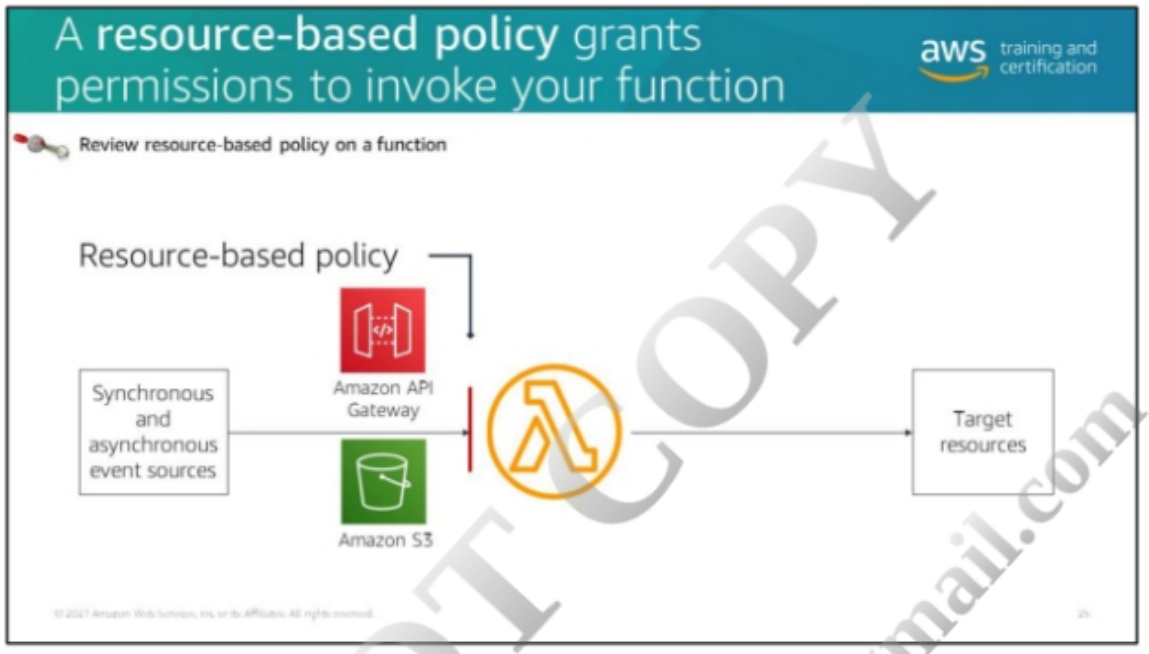
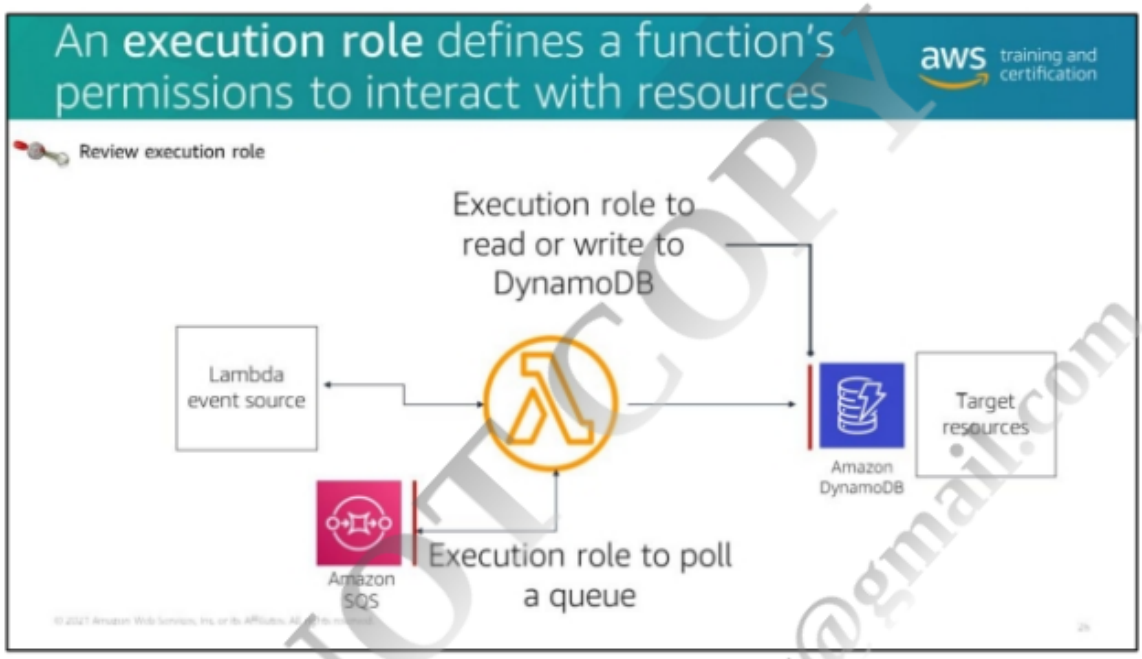


As you add triggers and targets to your function, you need to manage two key permissions: 1) permissions for your Lambda function to interact with other resources through an execution role and 2) permissions to invoke your function, which is dictated in the resource-based policy.

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

For services that trigger Lambda synchronously or asynchronously, when you add a trigger to your function with the Lambda console, the console updates the function's resource-based policy to allow the service to invoke it. Likewise, if you configure an integration with Lambda from another service console, the permissions to invoke the function are created for you. For example, if you create an API in Amazon API Gateway that targets a Lambda function, permissions to invoke that API are added when you create the API.

If you need to grant permissions to other accounts or services that aren't available in the Lambda console, you can use the AWS CLI to update the resource-based policy. You can also use AWS Identity and Access Management (IAM) roles to grant permissions to invoke. The key, as a developer, is that you need to provide permissions from source to function to trigger the function.

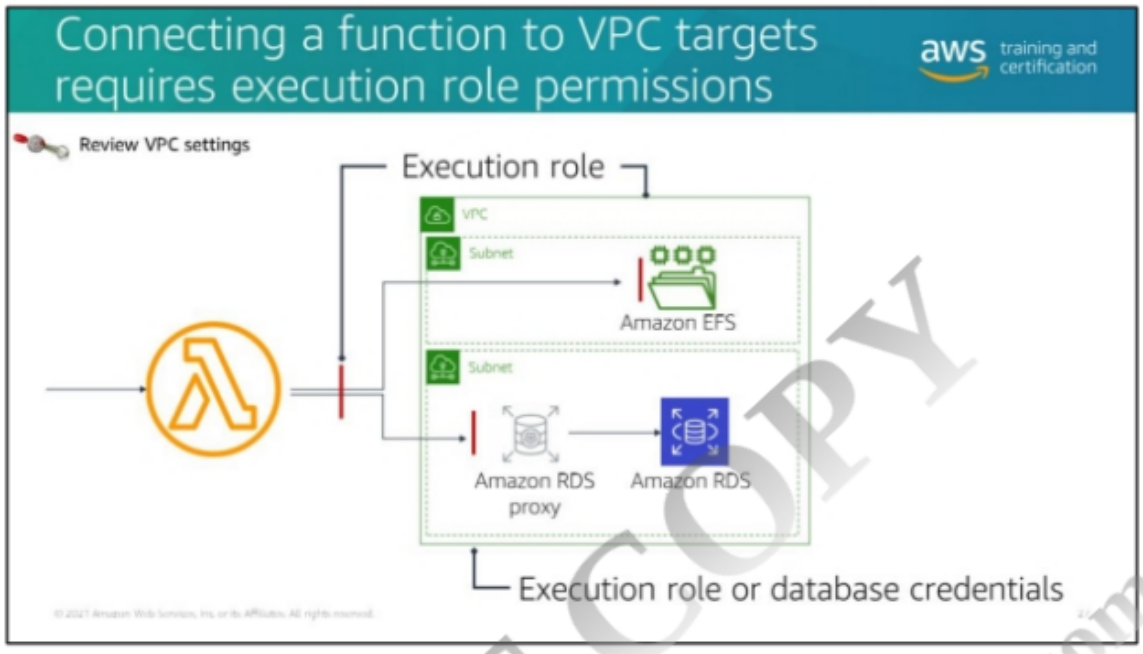


For more information about the Lambda execution role, visit https://docs.aws.amazon.com/lambda/latest/dg/lambda-intro-execution-role.html.

An execution role is an IAM role that Lambda has permissions to assume when you invoke a function. You need to select or create an execution role when you create the function, and you can modify the policies associated with the role via IAM. The default Lambda execution role includes permissions to write to Amazon CloudWatch Logs.

As you add additional targets (for example, writing to a database or sending an Amazon Simple Notification Service [Amazon SNS] topic), your Lambda function's execution role needs permissions to interact with those targets. The role needs permissions to write to a database or SNS topic, for example. IAM provides managed policies that you can add to your function to simplify getting the permissions that you need.

The execution role also comes into play if the event source that you are using is one that requires Lambda to retrieve messages from a queue or stream. In this case, although the stream or queue is the event source, Lambda actively polls the queue or stream to get batches of records. Lambda needs to interact with the source to delete records off of a queue or move the pointer on a stream. So, for these event source types, the execution role needs permissions related to the event source.



For more information about execution role and user permissions for VPCs, visit https://docs.aws.amazon.com/lambda/latest/dg/configuration-vpc.html#vpc-permissions.

To interact with resources that are in a VPC, in addition to execution role permissions to interact with the resource, you need to connect your function to the VPC. Your execution role permissions must include the ability to work with network interfaces. This is because, when you connect a function to a VPC, Lambda creates an elastic network interface for each combination of security group and subnet in your function's VPC configuration. A managed policy includes the permissions that you need.

