CloudFormation enables infrastructure as code

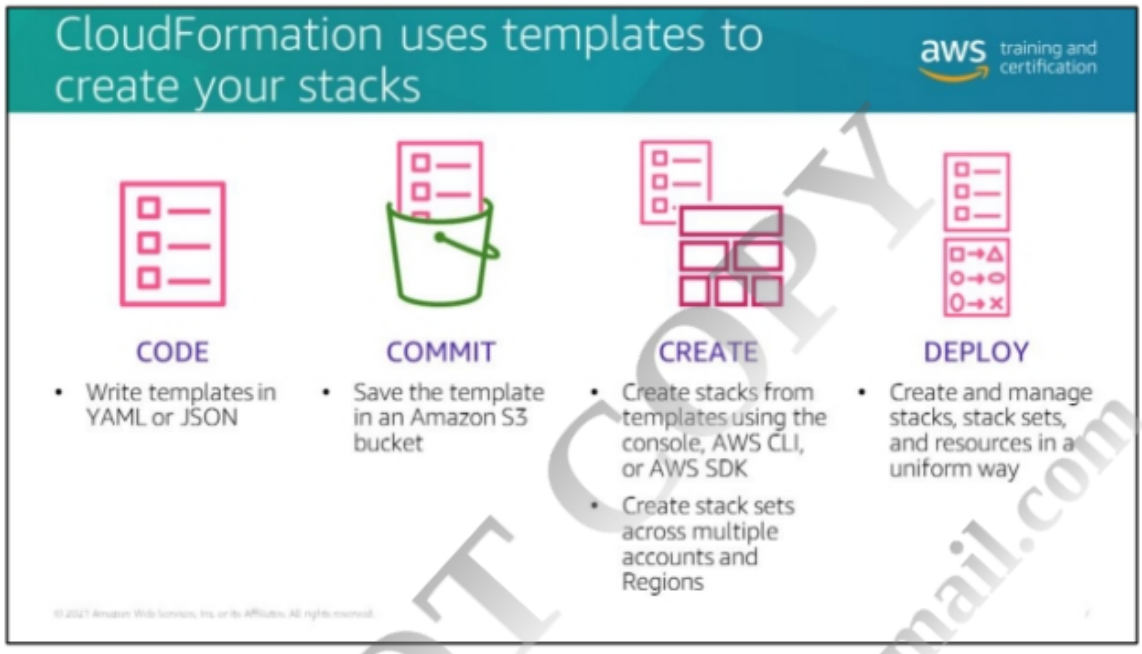
aws training and certification

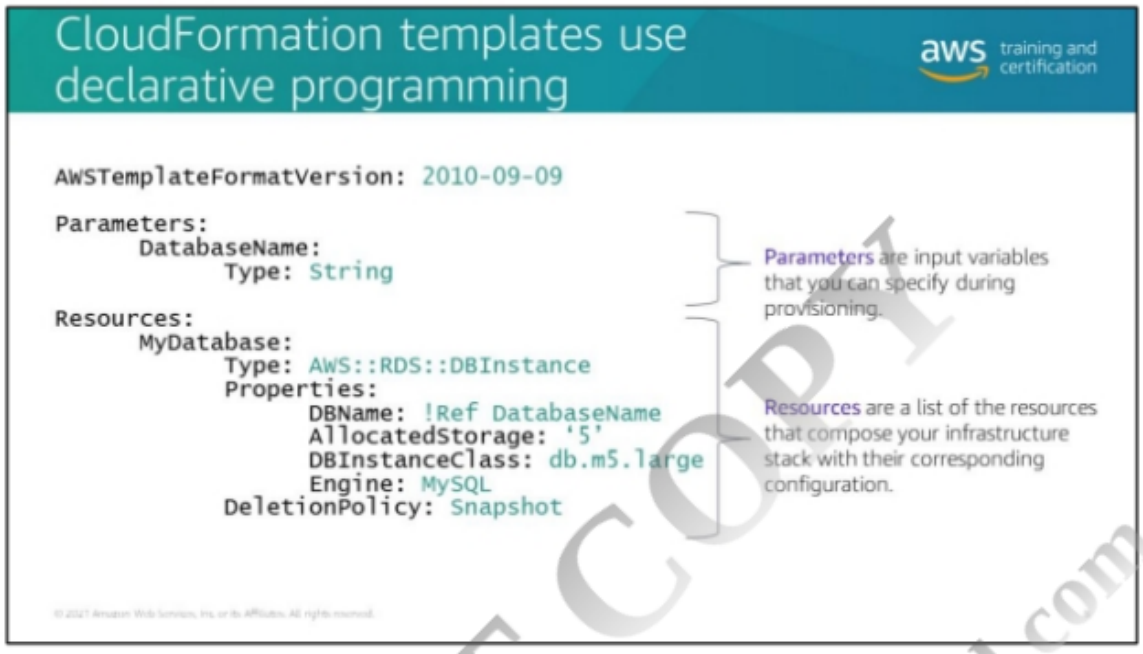
AWS CloudFormation

- Provides a common language to declare infrastructure
- Supports templates in YAML or JSON
- Automates the provisioning and ongoing updates of resources by creating CloudFormation stacks from a template
- Supports both declarative and imperative options

Declarative options:
- Basic YAML/JSON
- Transforms
- Macros
- Include
- AWS Serverless Application Model (AWS SAM)

Imperative options:
- AWS Cloud Development Kit (AWS CDK) (Python, TypeScript, Java, and C#)
- Troposphere (Python)
- SparkleFormation (Ruby)
- GoFormation (GoLang)





YAML and JSON are used to declare the desired state.

You're not explicitly telling AWS CloudFormation how to do things; rather, you're indicating what you want the desired state of your resources to be.

CloudFormation templates use declarative programming.

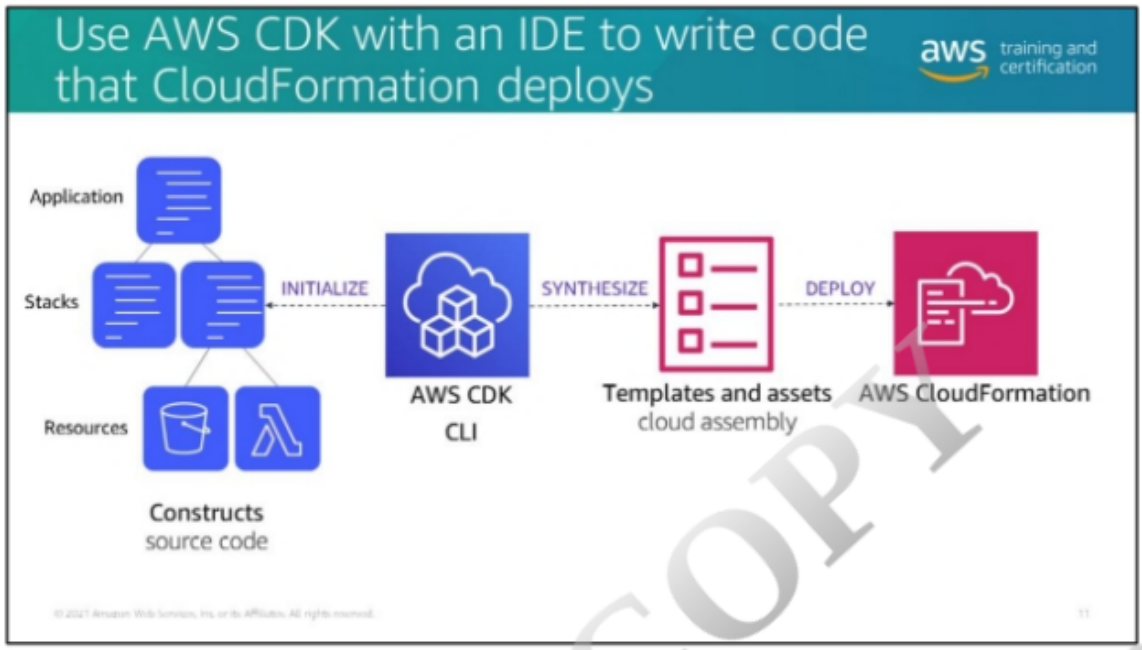
A template declares the desired state. It does not explicitly say how to achieve it.

AWS CDK enables infrastructure as code through imperative programming

aws training and certification

AWS Cloud Development Kit

- Uses the familiarity and expressive power of **programming languages** for modeling your applications **imperatively**
- Provides a library of **constructs** that covers many AWS services and features
- Provisions resources with **CloudFormation**

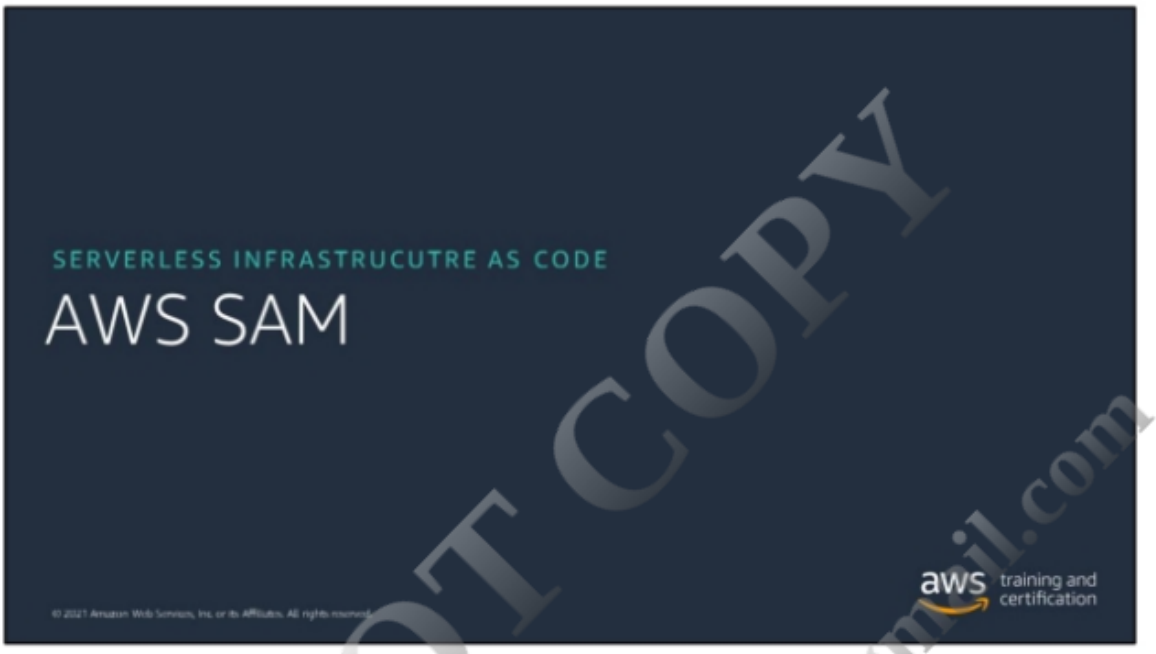


An AWS CDK application is an application written in TypeScript, JavaScript, Python, Java, or C# that uses AWS CDK to define AWS infrastructure. An application defines one or more stacks.

