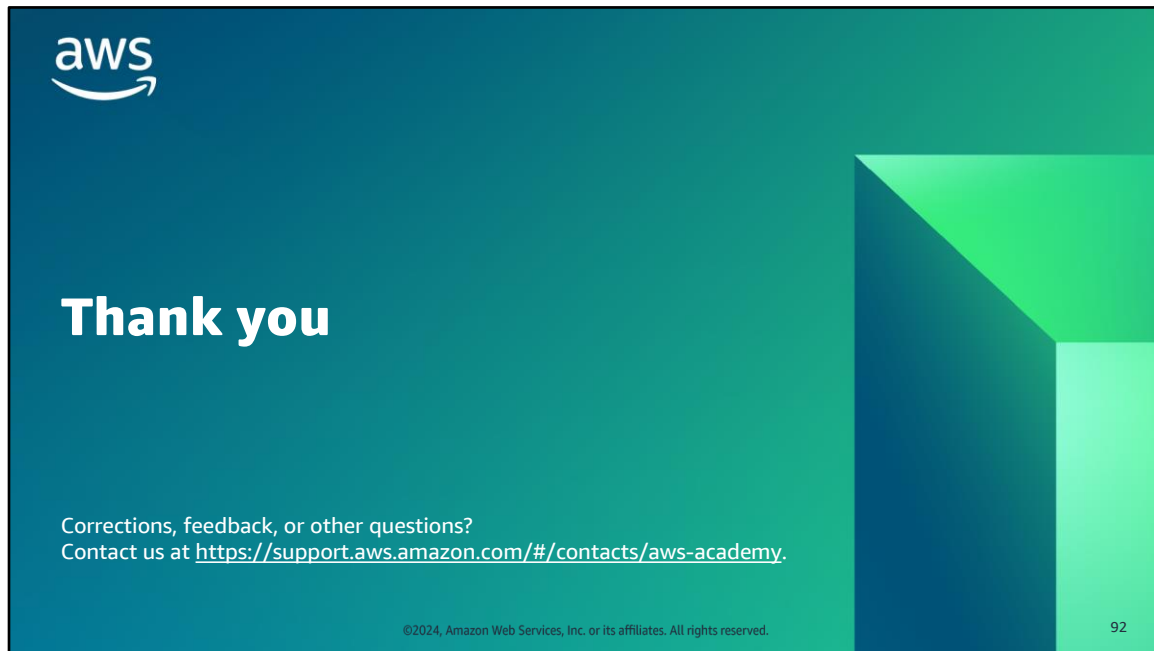
 ©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved. 91

Choice A (Create multiple services in a single container) doesn't take full advantage of a microservice architecture.

Choice B (Port the application to a new image) isn't the best choice. Just porting the application to a container won't result in refactoring or take advantage of a microservice architecture.

Choice C (Refactor and centralize common functions) doesn't align to best practices for microservices. Microservice architectures decentralize code. Refactoring the monolithic application into a smaller set of centralized functions doesn't accomplish the goal of breaking it up into separate, independent services.

Choice D (Create services for distinct functions and run each in a container) is the best choice. The first step in moving to a microservice architecture is to break up monolithic applications into services that each provide a more specialized function.



That concludes this module. The Content Resources page of your course includes links to additional resources that are related to this module.