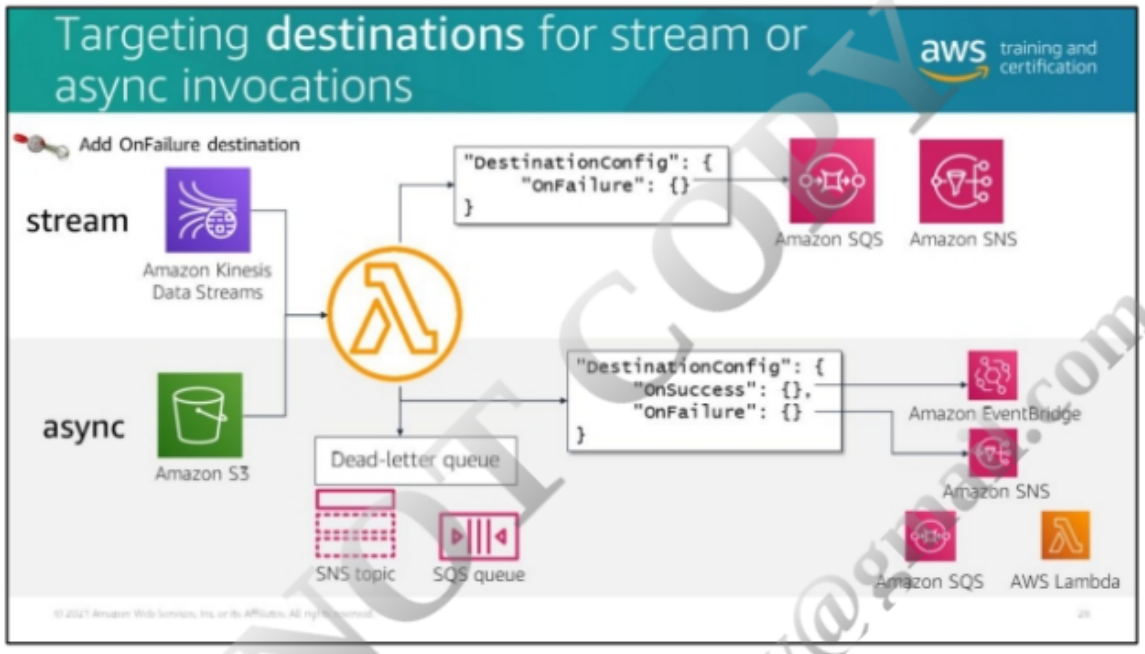
The IAM role of the user who is setting up the VPC connectivity for the function must have the permissions described on the VPC permissions page. For more information about configuring a Lambda function to access resources in a VPC, visit https://docs.aws.amazon.com/lambda/latest/dg/configuration-vpc.html.

The Lambda console has an option to connect to the VPC. Once that is set up, you can configure an Amazon EFS volume for the function, and you can configure the database proxy feature, which lets you create a pool of connections to an Amazon RDS instance. You won't be able to complete configuration of these options until you have connected the function to the appropriate VPC and subnet.

To use Amazon EFS, your function's execution role needs permissions to interact with Amazon EFS. For more information about configuring file system access for Lambda functions, visit https://docs.aws.amazon.com/lambda/latest/dg/configuration-filesystem.html.

To use an Amazon RDS proxy, you can set the authentication choice to execution role, and the Lambda console will add the necessary permissions to your execution role. Or, you can choose to use the same username and password that the function uses to connect to the database. The drawback of this second method is that you must expose the password to your function code, either by configuring it in a secure environment variable or by retrieving it from AWS Secrets Manager.

For more information about configuring an Amazon RDS database proxy to manage connections to an Amazon RDS instance, visit https://docs.aws.amazon.com/lambda/latest/dg/configuration-database.html and https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/rds-proxy.html.



A *destination* is an AWS resource that receives invocation records for a function. You must give a function permissions to the destination through the execution role. For more information about configuring destinations for asynchronous invocation, visit https://docs.aws.amazon.com/lambda/latest/dg/invocation-async.html#invocation-async-destinations.

Streaming event sources can use an OnFailure destination in conjunction with stream error-handling features to move failed records off of the stream and send them to services like Amazon SNS or Amazon SQS for handling offline.

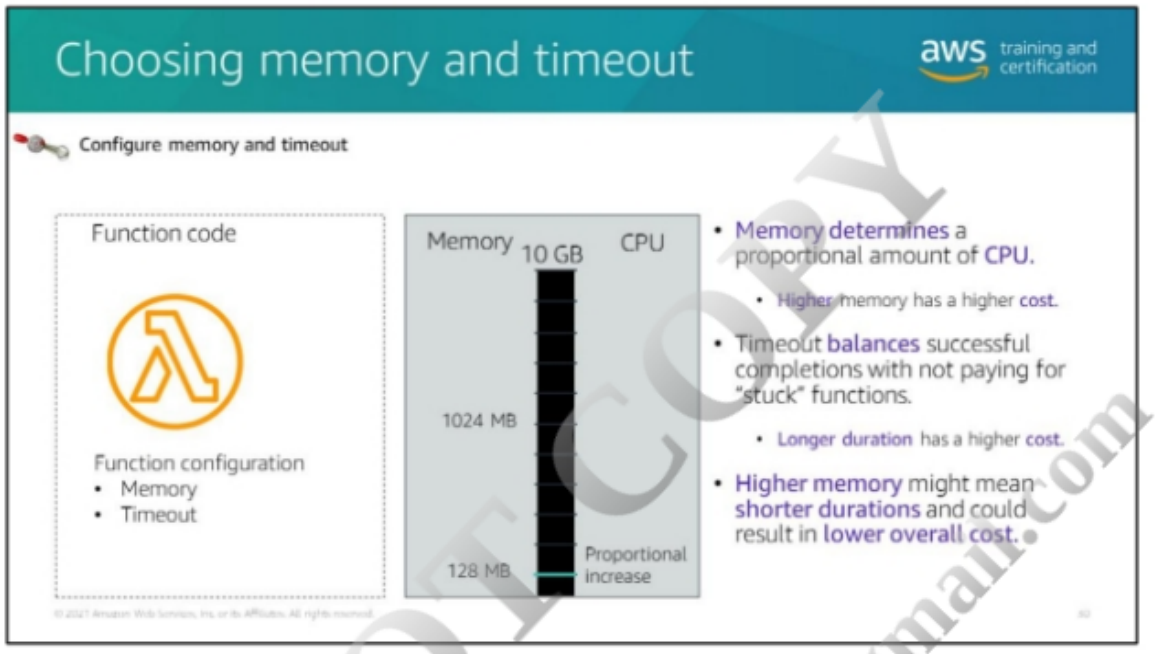
Asynchronous records can include both OnSuccess and OnFailure destinations. Use an OnSuccess destination to target other services like Amazon EventBridge or Amazon SNS, or another Lambda function. OnSuccess destinations receive invocation records if the record is processed successfully or completes without error.

OnFailure destinations will receive invocation records that fail all attempts at processing. You can configure services like Amazon EventBridge, Amazon SNS, and Amazon SQS as your OnFailure destination to handle the failed records.

As an alternative to an OnFailure destination, you can configure your function with a dead-letter queue to save discarded events for further processing. A dead-letter queue acts the same as an OnFailure destination in that it is used when an event fails all processing attempts or expires without being processed. However, a dead-letter queue is part of a function's version-specific configuration, so it is locked in when you publish a version. Another difference is that with dead-letter queues Lambda only sends the content of the event without response details. With destinations, Lambda sends the invocation record, which has details about both the request and response.

Your dead-letter queue can be an SNS topic or an SQS queue. The function's execution role must have permissions to send messages to the queue or publish messages to the topic.

Two important configurations for your function's performance are memory and timeout. The impact of these settings is discussed in more detail in the scaling module.

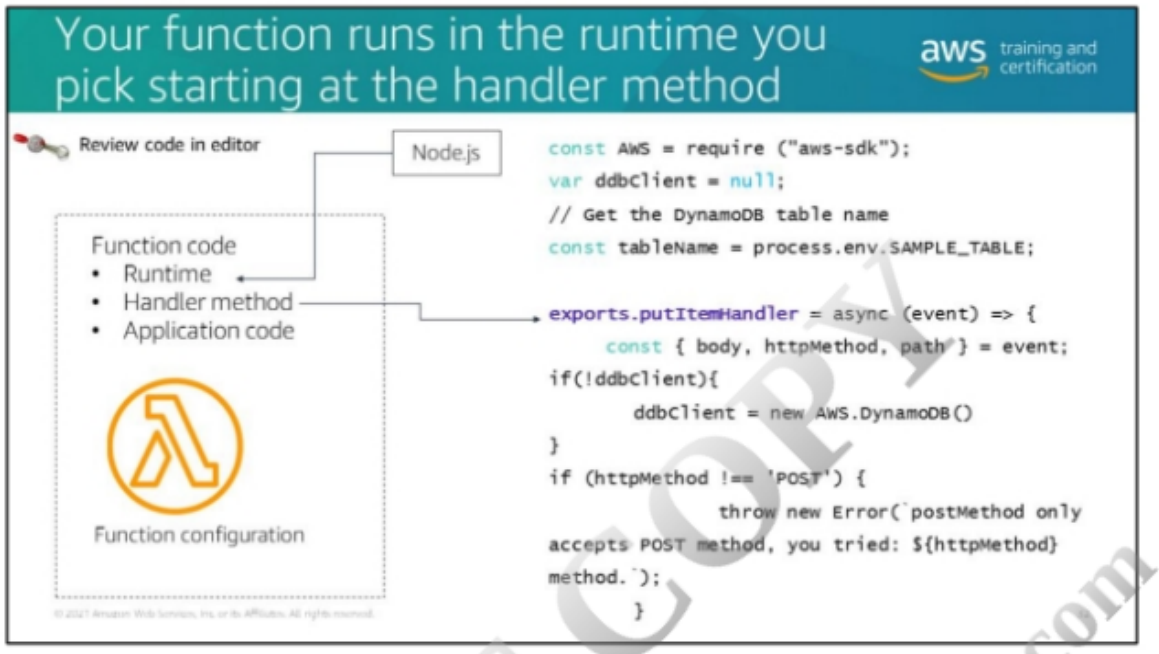


The memory setting within the basic settings on your Lambda function tells Lambda how much memory to provide to the function. The setting also determines how much CPU power your function will receive. The memory size is directly proportional to the CPU; for example, an environment with 256 MB of memory has double the capacity of one set for 128 MB.