Stacks (equivalent to CloudFormation stacks) contain constructs.

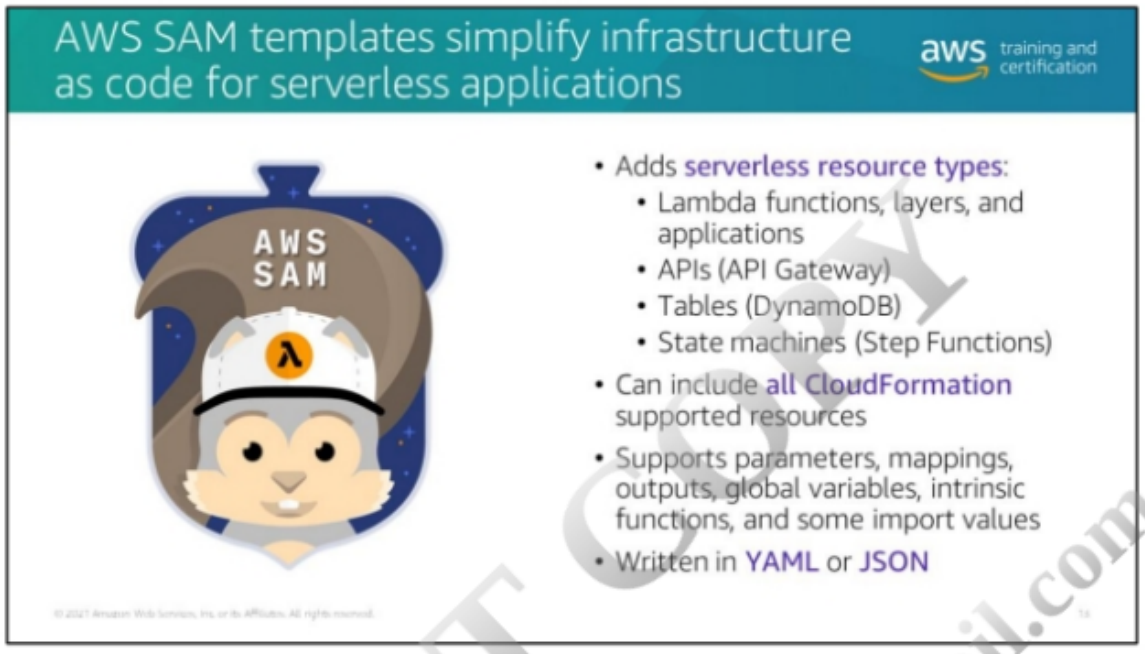
Each construct defines one or more concrete AWS resources, such as Amazon Simple Storage Service (Amazon S3) buckets and Lambda functions.

The AWS CDK Toolkit (also called the CLI command **cdk**) is a command-line tool for working with your AWS CDK applications and stacks. The toolkit provides the ability to convert one or more AWS CDK stacks to CloudFormation templates and related assets (a process called *synthesis*) and to deploy your stacks to an AWS account.



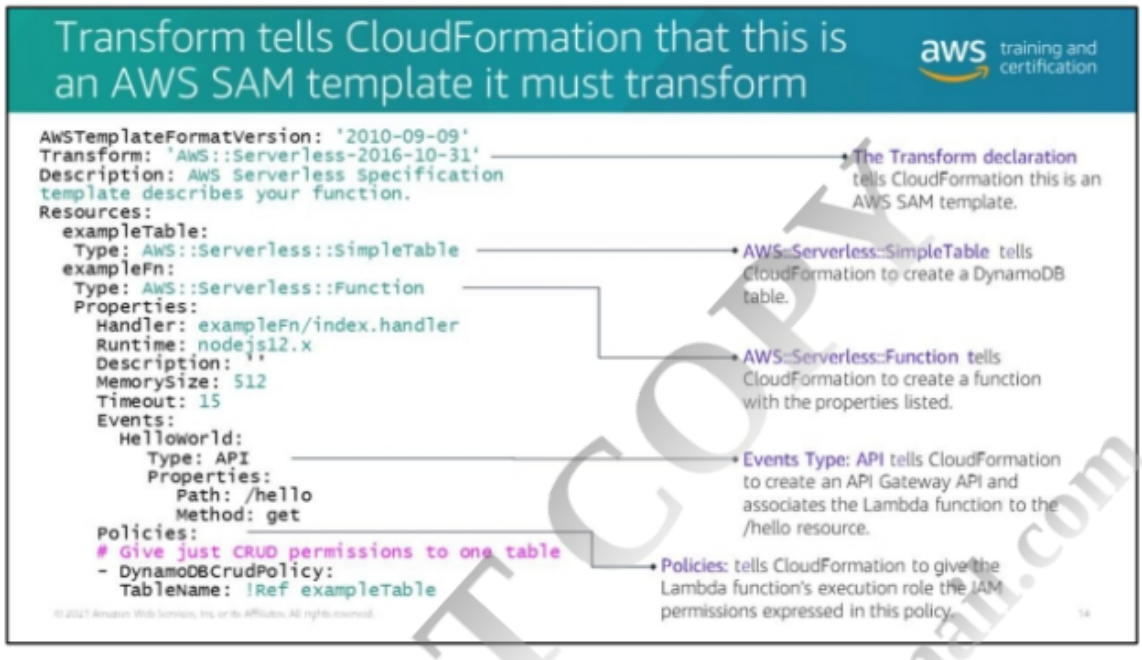
AWS SAM is an extension of CloudFormation optimized for serverless.

AWS SAM consists of AWS SAM templates and the AWS SAM CLI.



AWS SAM is an open-source framework for building serverless applications. It provides shorthand syntax to express functions, APIs, databases, and event source mappings.

For more information about the AWS SAM resource and property reference, see https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/sam-specification-resources-and-properties.html .



CloudFormation transforms the AWS SAM template and builds your stack on deploy

aws training and certification

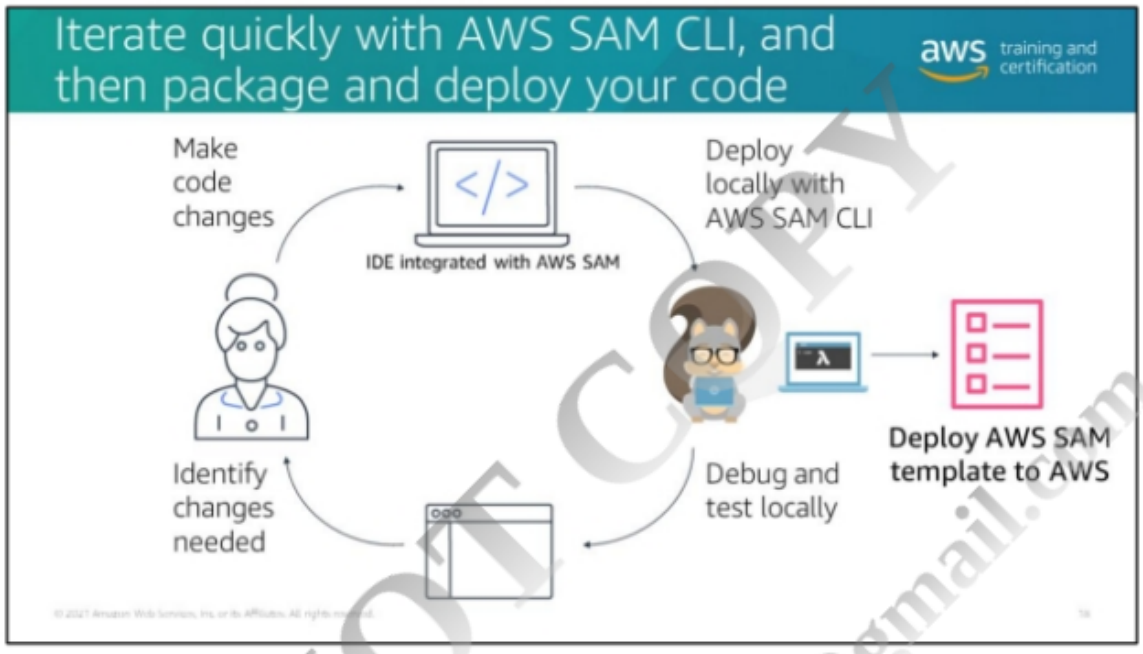
```
AWSTemplateFormatVersion: '2010-09-09'
Transform: 'AWS::Serverless-2016-10-31'
Description: AWS Serverless Specification
template describes your function.
Resources:
  exampleTable:
   Type: AWS::Serverless::SimpleTable
  exampleFn:
   Type: AWS::Serverless::Function
   Properties:
     Handler: exampleFn/index.handler
     Runtime: nodejs12.x
     Description: ''
     MemorySize: 512
     Timeout: 15
     Events:
      Helloworld:
          Type: API
       Properties:
        Path: /hello
        Method: get
     Policies:
     # Give just CRUD permissions to one table
     - DynamoDBCrudPolicy:
       TableName: !Ref exampleTable
```

Not all resources can be created in an AWS SAM template syntax, but you can use CloudFormation syntax to create or update other resources.