Choose a timeout value that balances enough time for your function to complete versus limiting how long Lambda will wait on an invocation that stops responding. When a function reaches the timeout and has not completed, Lambda forcibly stops it.

Because function cost is tied to memory allocation and duration, you want to find the sweet spot that optimizes the function's performance for your scenario. We'll talk more about this in the scaling module.

In addition to the best practices related to the Lambda lifecycle, follow coding best practices in your functions.



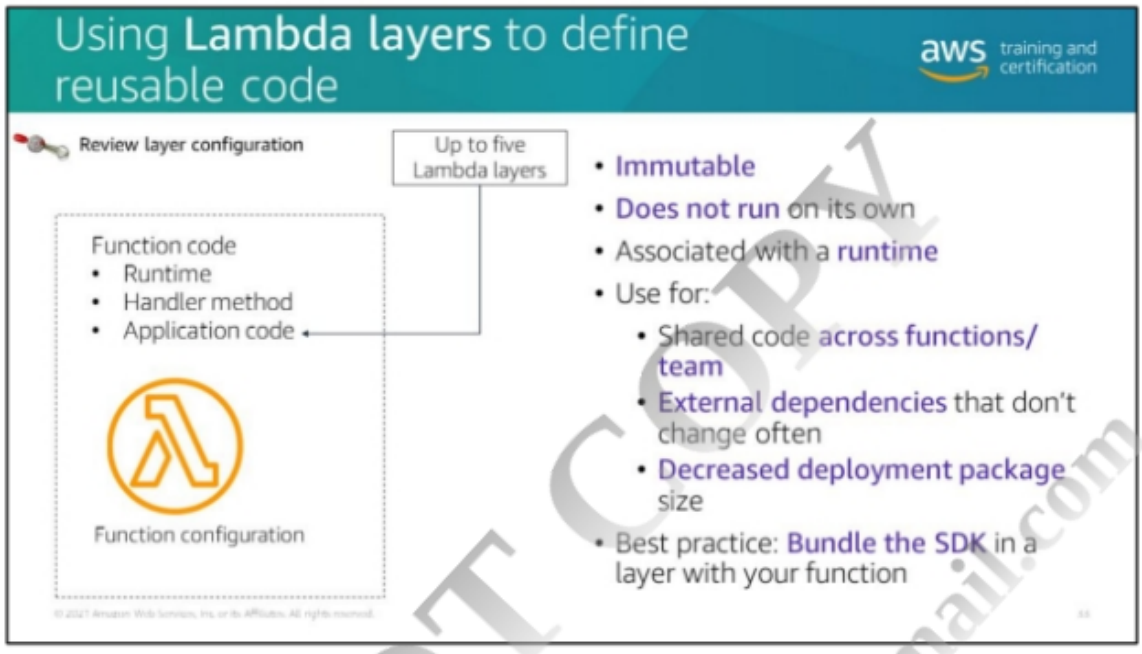
The function itself is where all of the application code lives. You select the runtime as part of configuring the function, and Lambda loads that runtime when initializing the environment.

For more information about Lambda runtimes, visit https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtimes.html.

For more information about Lambda sample applications, visit https://docs.aws.amazon.com/lambda/latest/dg/lambda-samples.html.

It is a best practice to separate the Lambda Handler from your business logic.

You can see an example of this in the **error-processor-randomerror** function. The handler consists of a single line of code where **exports.handler = myFunction**. All of the other logic happens in the function defined by the line that starts with **var myFunction = async function(event, context).**

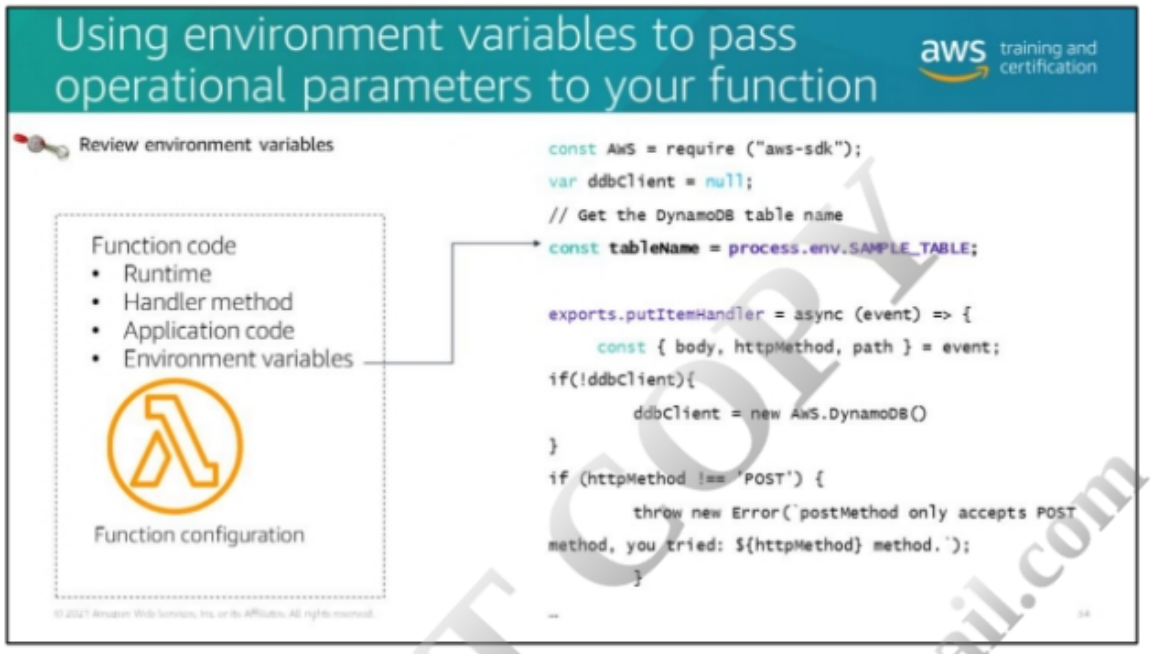


Lambda layers were explicitly designed for reuse of application code. You can use a layer that matches the runtime of your function to attach reusable modules or libraries.

A function can have a limited number of layers associated with it, but layers can be used across any of your functions. You need permissions to view and use the layer if it is not public. You can't run a layer on its own.

A layer reduces your actual Lambda function deployment size, which speeds up your CI/CD cycle. For example, you might be able to reduce the size of a Lambda function from 10 MB to 2 MB. You can specify multiple runtimes for a layer. Note, the runtime is a tag rather than any check on the type of code that you include. For example, if you indicate that the runtime is Node.js, but it contains Python code, that layer will be available to your Python functions. However, the layer would fail when the function tries to run it.

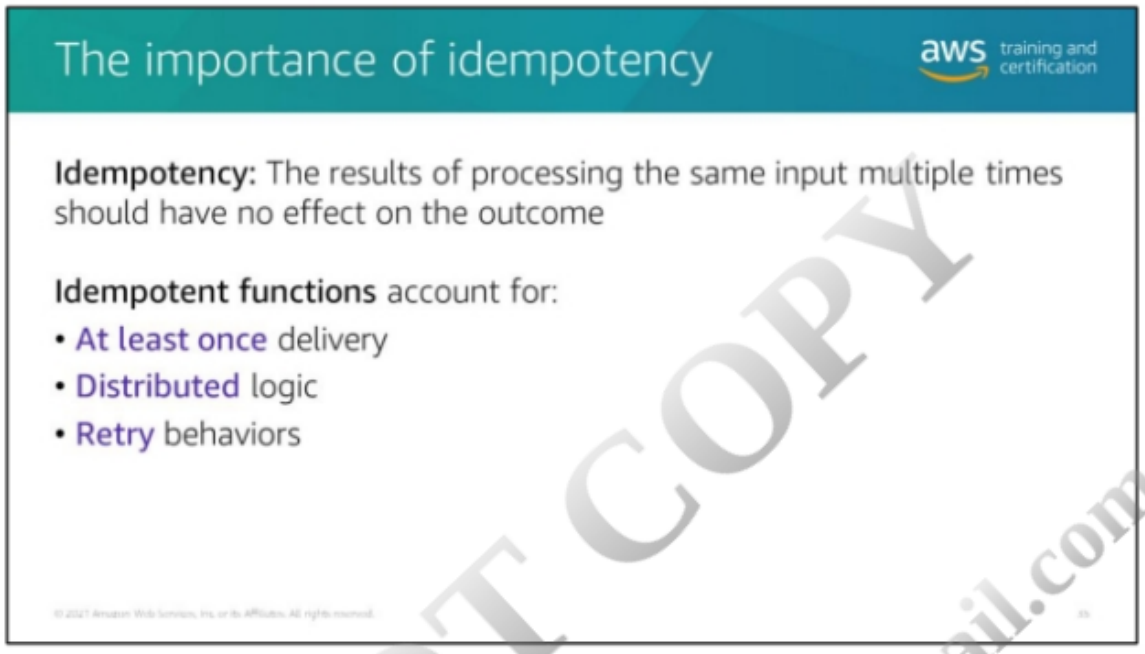
For more information about working with Lambda and Lambda layers in AWS SAM, visit https://aws.amazon.com/blogs/compute/working-with-aws-lambda-and-lambda-layers-in-aws-sam/.



For information about using Lambda environment variables, visit https://docs.aws.amazon.com/lambda/latest/dg/configuration-envvars.html.

You can use environment variables to store secrets securely and adjust your function's behavior without updating code. An environment variable is a pair of strings that are stored in a function's version-specific configuration. The underlying invocation environment for the runtime provides additional libraries as well as environment variables that you can access from your function code. You can also create your own environment variables to use in your code.

When you publish a version, the environment variables are locked for that version along with other version-specific configurations.