AWS SAM CLI makes it easier to code, debug, and test serverless applications locally

aws training and certification

- Supports API Gateway proxy-style and Lambda service API testing
- Makes response object and function logs available on your local machine
- Uses open-source docker-lambda images to mimic Lambda's invocation environment
- Can generate sample events
- Can tail production logs from CloudWatch Logs
- Can help build in native dependencies

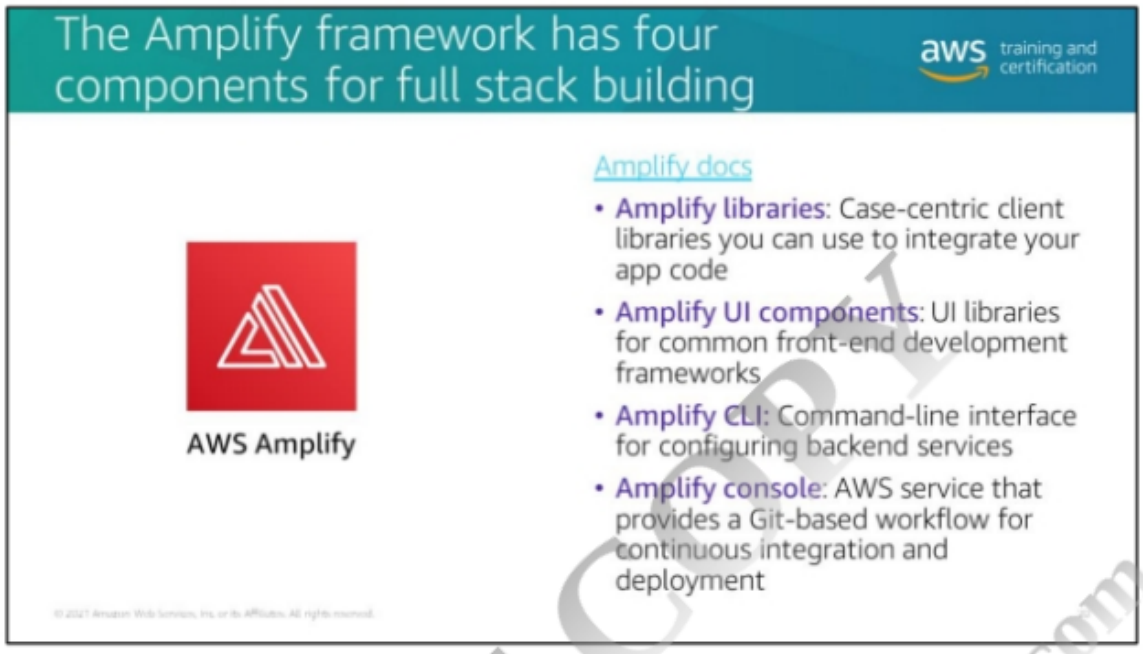


AWS SAM CLI commands let you run, package, build, and deploy serverless apps.

For more information about available commands, see the AWS SAM CLI command reference at https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/serverless-sam-cli-command-reference.html.

This link is also available in the OCS.

AWS Amplify is a set of purpose-built tools for developing, delivering, and managing scalable full stack applications built on popular web frameworks and mobile platforms.



You may find it helpful to follow the next part of the discussion by navigating to the Amplify documentation at https://docs.amplify.aws/start.

These components are available from the top-level menu.

Client libraries and UI components simplify client-side coding for your app. The following are examples:

- Authentication
  - Sign-in, sign-out, and forgot password
  - Retries
  - Rotating credentials
- Amplify DataStore
  - Offline and online scenarios
  - Model generation

Amplify CLI is category based with best practices built in to default options. The following are examples:

amplify add auth

CLI lets you choose a default configuration with or without a third-party federated login and configures the resources for you

amplify add api

CLI guides you through choices for GraphQL or REST API and sets up a default set of resources

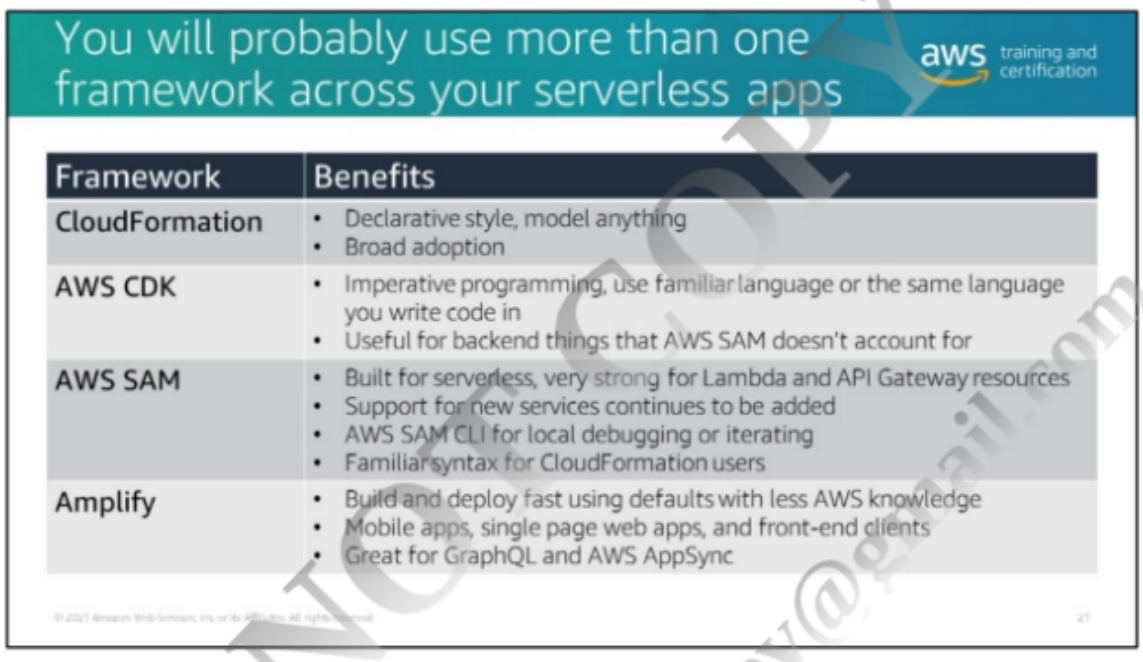
amplify mock api

Lets you do local mocking and testing

The Amplify console simplifies deployment of the front end and backend in a single workflow. The following are examples:

Feature branch deployments
Custom domains and global availability
Pull-request previews
End-to-end tests
Password-protected branches
Redirects

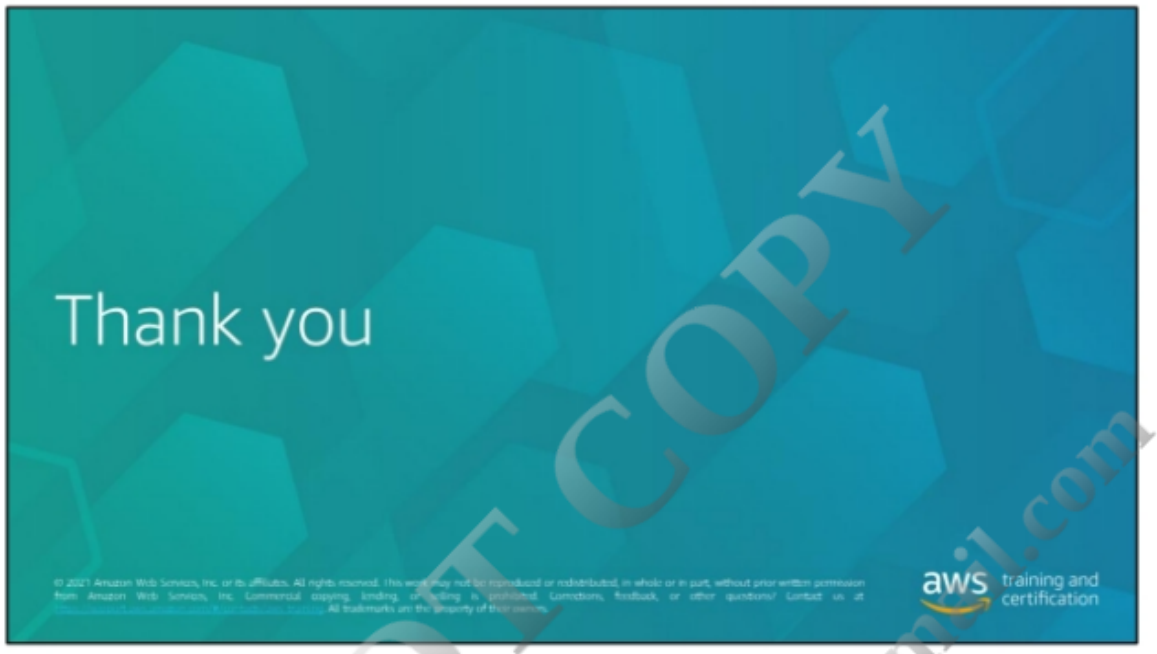
For more information about hosting your static website with continuous deployment using the Amplify console, see https://aws.amazon.com/amplify/hosting/.

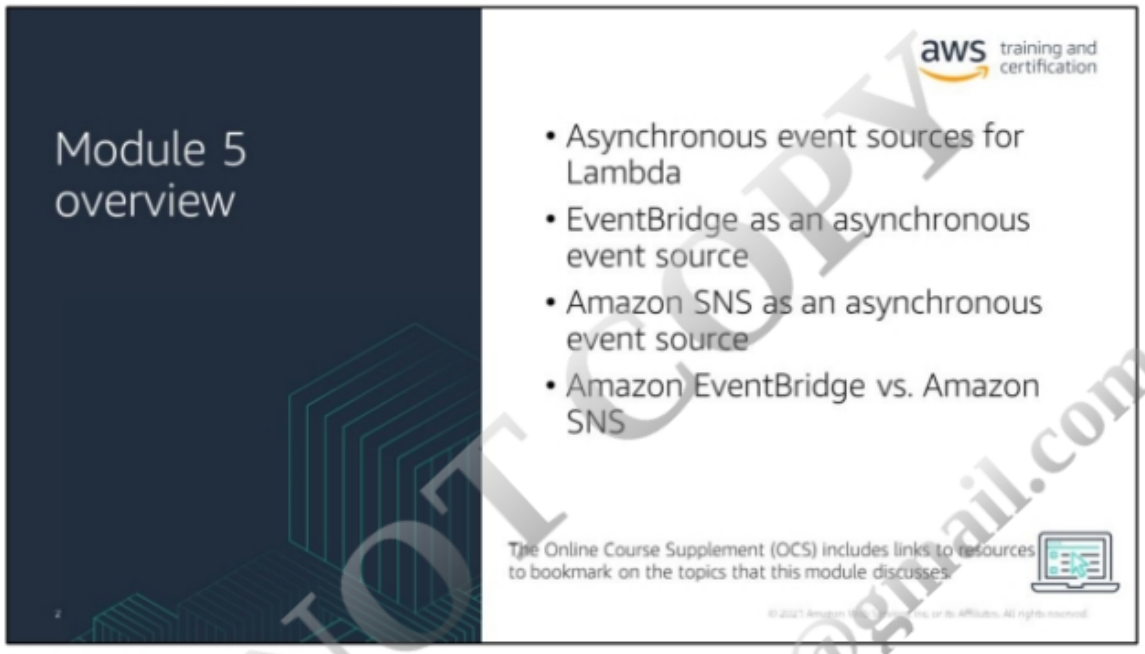


Each of the frameworks you've discussed has particular strengths to consider for your organization and application. You probably won't use a single framework for all aspects of your applications.