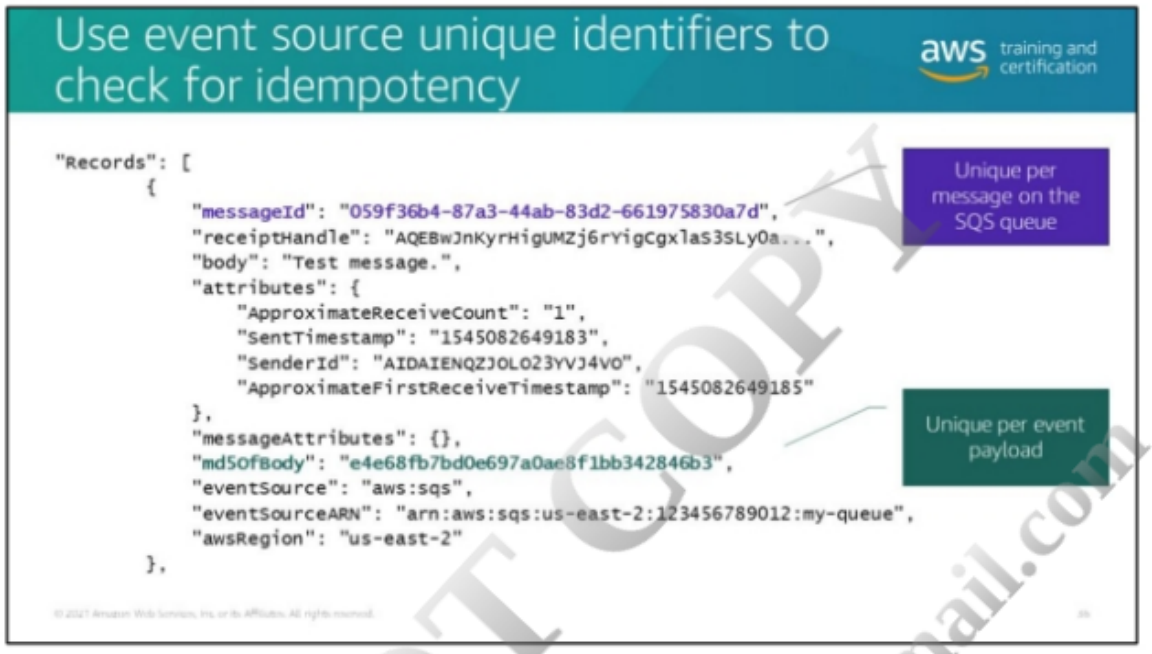


For your code to be *idempotent*, the results of processing the same input multiple times should have no effect on the outcome after it runs the first time with that input.

There are several reasons why this is particularly important with your serverless applications:

1. Many event sources provide at-least-once delivery. This means that a record may be delivered more than once. For example, both Amazon SNS and Amazon SQS have at-least-once delivery. When Lambda uses its internal queue to process asynchronous requests, the function could get the same request twice.
2. With distributed logic and distributed data stores, there's a higher likelihood that you could have a scenario where one of your functions receives the same request more than once.
3. Retry behaviors that deal with batches of records (with Amazon Kinesis Data Streams or Amazon SQS event sources, for example) include sending the same records to the Lambda function more than once. We'll talk about what that looks like later in this module.

Your function code should recognize a duplicate record and only process it the first time. For example, you would add code into your handler that checks a unique value in the incoming event and ignores it if it already exists in the database.



Idempotent functions account for:

- At-least-once delivery
- Distributed logic
- Retry behaviors

Every event source has some unique ID. The *AWS Lambda Developer* Guide has example events for each Lambda event source:
https://docs.aws.amazon.com/lambda/latest/dg/lambda-services.html

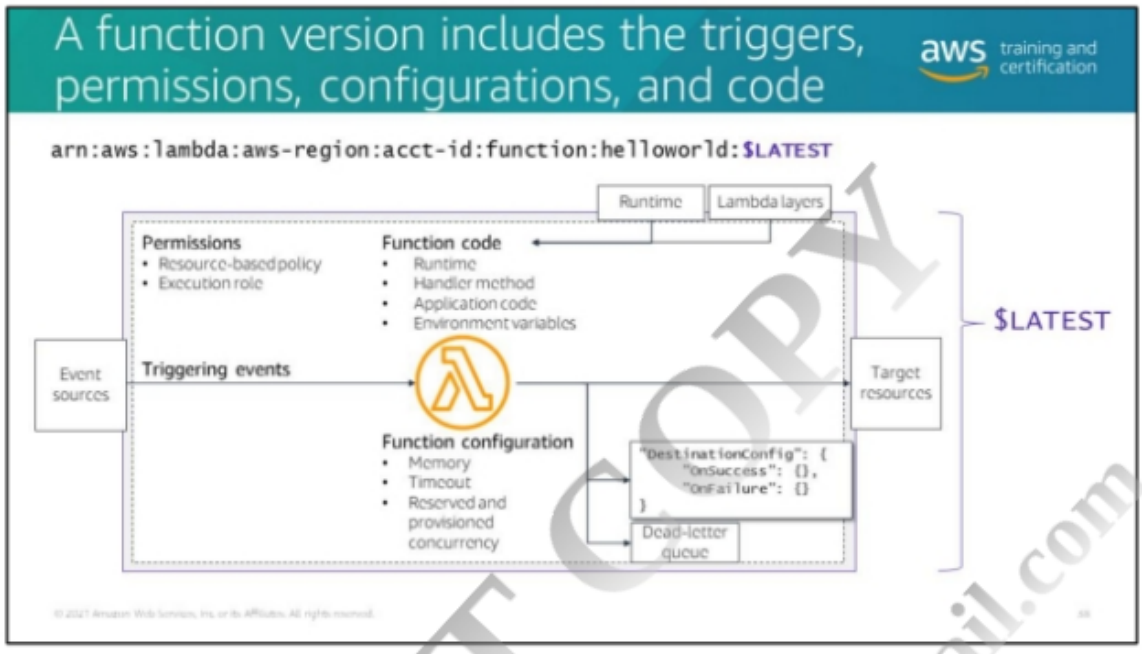
For example, https://docs.aws.amazon.com/lambda/latest/dg/with-sqs.html has an example of an SQS standard queue event (excerpted on the slide).

MessageID is a unique identifier that Amazon SQS generates. If you want to verify the uniqueness of the record on the SQS queue, you could use the messageID attribute.

Another scenario might be that you actually need to verify the uniqueness of the payload. For example, if an upstream producer sent the same record to Amazon SQS twice, each of those events would have a unique messageID in Amazon SQS but would really be the same payload from your application's perspective. You can avoid processing the same payload twice using the **md5OfBody** attribute in the Amazon SQS event. This field represents a unique hash of the payload.

First-In-First-Out (FIFO) queues have additional attributes in the event that are related to deduplication and sequencing to maintain record order and provide one-time-only delivery. You should still include code to prevent processing the same payload twice.

Likewise, although FIFO SNS topics and all Kinesis data streams are designed to prevent duplicate records from being processed, scenarios exist that could create a situation where your Lambda function receives the same record. Always check for uniqueness of the event.

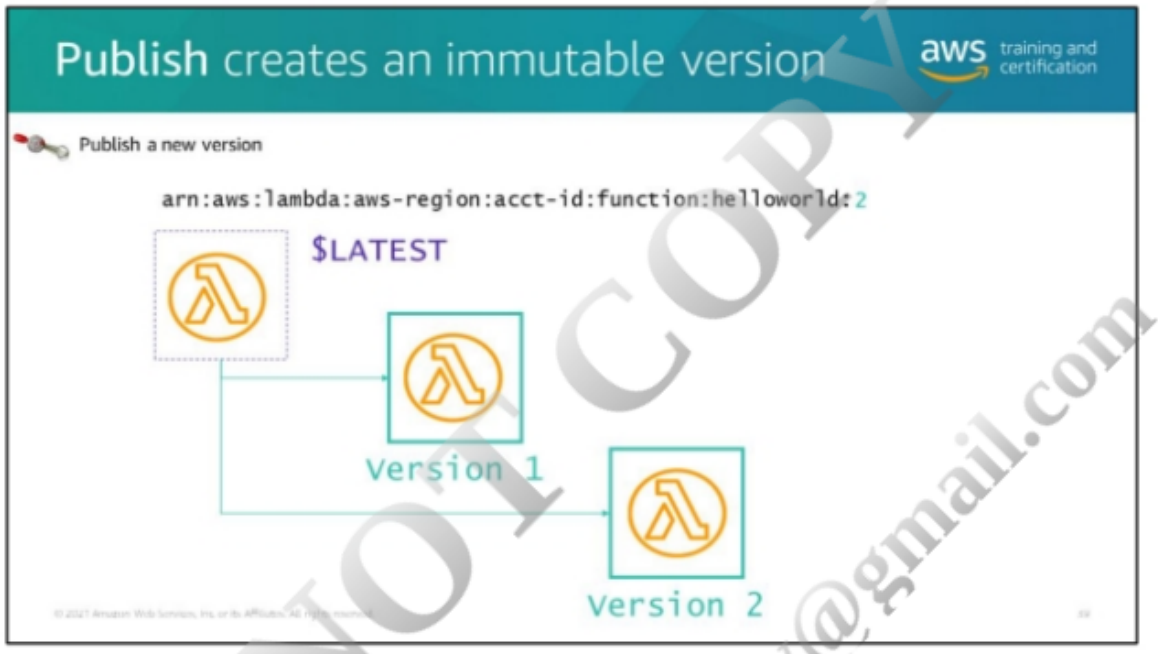


For information about Lambda function versions, visit https://docs.aws.amazon.com/lambda/latest/dg/configuration-versions.html.