Features and capabilities also continue to be added.

As noted earlier, there are also third-party frameworks that may be a fit for your organization.





As noted earlier, an event source invokes AWS Lambda.

You can invokeLambda functions synchronously or asynchronously or for polling events through an event mapping.

When you choose an AWS service as an event source, the service determines how Lambda is invoked.

In contrast to a synchronous model, in an asynchronous design, the client sends a request and may get an acknowledgement that the event was received, but the client doesn't receive a response that includes the results of the request.

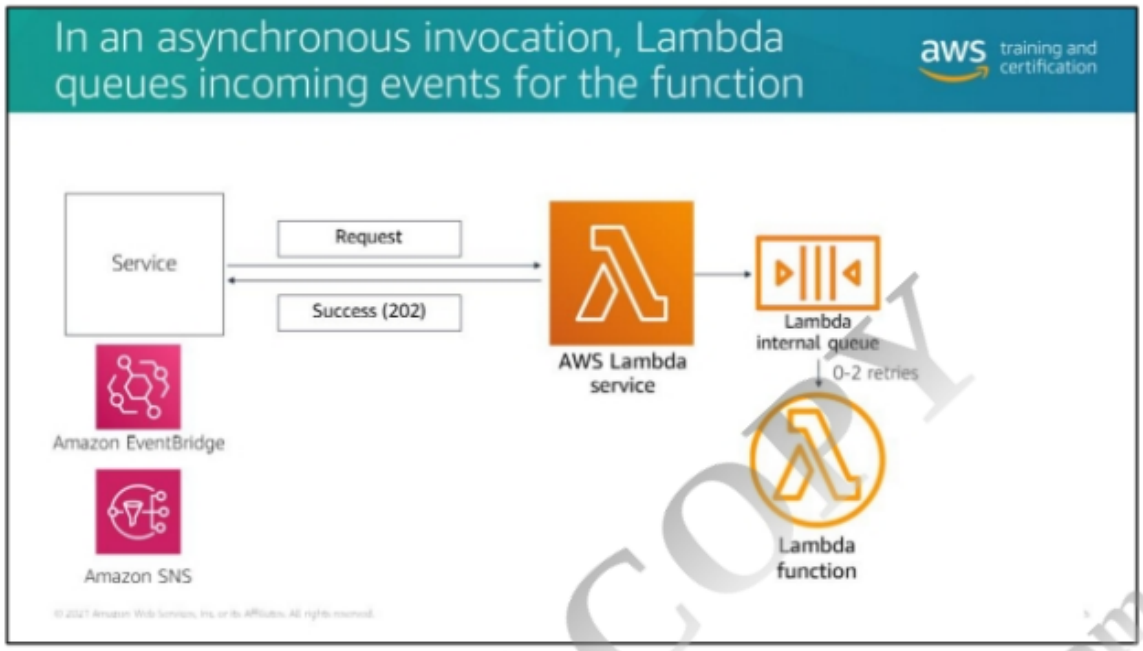
When the client makes a request to service A, service A surfaces the event to its consumers (in this example, service B) and moves on. The disadvantage is that there is no channel for service B to pass back information to service A. In a lot of cases, you really don't need that explicit coupling even though this may be how you are used to writing your code.

The advantage of this approach is that you reduce the dependencies on downstream activities, and this reduction improves responsiveness back to the client. This means you don't have to put logic into your code to deal with long wait times or to handle errors that might occur downstream. Once your client has successfully handed off the request, you can move on.

If you go into your design thinking in terms of asynchronous connections, you can create a much more flexible and resilient application.

In the diner example, an asynchronous event might be the customer ordering a slice of pie that the downstream service needs to prepare. The counter attendant acknowledges the order, generates an order number, and then tells the downstream services to make the pie associated with that order number.

When the pie is ready, a downstream method delivers the pie or lets the customer know that the order is ready, but it does not come back through the same channel.



Lambda has built-in support for asynchronous events.

In an asynchronous invocation, Lambda puts the event in its own internal event queue and returns a success response to the client. A separate internal process within the Lambda service reads events from the queue and sends them to your function.

If Lambda isn't able to add the event to the queue, the error message appears in the command output.

Lambda manages the internal event queue, and if your function returns an error, it will automatically retry the event two additional times over an increasing interval. After exhausting all retries, you can direct failed attempts to a dead-letter queue or send them to a Lambda on failure destination. You'll learn more about those in modules 7 and 8.

If the function can't be invoked because you don't have available concurrency, events could be waiting in the Lambda internal queue for hours or days.

Developers don't have control over a Lambda queue, but you can modify the retry count (0-2) and the maximum amount of time that Lambda should retry a failed item before either discarding it or sending it to an alternative destination.

Even if your function doesn't return an error, it's possible for it to receive the same event from Lambda multiple times because the queue itself is eventually consistent. If the function can't keep up with incoming events, events might also be deleted from the queue without being sent to the function.

Make sure that your function code gracefully handles duplicate events and that you have enough concurrency available to handle all invocations.

To invoke a function asynchronously with the AWS Command Line Interface (AWS CLI), use the invoke command with the invoke type of "Event."

For a list of AWS services that invoke Lambda asynchronously, see the section titled "Services that invoke Lambda functions asynchronously" at https://docs.aws.amazon.com/lambda/latest/dg/lambda-services.html.

For more details about asynchronous invocations, see the following information:

Lambda Developer Guide
- Asynchronous invocations: https://docs.aws.amazon.com/lambda/latest/dg/invocation-async.html
- API reference – Invoke: https://docs.aws.amazon.com/lambda/latest/dg/API_Invoke.html

These links are also available in the OCS.

In this course, you will review Amazon EventBridge and Amazon Simple Notification Service (Amazon SNS) as asynchronous event sources.

EventBridge and Amazon SNS are both asynchronous event sources for Lambda, and both are well-suited for initiating multiple parallel actions off of a single event. Both allow you to create filtering rules for subscribers.

In asynchronous communications, it's important that the client has a way to know when the downstream task is done so that it can complete the next steps. In the diner example, how does the customer or waiter know that their order is ready?

There are three ways this typically would happen:

- The customer checks for their order at the bakery counter every few minutes.
- The bakery gives the customer a device that vibrates when their order is ready, and the customer then collects their order.
- The customer receives a number to put on their table, and the waiter brings the customer's order to them when it's ready.

These equate to roughly three options for getting status to your client:

- The client polls for the status of the event.
- The client uses a webhook to get notified when the event is complete.
- The client uses WebSockets to open a two-way connection so that the results of the process can be delivered directly to the client.

The OCS provides detailed examples of how you might implement these patterns in a serverless application.

EventBridge is a serverless event bus that invokes Lambda functions asynchronously.

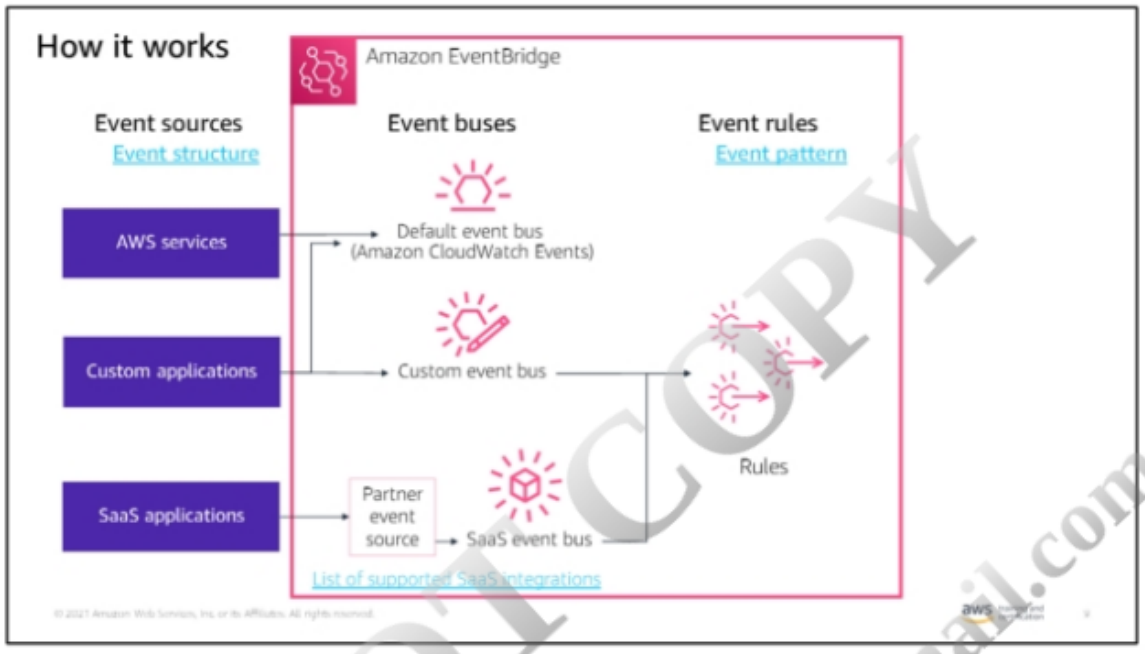
You can use EventBridge to connect your services and applications, and it can also consume data from integrated software as a service (SaaS) applications.

It extends its predecessor, Amazon CloudWatch Events.

EventBridge makes it easier to build event-driven applications

aws training and certification

Developer features	Security and scale features
• Sources including your own applications, AWS services, or SaaS applications • Routing rules to filter events • Event scheduling • Large number of targets • Schema registry/discovery to define event structures	• Fully managed and scalable • Near-real-time stream of system events regarding AWS resources • Integrated CloudWatch metrics • Integrated IAM support including resource policies • SaaS-integrated application events travel over the AWS internal network



Let's look a bit closer at the event sources in the "How it Works" diagram on https://aws.amazon.com/eventbridge/.

EventBridge can be used to route events from the following:

- AWS services
- Custom events you create within your application
- SaaS applications that have added EventBridge integration

Your event sources put events on to the bus you configure.

Every account has a default event bus. This is what Amazon CloudWatch Events uses and where events from AWS services are routed. You can put events from custom applications onto your default bus or create custom event buses.

SaaS applications use a partner event source connected to an SaaS event bus in order to route traffic from integrated SaaS applications. Each uses it own custom event bus.

EventBridge can ingest data from supported SaaS applications and then route it to AWS service targets as if an AWS service were intiating it. This creates a logical connection between SaaS and an AWS account without needing cross-account AWS Identity and Access Management (IAM) roles and credentials.

Traffic travels on the AWS internal network rather than the public internet.

For an example of EventBridge integration with Zendesk, see the following video: https://youtu.be/NakNmzsN6LI (11:01).

This video is part of the EventBridge playlist provided in the OCS.

Use these resources to find details about available SaaS integrations.

For more information about an overview of SaaS integrations, see https://aws.amazon.com/eventbridge/integrations/.

For more information about a list of all the available EventBridge integrations, see https://docs.aws.amazon.com/eventbridge/latest/userguide/create-partner-event-bus.html#eb-supported-integrations.

Events in EventBridge are represented as JSON objects. All events have a similar structure and the same top-level fields.

For more information about AWS events and details about the common fields included in all AWS event structures, see https://docs.aws.amazon.com/eventbridge/latest/userguide/aws-events.html.

You create rules to filter and route events that are put on to an event bus.

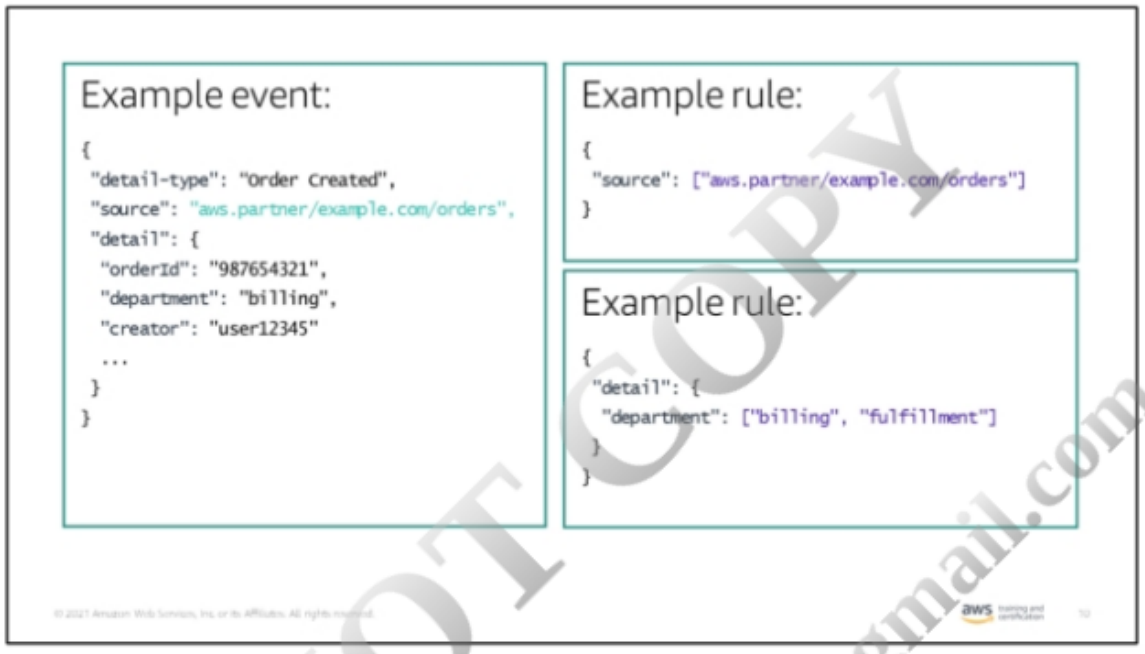
A rule is also expressed in JSON and uses event patterns to match AWS events on an event bus. When the pattern matches, the rules routes that event to a target.

Event patterns have the same structure as the events they match.

For more information about event patterns, see https://docs.aws.amazon.com/eventbridge/latest/userguide/filtering-examples-structure.html.

For more information about content-based filtering with event patterns, see https://docs.aws.amazon.com/eventbridge/latest/userguide/content-filtering-with-event-patterns.html.

These links are also available in the OCS.



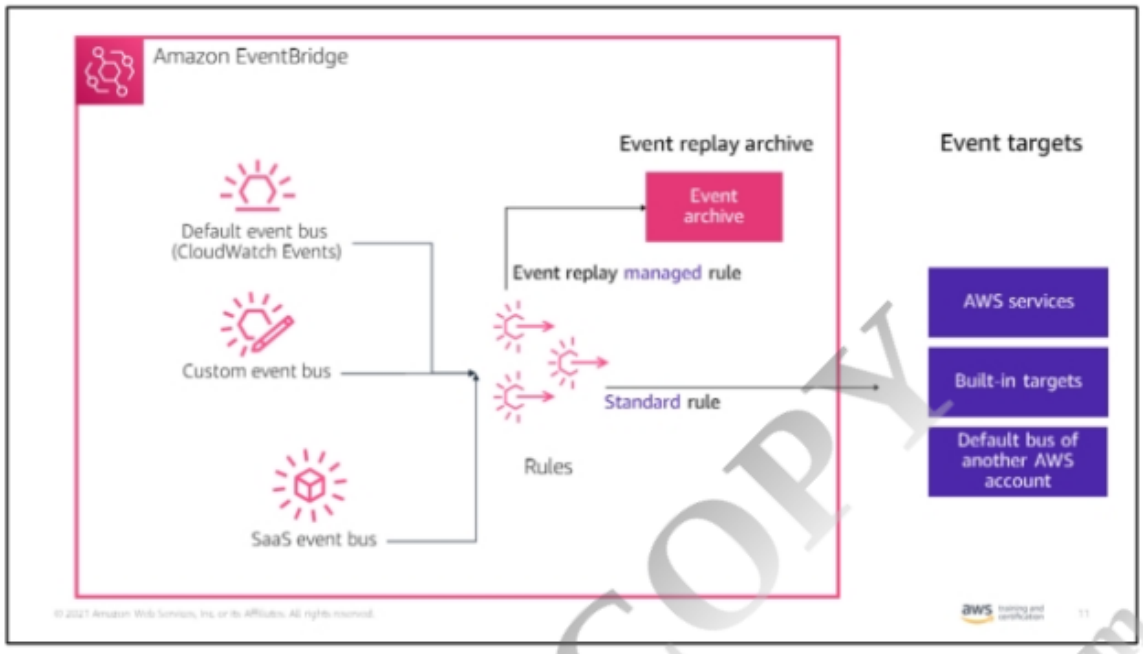
Here are a couple of quick examples.

The event always includes a detail type, a source, and details about the event.

You can create filters based on each of these components.

In this example, the first rule is looking for matches in the source field, and the second is looking for a specific element within the detail section of the event.

The first is a match because it matches the source. The second is a match because it matches one of the departments in the detail section of the rule.



You can associate multiple targets with each rule.

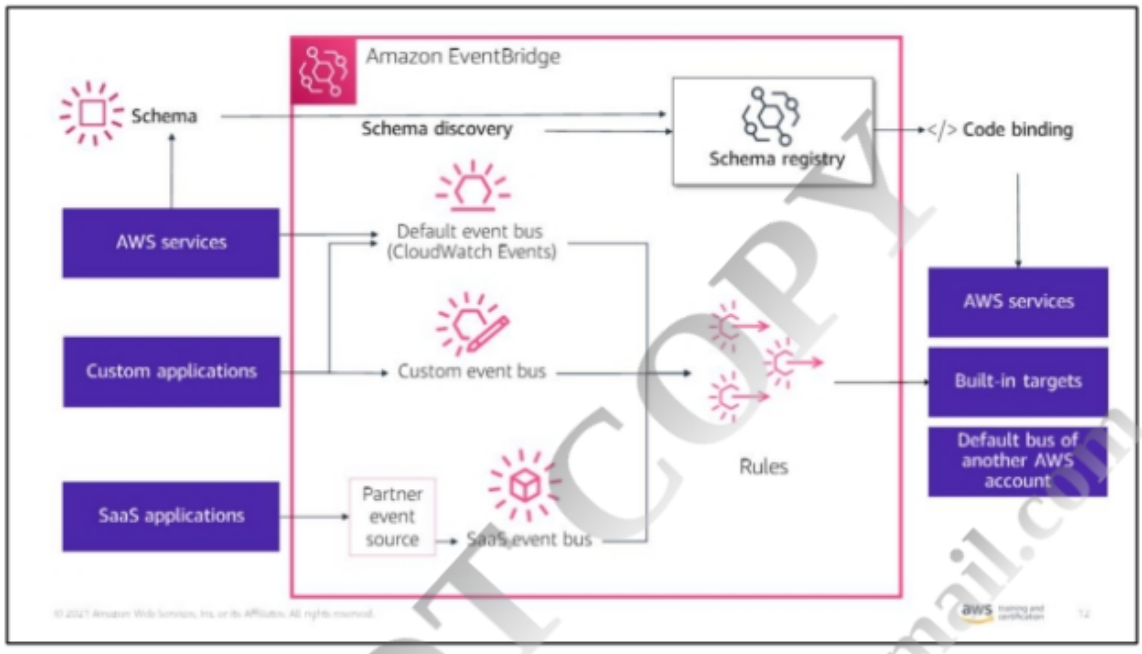
Going back to the diner example, the donut service, the invoice service, and the inventory service are all interested in a new order for donuts, so your donut rule is associated with each of those targets.

In event-driven architectures, it can be useful for services to access past events: for example, if you need to recover from a system error or when you want to validate new functionality.

Creating the mechanisms to do this can be complex. With EventBridge's archive and replay feature, you can now record any events that any type of event bus produces. Replay stores these recorded events in archives. You can choose to record all events or filter events to be archived by using the same event pattern matching logic used in rules.

For a blog post about using this feature, see https://aws.amazon.com/blogs/compute/archiving-and-replaying-events-with-amazon-eventbridge/.

For documentation about feature details, see https://docs.aws.amazon.com/eventbridge/latest/userguide/eb-archives.html.