All of the pieces just discussed make up a version of your Lambda function:
- Triggers that define when your function will be invoked
- Permissions that define who can invoke the function and what it can interact with
- Configurations that you select to influence how Lambda manages the invocation environments
- Your actual function code

When you create a function, there is only one version: $LATEST

Two Amazon Resource Names (ARNs) are associated with the function:
1. Qualified ARN – The function ARN with a version suffix
2. Unqualified ARN – The function ARN without a version suffix

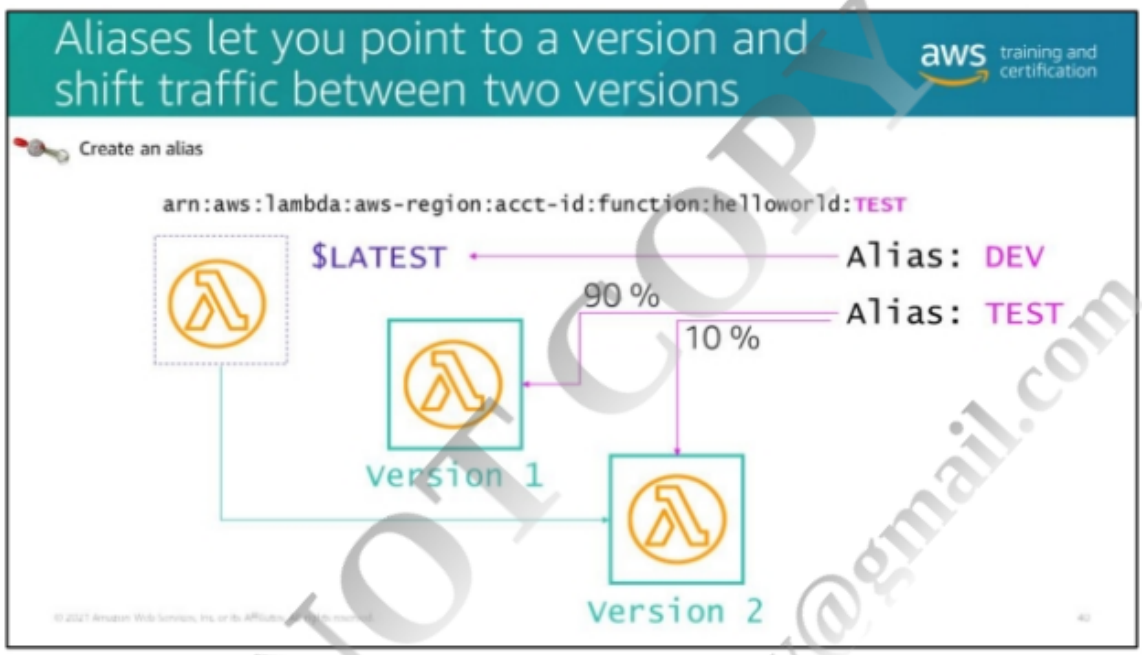


When you save updates to your Lambda function, $LATEST is updated by default. When you choose the option to "publish" a version of the function, Lambda creates a copy of the unpublished $LATEST version and gives it a sequential number. That version is immutable. The code and most of the settings are locked to provide a consistent experience for users of that version.

The qualified ARN for that version puts the version number as a qualifier.

You edit the $LATEST code and publish new versions as needed.

Aliases are a powerful way to manage your function versions.

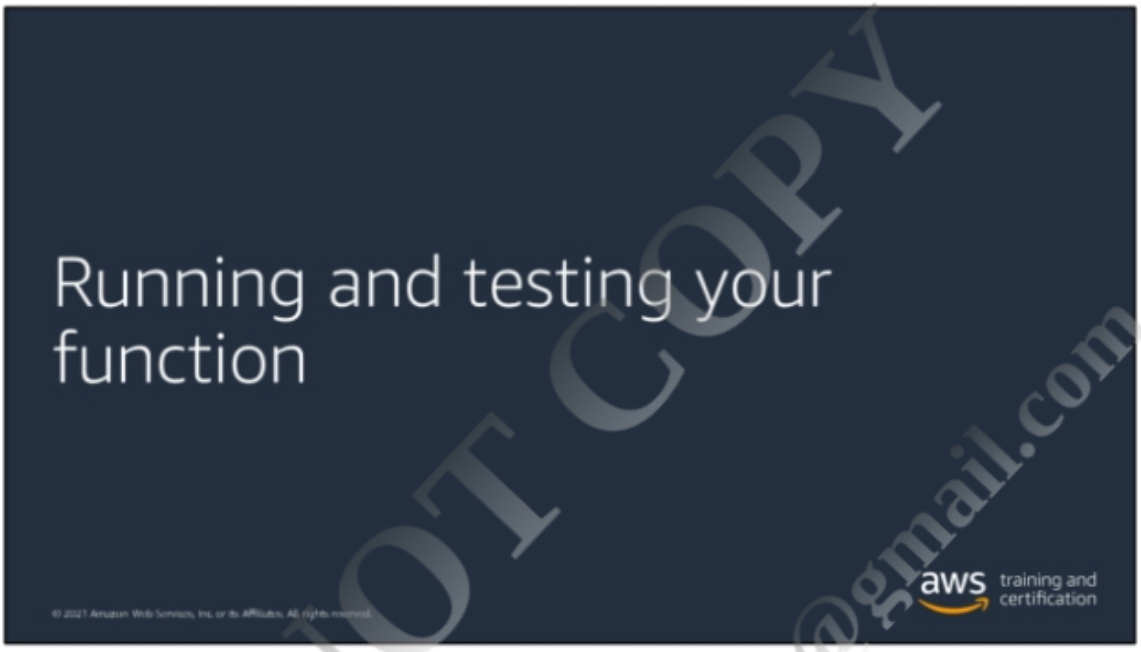


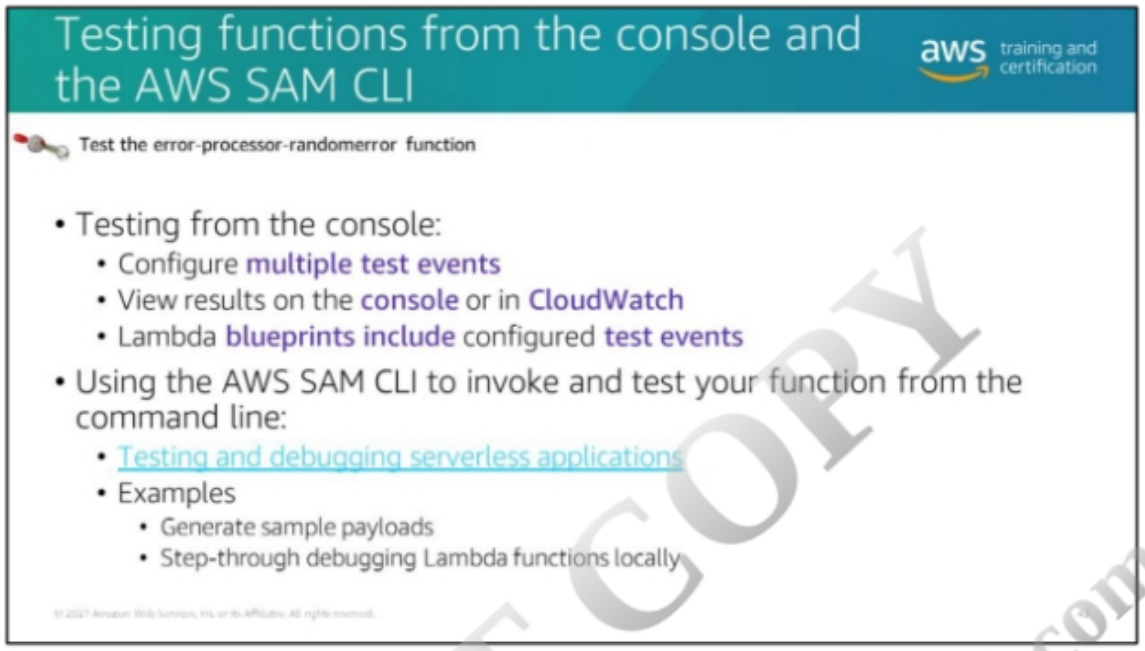
For information about Lambda function aliases, visit https://docs.aws.amazon.com/lambda/latest/dg/configuration-aliases.html.

You can create an alias and associate it to any version of your function, including $LATEST. This allows you to reference your function using the alias as a qualifier in your ARN. So, anywhere you want to reference the function, you can use the alias, and when the function code is updated, you can point the alias at the new version of the code without requiring any coding changes to other resources that reference the function.

A common use of aliases would be to have DEV and TEST aliases. When version 2 has been unit tested and is ready for broader testing, you can point the TEST alias at version 2 instead of version 1.

With Lambda, you can also use alias routing to have an alias point to two versions, sending a percentage of traffic to each version. So, for example, when you first start using version 2 as your test version, you might only send a small portion of traffic to make sure there are no unexpected integration issues. Then, you could update the configuration to send 100% of traffic to version 2 when you are confident it is working as expected.



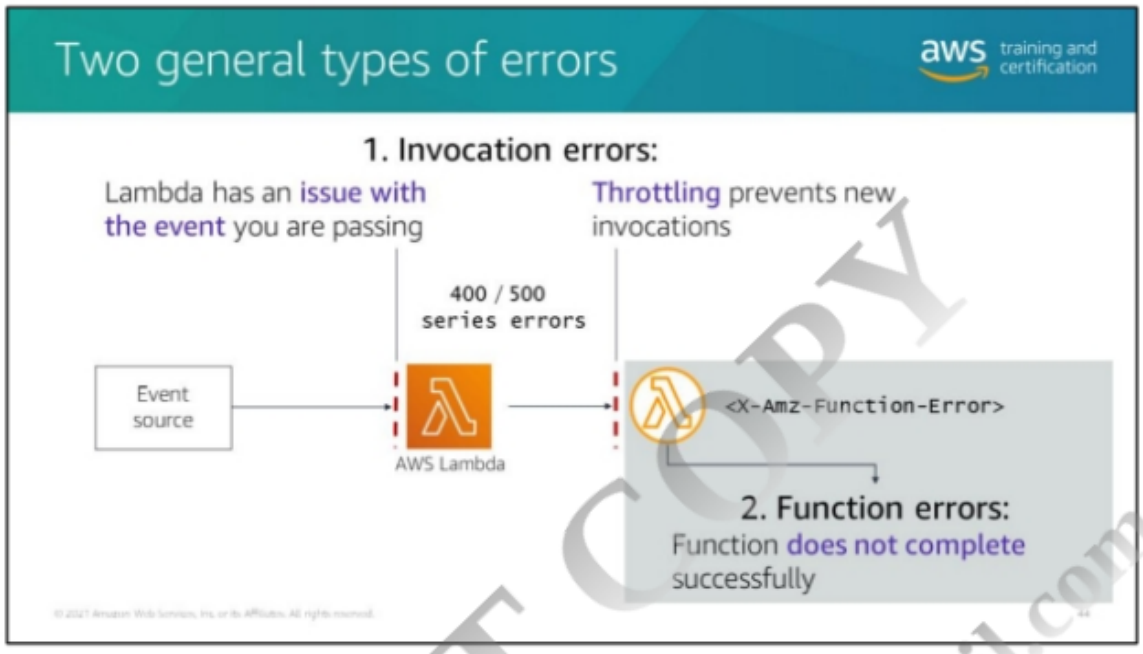


You can create test events and then test any version of your function from the console. Your results display in a panel above the main console sections.