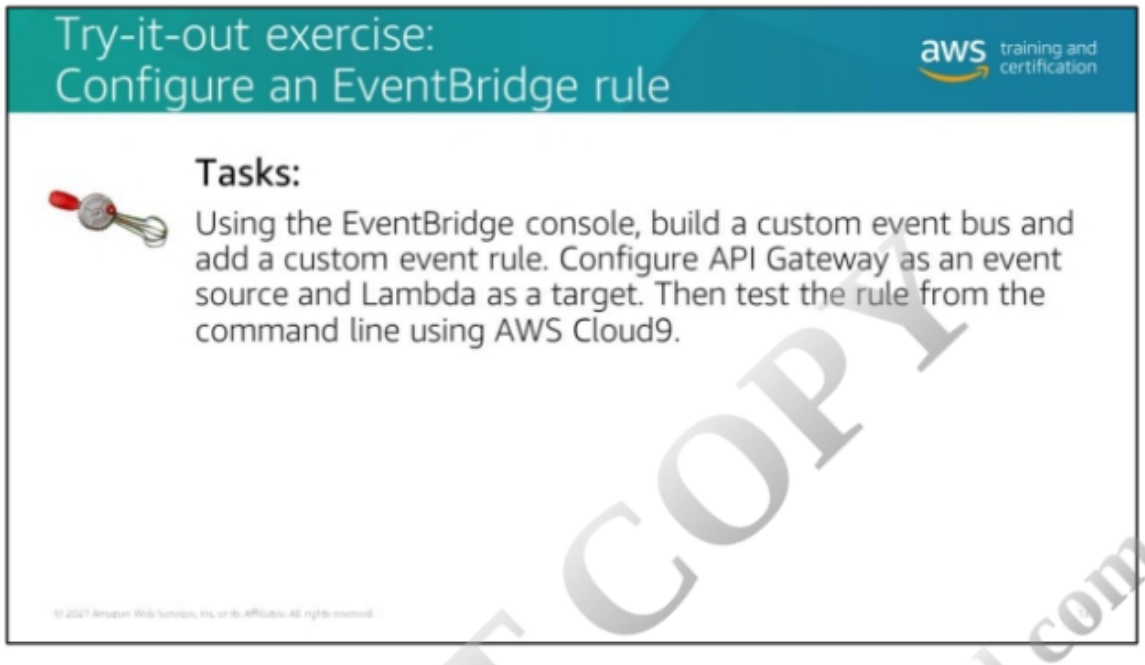


The option for so many sources can make it difficult to know what to expect when loading events.

The schema registry stores an event schema in a searchable registry and generates code bindings that you can use to represent the event as a strongly typed object in your code. This simplifies using events as objects in your code.

The EventBridge schema registry includes a schema generated by your organization's applications, AWS services, or SaaS applications.

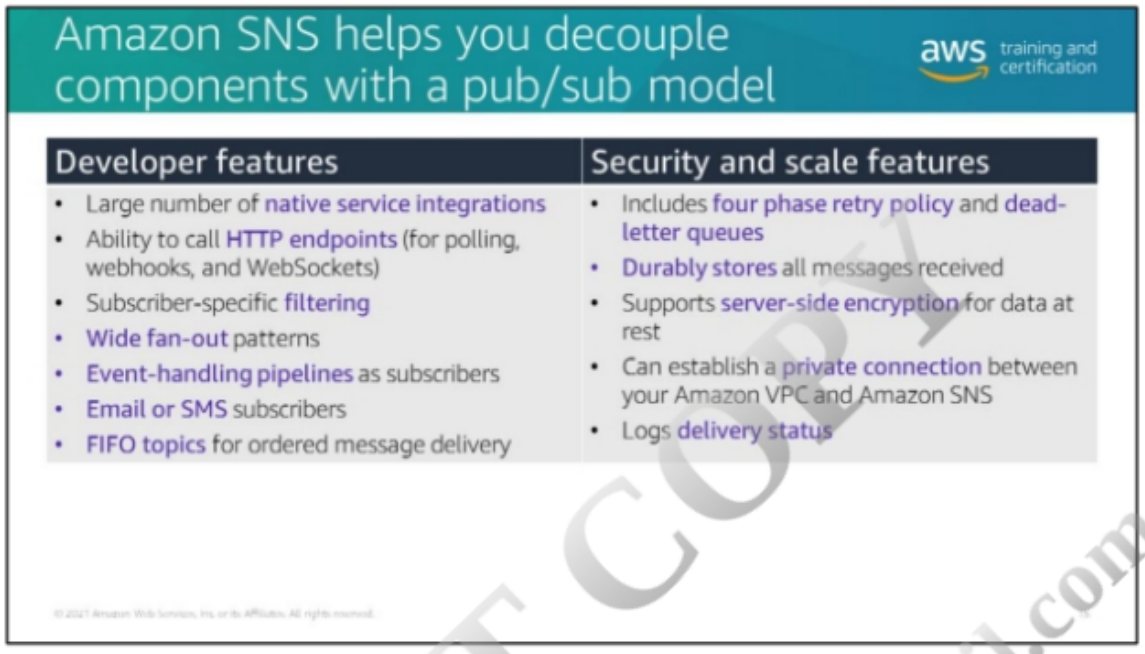
The schema registry gives you visibility in to all AWS service schemas. It also automatically discovers and adds schemas to the registry with schema discovery.

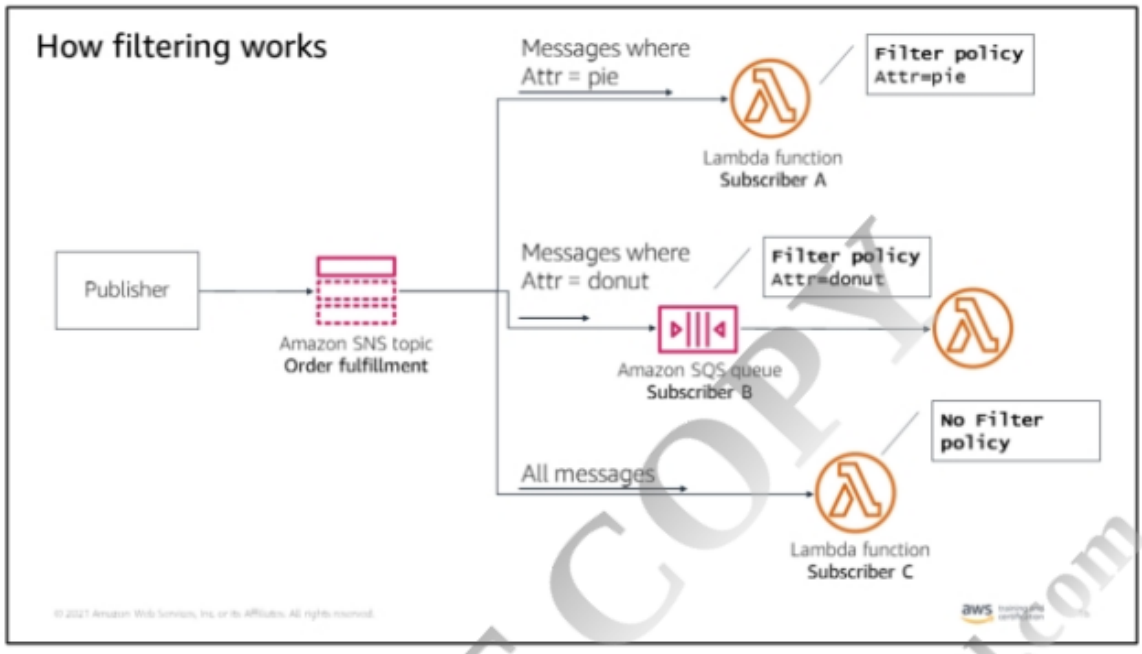


In this try-it-out (TIO) exercise, you set up an event bus that could be used to route events to different Lambda functions based on the type of snack indicated in the event. You set up a rule that routes events about the snack type of pie to a Lambda function called pieFunction. You add a POST route to the HTTP API you set up earlier and attach the integration for your custom EventBridge bus (serverless-bus). Finally, you test it via the command line in AWS Cloud9 and verify the results using CloudWatch Logs.

Amazon SNS is a highly available, durable, secure, fully managed pub/sub messaging service that enables you to decouple microservices, distributed systems, and serverless applications.

Amazon SNS invokes Lambda functions asynchronously.





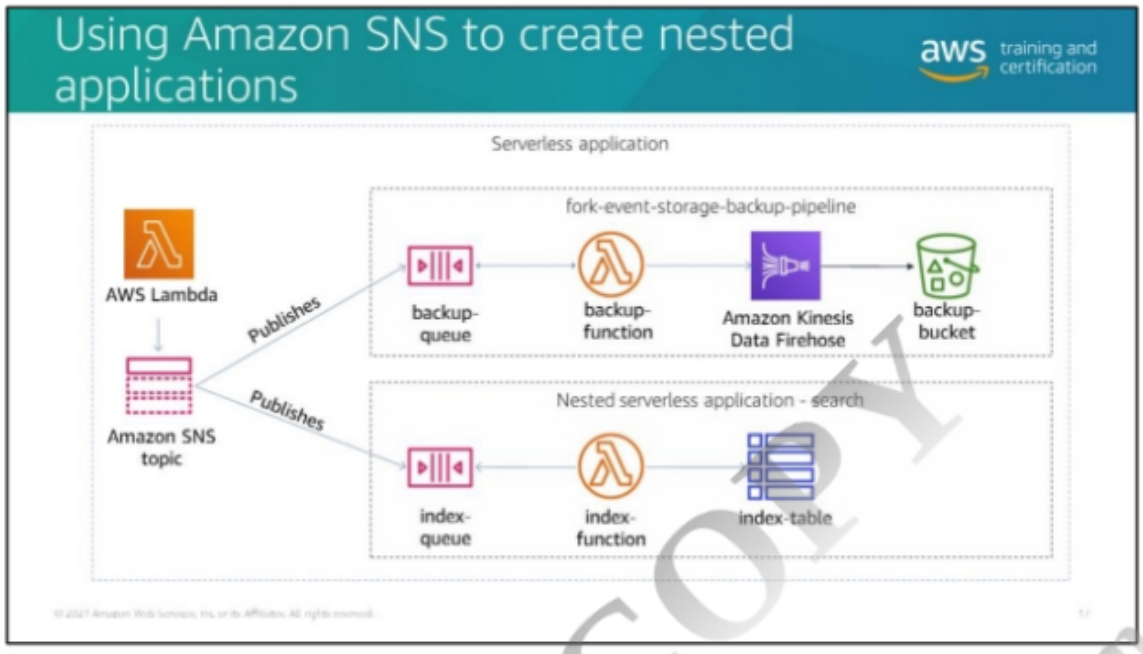
How it works

Using Amazon SNS for message filtering, a subscriber assigns a filter policy to the topic subscription. The publisher includes logic to produce the full context of the message for the topic and provides the attributes as key-value pairs along with the message payload.

The filter policy contains attributes that define which messages that subscriber receives, and Amazon SNS compares message attributes to the filter policy for each subscription.

Amazon SNS compares message attributes to the filter policy for each subscription and delivers only the messages that subscriber is interested in.

This means the publisher doesn't need to route messages to subscribers, and the subscribers don't have to filter out irrelevant messages. Amazon SNS makes sure each subscriber gets the messages they care about.

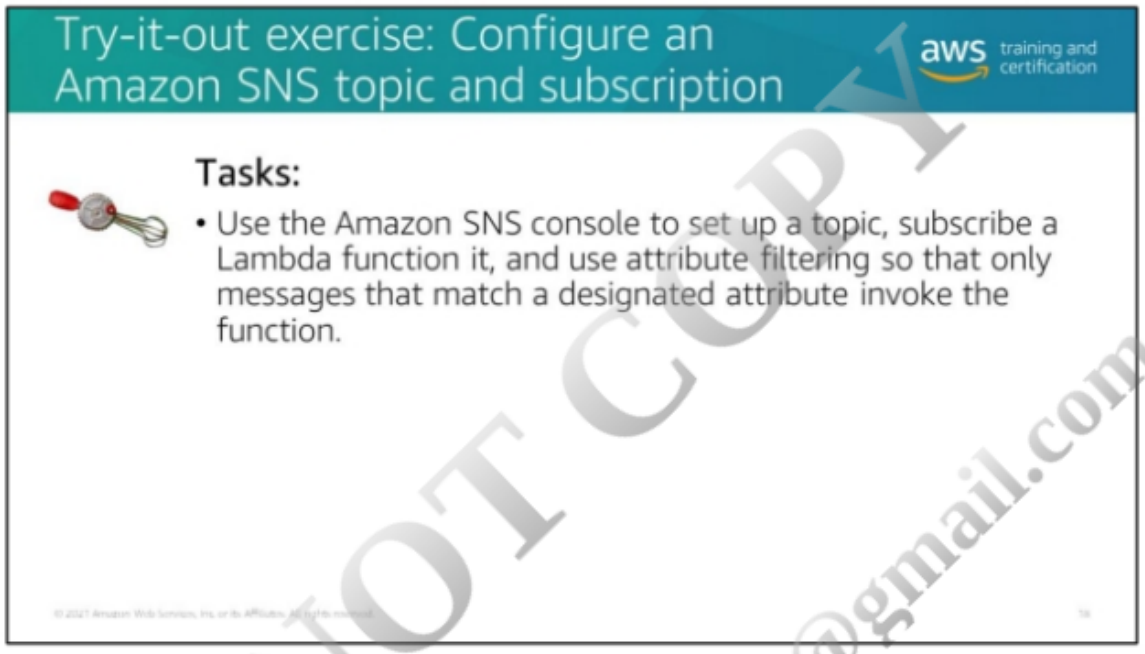


Amazon SNS is also a good way to fan out to nested applications within a larger application.

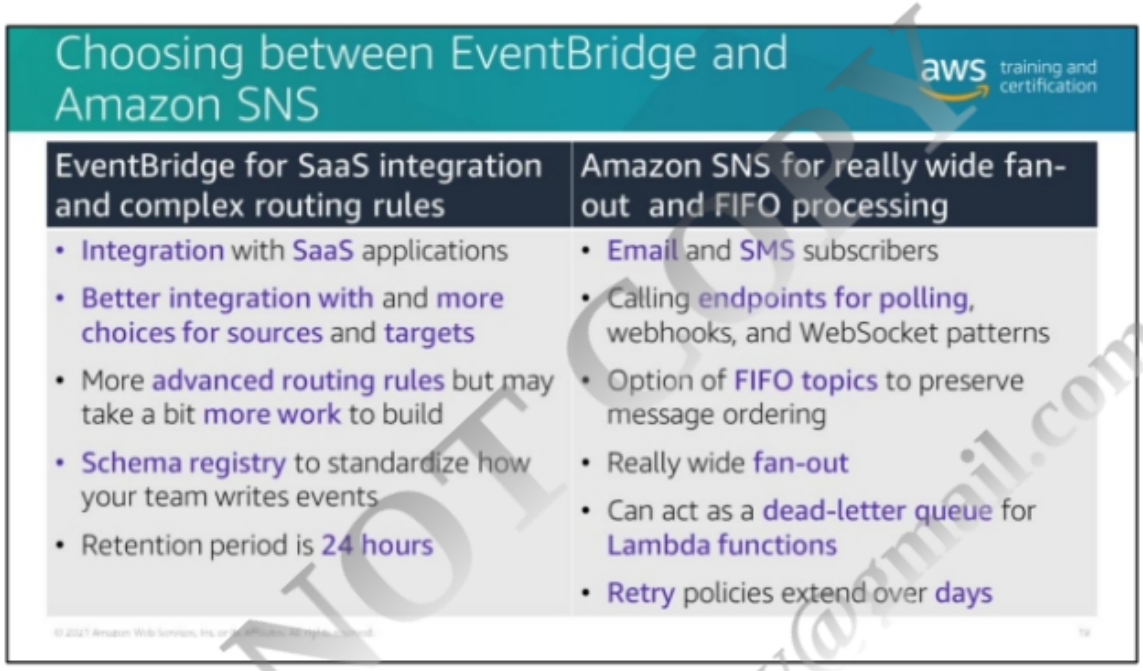
As patterns emerge in your application building, you can build reusable smaller applications and incorporate them as nested applications into your larger application.

The top application in this example is one of the AWS Event Fork Pipelines available through the AWS Serverless Application Repository.

For more information about AWS Event Fork Pipelines, see https://docs.aws.amazon.com/sns/latest/dg/sns-fork-pipeline-as-subscriber.html.



In this guided task, you set up an Amazon SNS topic that can be used to provide subscribers with snack event information. You'll subscribe the pieFunction to the topic and configure attribute filtering on the subscription so that only events that include the snack type of pie are delivered to the pieFunction subscriber. You'll use the Amazon SNS console to create test messages and test the subscription filter policy.



Both EventBridge and Amazon SNS provide an asynchronous buffer to upstream producers and can fan out to multiple consumers for parallel processing. As features are updated, there may not be a clear choice on which service is best suited, but you can always make a decision on a case-by-case basis.

EventBridge has native integration for more event sources and targets and an increasing number of SaaS integrations. Amazon SNS does not.

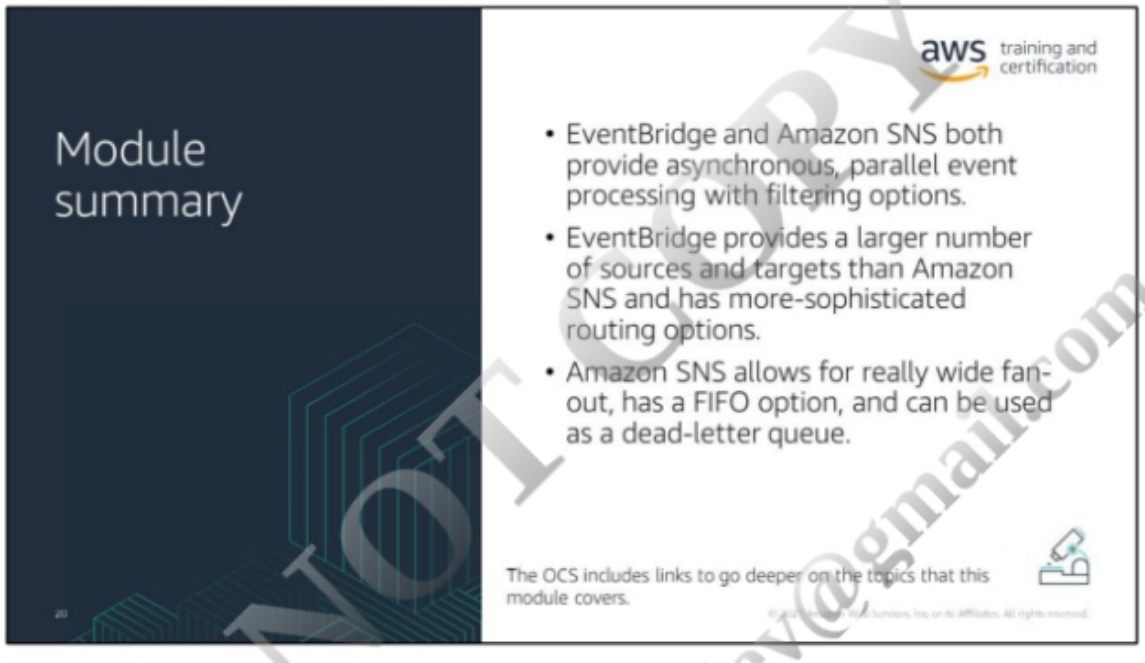
The rules you can create in EventBridge are more advanced, but it may be a little more work to set them up versus with Amazon SNS.

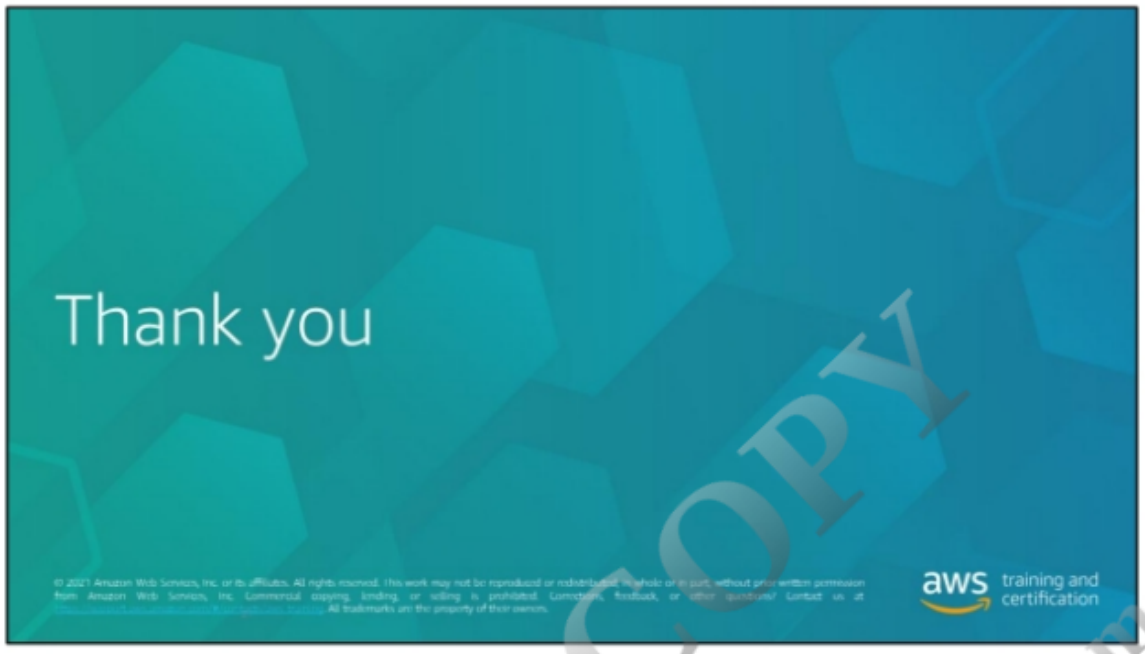
Amazon SNS is a good match for SMS subscriptions and for use in workflows that use polling (for example, with webhooks or WebSockets patterns).