Lambda has built-in metrics and logs that are available directly from the Lambda console and through links to CloudWatch. For information about monitoring functions in the Lambda console, visit https://docs.aws.amazon.com/lambda/latest/dg/monitoring-functions-access-metrics.html. This link is also available in the Online Course Supplement.

You will learn more about Lambda logging and metrics in the observability and monitoring module.

You can also perform local testing on your functions using the AWS SAM CLI. For more information about testing and debugging serverless applications, visit https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/serverless-test-and-debug.html. For example, the Generating Sample Event Payloads page tells you how to create sample payloads that you can use for testing. This link is also available in the Online Course Supplement.



As a developer, be aware of both invocation errors and function errors associated with your function.

An *invocation error* occurs when the Lambda service is unable to invoke your function. Two scenarios generate invocation errors:

- The first scenario occurs when the event passed to Lambda by the event source has an issue. For example, the source doesn't have the proper permissions to invoke the function, the JSON is invalid, or the payload is too large.
- The second scenario occurs when the Lambda service receives the event but cannot initiate an invocation of the function. The most common reason for this type of error is that not enough concurrency is available to start a new invocation environment, so Lambda throttles the request.

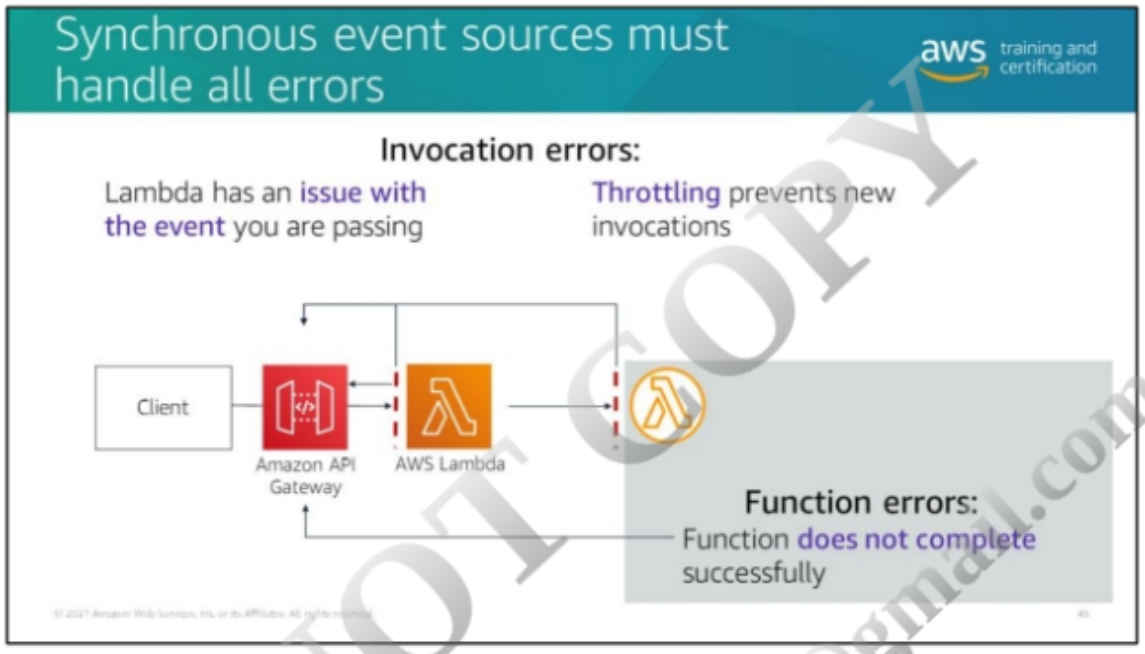
Invocation errors result in a 400- or 500-series error code. For information about the invoke function, visit https://docs.aws.amazon.com/lambda/latest/dg/API_Invoke.html.

In the context of the diner analogy, the first error type might occur when the cook can't start working on your order because they can't read the writing on the order pad. The second invocation error type is the example you heard about earlier—the cook takes the order but can't start on it because no burners are available.

The other type of error is a *function error*. In this case, Lambda invokes the function successfully, but something goes wrong; for example, the function times out or throws an exception. When a function error occurs, Lambda does not generate a 400 or 500 error. Lambda indicates a function error by including a header, named X-Amz-Function-Error, and a JSON-formatted response with the error message and other details.

In our diner analogy, this might be like flipping one of the pancakes onto the floor in the process of cooking them.

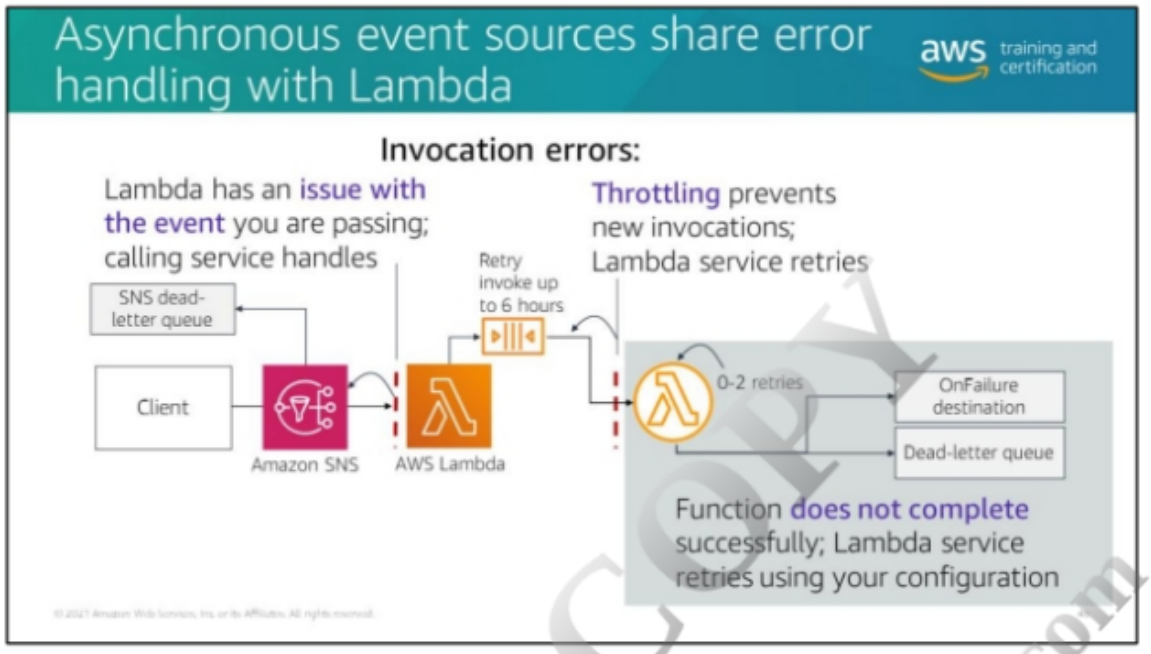
Lambda and the other managed serverless services have built-in error-handling mechanisms that you can use here. As a developer, you need to understand what these mechanisms are, how they differ between Lambda event sources, and what to look for when something goes wrong.



Remember that with synchronous event sources, it's a request/response API call. The event source or the client that it is attached to must handle any error that occurs while invoking or running the function. For example, if API Gateway is being used to proxy a client request to the Lambda function, that client needs to handle the errors that come back.

The AWS CLI and AWS SDK include backoff and retries by default, so take advantage of those to respond to errors.

In our diner analogy, our synchronous transaction is ordering a slice of pie at the counter. The person behind the counter needs to get the pie from the case, return it to the customer, and complete the transaction. If the counter person has any issue (for example, they drop the pie or find out no cherry pie is left), the person has to deal with completing the transaction and messaging back to the customer that they won't get pie.



Remember that with an asynchronous invoke, the event source that requests the invocation does not wait for the results of the function, but the event source is responsible for making sure that the request is handed off successfully.

The diner analogy is the waiter handing an order over to the kitchen manager. The waiter isn't waiting at the counter to verify that the order is completed successfully, but the waiter does need to make sure they've delivered the order in a way and at a location so that the cook will receive it. If the kitchen manager can't read the order or a customer requests an item that isn't on the menu, the waiter must deal with that error. If the kitchen manager is out on a break, the waiter must deal with that.

With asynchronous event sources, the client or invoking service is still responsible for dealing with most errors that prevent Lambda from invoking the function; for example, permissions issues or invalid JSON. If you are writing your own asynchronous event sources, you need to manage those types of errors and design retry mechanisms.

When you use an AWS service as an asynchronous event source, these types of upfront invocation errors are retried or discarded based on the retry behaviors associated with the **service invoking the function**. For example, when Amazon SNS is the event source, if it can't hand off the request to Lambda, it applies its own retry policy. For more information about message delivery retries, visit https://docs.aws.amazon.com/sns/latest/dg/sns-message-delivery-retries.html.

Amazon SNS won't retry deliveries that occur due to client-side errors. An example of this would be trying to deliver the message to a function that's been deleted. (There's no cherry pie, but you tried to order it. There's no point in trying to order it again.)

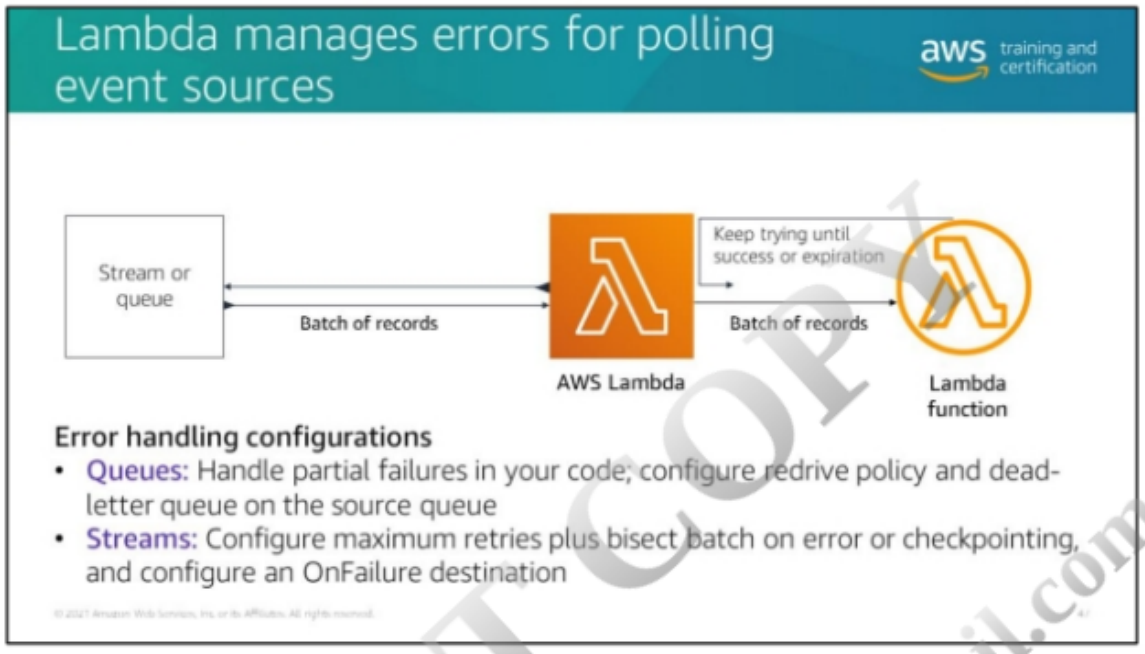
Server-side errors can happen when the system responsible for the subscribed endpoint becomes unavailable or returns an exception that indicates it can't process a valid request from Amazon SNS. (The order is valid, but the cook is on a break.) Amazon SNS retries server-side errors based on the retry policy for the type of endpoint. When Lambda or another AWS managed endpoint is the subscriber, retries extend for a very long period, and the time increases between attempts.

When Amazon SNS gets a client-side error or a server-side error exhausts its retries, the message is discarded unless you configure a dead-letter queue associated with the SNS topic. For more information about Amazon SNS dead-letter queues, visit https://docs.aws.amazon.com/sns/latest/dg/sns-dead-letter-queues.html.

If the asynchronous event source successfully hands the request off to Lambda, the event source receives a 202 message indicating that it was successful. At this point, Lambda is managing the events from its own internal queue. Invocation errors that happen after that (for example, throttling errors) are Lambda's responsibility, and it will retry to invoke them. (A kitchen manager still handles valid orders if the kitchen is backed up.)

Lambda controls the frequency and backoff pattern of retries for this type of failure, but you can configure how long failed invocations are retried in the function configuration (up to 6 hours). If Lambda successfully invokes the function but your function throws an exception or doesn't complete, Lambda retries running the function up to two more times. You can set this retry value from 0-2 in the function configuration.

You can send Lambda invocation failures that continue to fail beyond the duration you set, as well as function errors that exhaust your retry configuration, to an OnFailure destination or a dead-letter queue. Your application should include methods for handling these records, perhaps replaying them after some condition is resolved or sending them to a different process for resolution. If EventBridge is your event source, you can configure the event replay feature to archive events to be replayed later. This can be useful for recovering from processing errors.

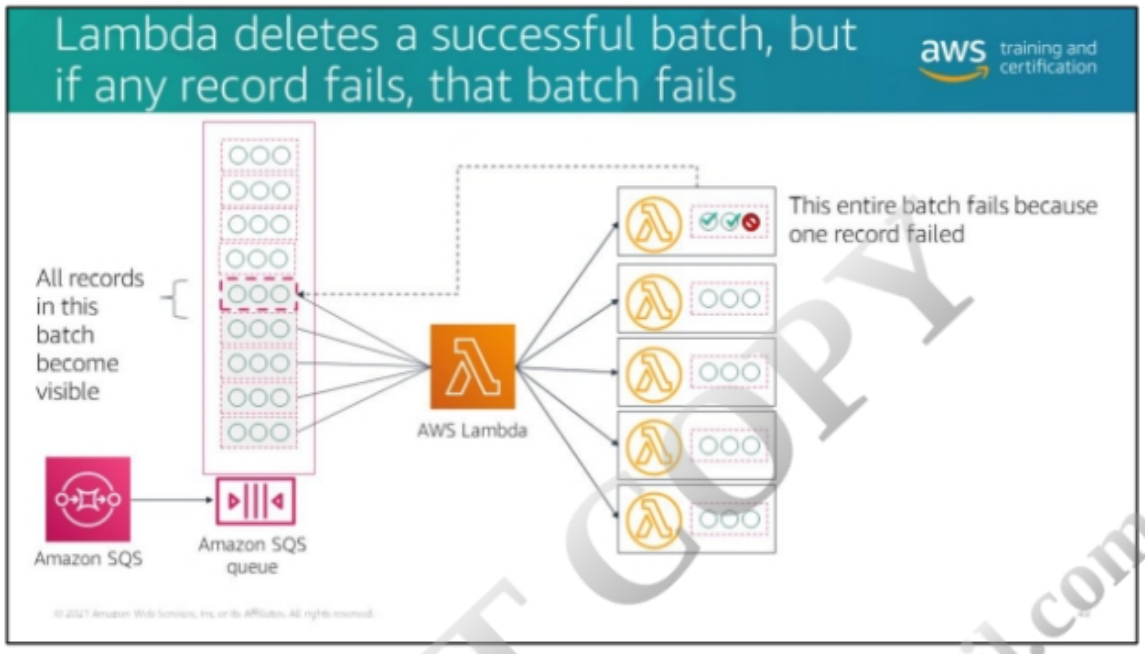


With both streams and queues, records are produced upstream and added to the stream or queue. Lambda gets a batch of records off the queue or stream and then invokes the function synchronously with that batch.

Generally speaking, Lambda owns the error handling between itself and the event source, and Lambda will keep trying a batch until it succeeds (by default) or the records expire. For these event sources, be particularly aware of queue and stream behaviors that indicate Lambda may be running into a high number of errors.

With both queues and streams, you have some configuration options to modify the default error-handling behaviors to get failed records out of the way, and it is a best practice to use these.

Additionally, because Lambda is processing records in batches, some of the records in the batch might succeed while others fail. This approach also increases the chance that a record may be processed more than once. As a developer, this means you need to account for partial failures and make sure that you are following the best practice of accounting for duplicate records.



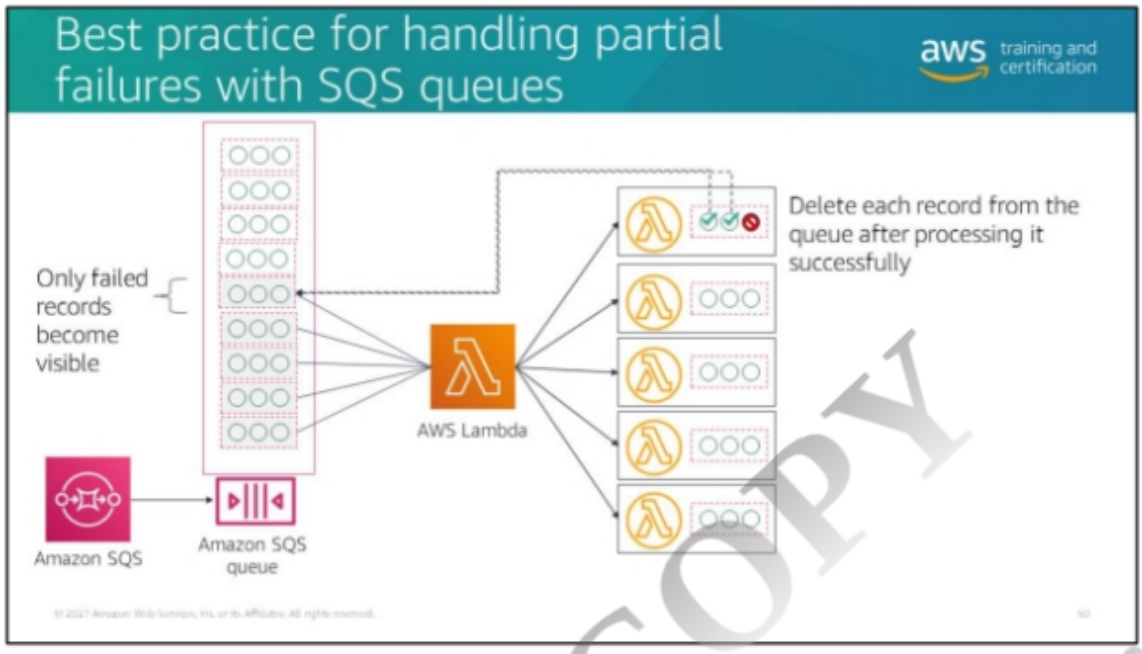
As discussed in the module on event-driven development using polling event sources, when an SQS queue is used as the event source, Lambda polls the queue in batches and synchronously initiates one invocation per batch of records. Lambda starts with five polling processes, which will invoke five concurrent instances of your function.

If any record in the batch fails, the Lambda service identifies this as a failure, and the entire batch fails. This includes the function timing out before all of the records in the batch were processed. When the batch fails, those records become visible again on the queue.

If this happens routinely, you might have chosen a Lambda function timeout that is too short for the average time that it takes to process your batch. For example, if your function timeout is set for 30 seconds and you have batches of three records that each take on average 10–15 seconds to process, your function would frequently time out before all of the records have been processed.