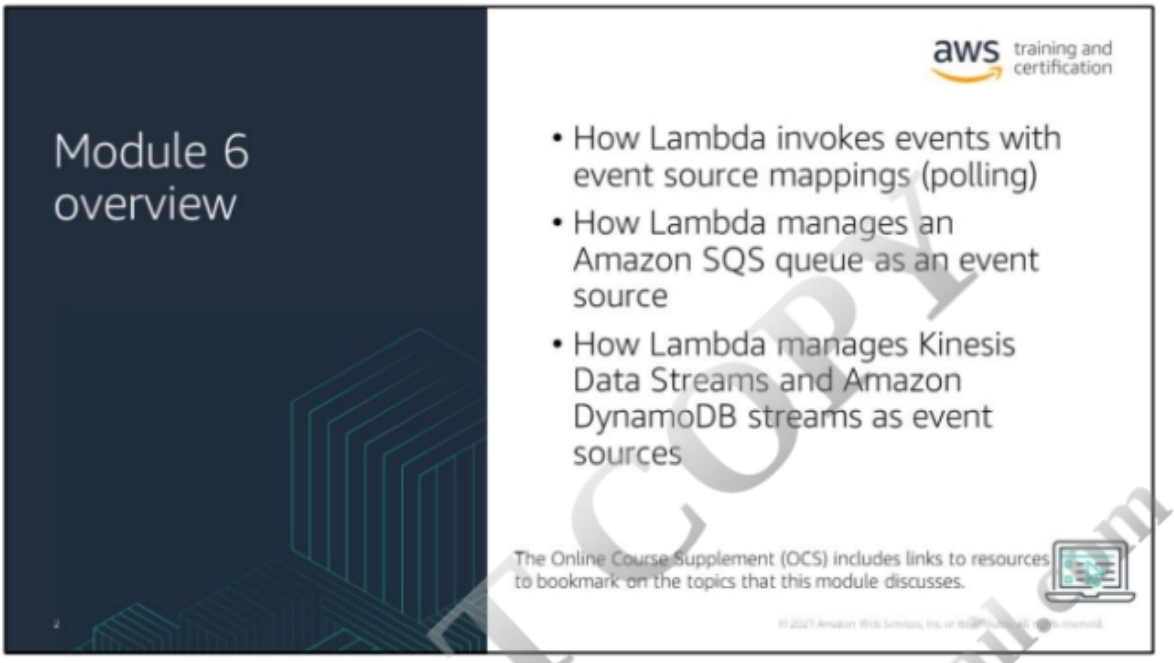
Amazon SNS FIFO topics are a good option when you need to preserve message ordering. For example, you might use Amazon SNS FIFO topics together with an Amazon Simple Queue Service (Amazon SQS) FIFO queue to perform tasks like inventory management at the lowest possible cost.

Amazon SNS allows more consumers and lower latency than EventBridge although both are designed for scaling asynchronous integrations.

Amazon SNS has an extended retry period with retry policies that you can configure for HTTP endpoints. This can be particularly useful if you're connecting to endpoints that are prone to outages or that exist on a less reliable infrastructure.

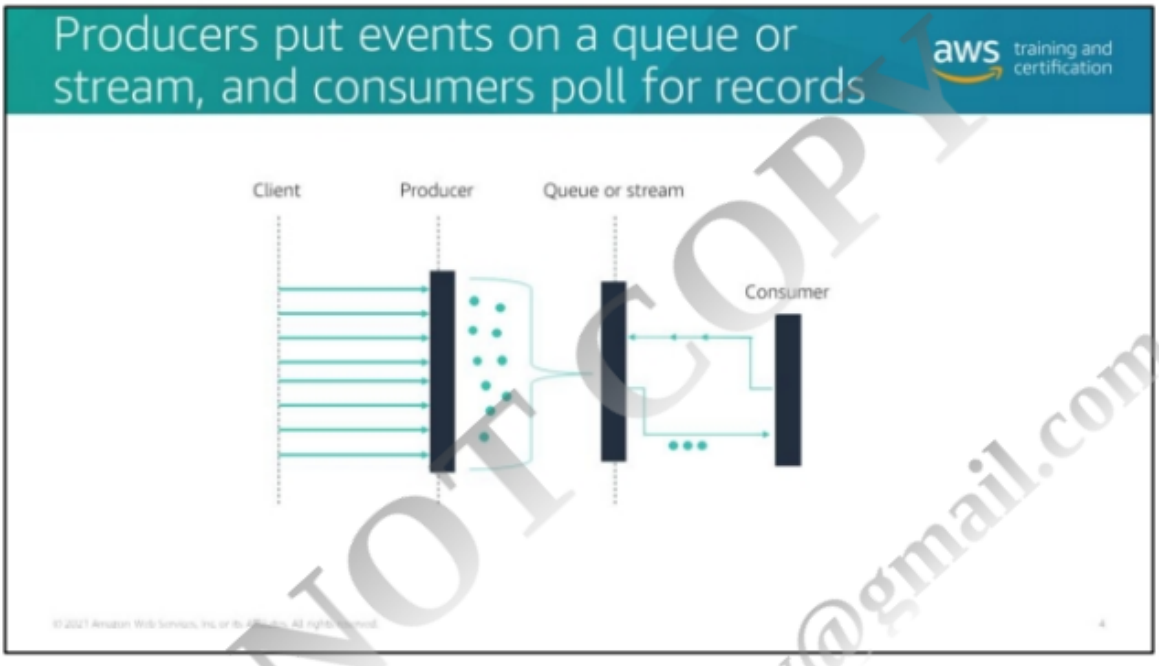






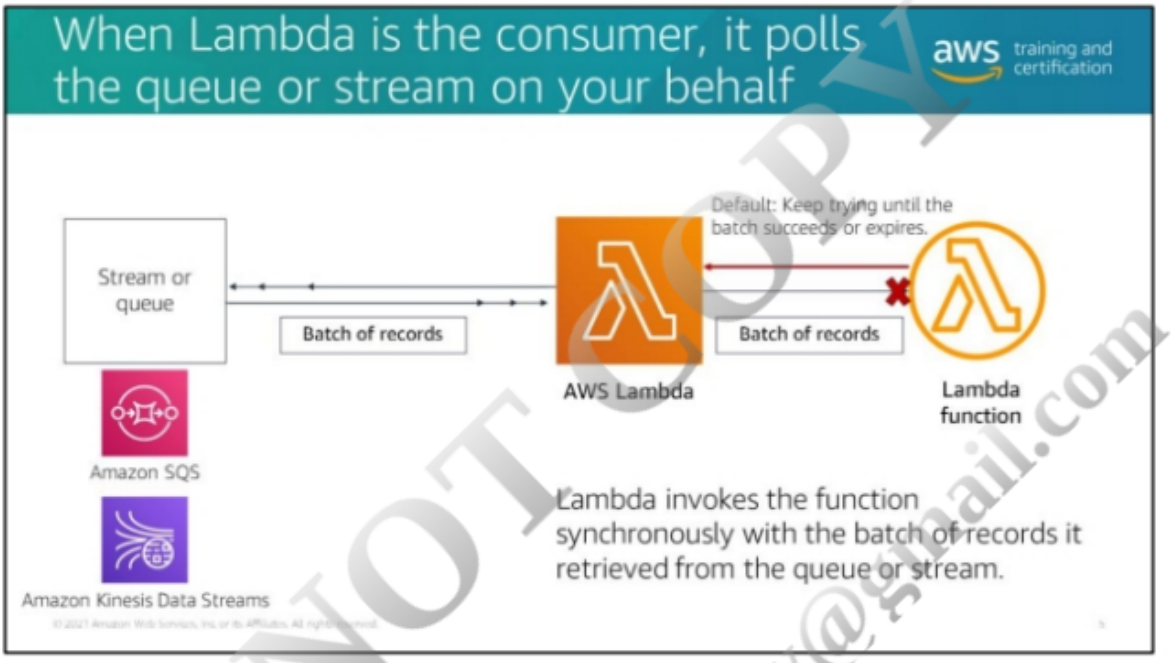
In this module, you'll look at how polling event sources work.

An event source mapping is the way that AWS Lambda invokes functions by polling a queue or stream as the event source.



In a polling event, consumers poll the producer for messages in batches and then process the batch before returning for a new batch of records.

Consumers poll the producer for messages in batches and then process the batch before returning for a new batch of records.



When you configure a queue or stream in front of Lambda, Lambda handles the work of polling it for you.

Lambda invokes your function synchronously with a batch of records it pulled off the queue or stream.

With both queues and streams, Lambda has built-in retry behaviors that you have some ability to configure, but Lambda is handling most of the integration work that you would otherwise need to do to set up a queue or stream consumer in your application.

By default, Lambda will keep retrying a batch of records until the batch succeeds or the records expires.

The Lambda integration for queues and streams has built-in error handling configurations that you should use, and you'll look at those in more detail in the Writing Good Lambda Functions module.

Queues vs. streams

aws training and certification

	Queues	Streams
Data value	• Value comes from processing individual messages.	• Value comes from aggregating messages to get actionable data.
Message rate	• The message rate is variable.	• The message rate is continuous and high volume.
Message processing	• Messages are intended for one consumer and are deleted after they are successfully processed.	• Messages are available to multiple consumers to process in parallel, and each consumer must maintain a pointer but does not delete records.
Default retry behavior	• Failed messages become visible again on the queue to be retried in a future batch. Processing continues with the next batch of messages.	• Failed messages block processing. A failed batch is retried until the messages succeed or expire.

Queues and streams are suited to different types of data patterns, and each processes the data a bit differently.

With queues:

- An individual message is the core entity where you apply some processing or compute. It could be a financial transaction or a command to control an Internet of Things (IoT) device.
- The occurrence of messages typically varies.

Messages must be deleted off the queue once they have been processed.

The queue's redrive policy controls how many retries a record will get, and you can connect a dead-letter queue to the Amazon Simple Queue Service (Amazon SQS) queue for records that exceed the max receive count you set.