If you do not configure any additional error handling options, those records will continue to be retried until they exceed the message retention period set on the queue.

This is one of the reasons idempotency is important. Records may be processed repeatedly as part of a batch that doesn't complete successfully.

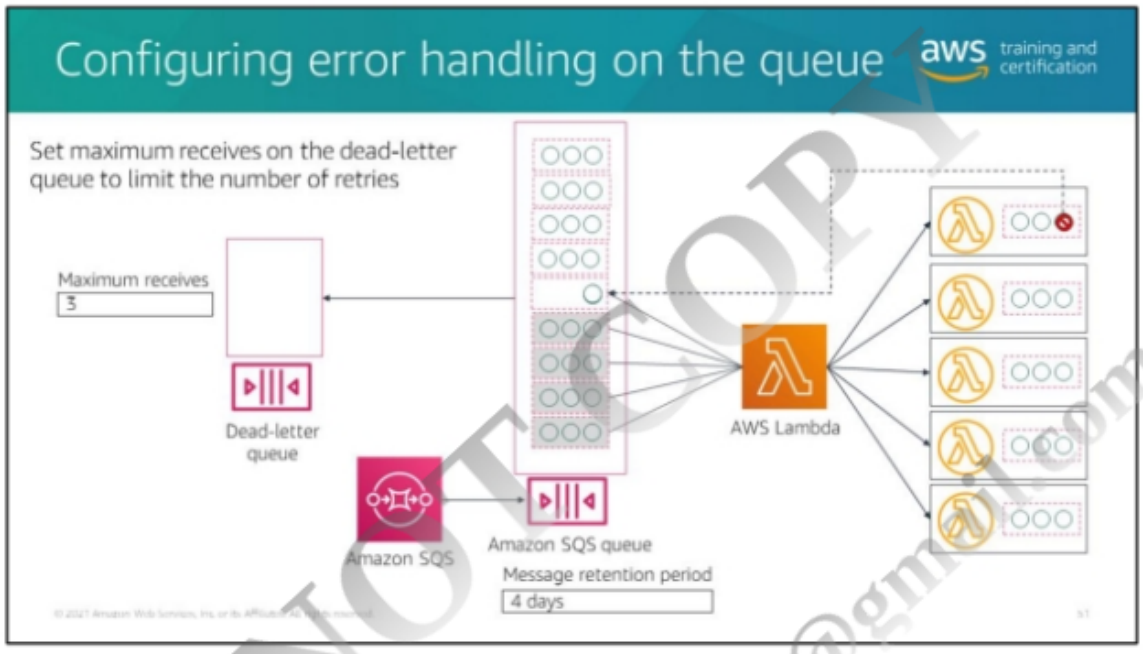


Rather than letting Lambda continually process the same records because of the partial failure of a batch, it's a best practice to have your Lambda function code delete successfully processed records off of the queue as soon as processing is complete. This way, if part of the batch fails, only the failed record becomes visible again on the queue.

New!

AWS Lambda now supports partial batch response for SQS as an event source. With this feature, when messages on an SQS queue fail to process, Lambda marks a batch of records in a message queue as partially successful and allows reprocessing of only the failed records. By processing information at a record-level instead of batch-level, AWS Lambda has removed the need of repetitive data transfer, increasing throughput and making Amazon SQS message queue processing more efficient.

https://docs.aws.amazon.com/lambda/latest/dg/with-sqs.html#services-sqs-batchfailurereporting

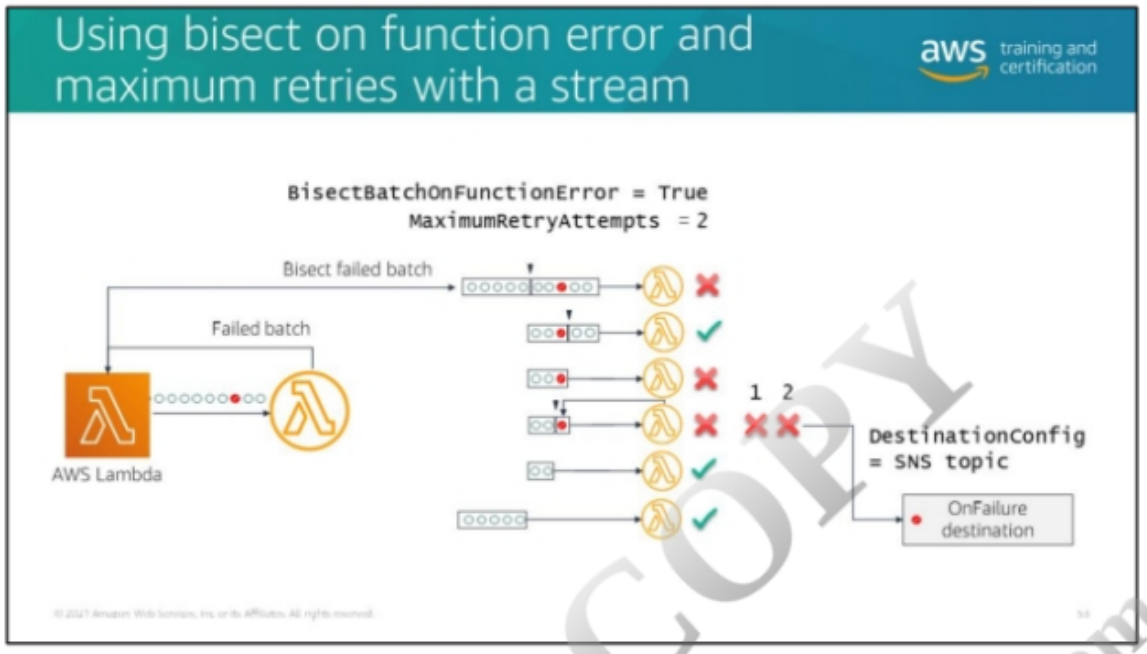


Although Lambda manages the SQS queue, you have some options for configuring how long-erroring records are retried.

You can limit the retention period on the queue itself. Messages are discarded if they remain on the queue beyond the time limit that you set.

It is a best practice to configure a dead-letter queue with a "maximum receives" value, which limits how many times Lambda will retry a failing record. When a record hits the maximum receives value, it is sent to the dead-letter queue where you can perform offline analysis or another automated process on failed records.

Note: When Amazon SQS is the event source, the dead-letter queue that you set is on the source queue, not the Lambda function. The dead-letter queue on the function is used for asynchronous invocations of the function.



As a best practice, use bisect on error with maximum retries or maximum age to prevent your stream processing from getting blocked indefinitely.

Bisect batch on function error tells Lambda to split a failing batch into two and retry each batch separately.

Maximum retry attempts and **maximum record age** let you limit the number or duration of retries on a failed batch.

An **OnFailure destination** lets you send failed records to an SNS topic or SQS queue to be handled offline without having to add additional logic into your function.

The slide illustrates how this works with a batch of 10 records.

In this example, **BisectOnFunctionError** = True, **MaximumRetryAttempts** = 2, and **DestinationConfig** includes an **OnFailure destination** that points to an SNS topic. In this example, the third record (in this batch of 10) returns a function error.

When the function returns an error, Lambda splits the batch into two and then sends them to your function separately, still maintaining record order. Lambda also resets the retry and max age values whenever it splits a batch.

Now you have two batches of five. Lambda sends the first batch of five, and it fails, so the splitting process repeats. Lambda splits that failing batch, yielding a batch of two and a batch of three records. Lambda resets the retry and max age values and sends the first of those two batches for processing.

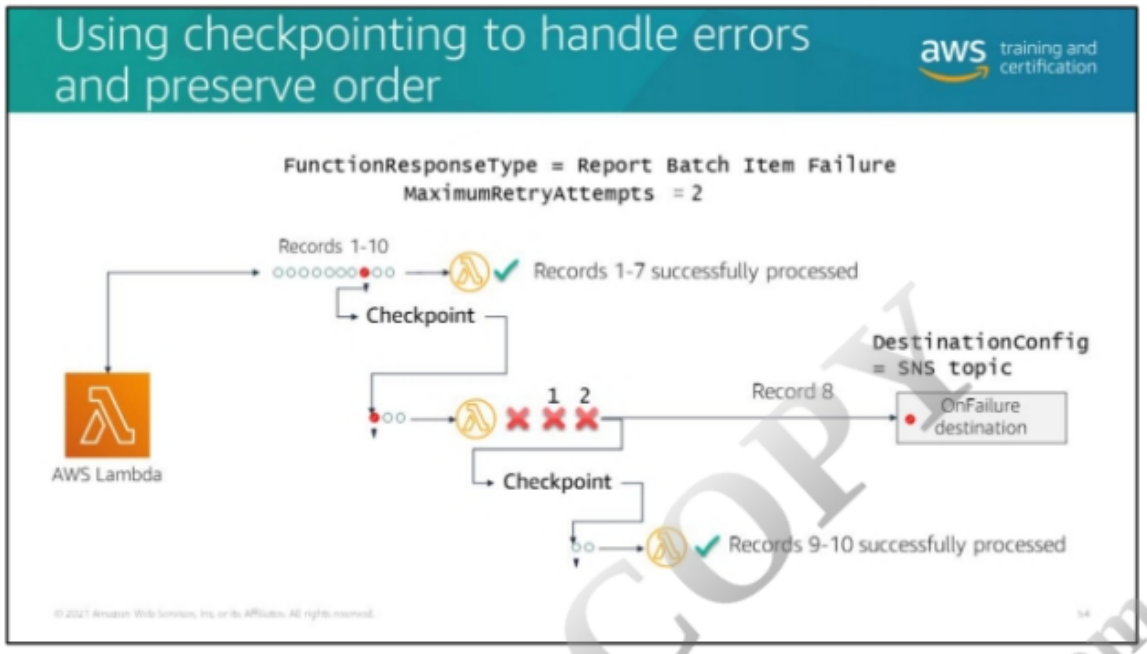
This time, the batch of two records processes successfully, so Lambda sends the batch of three to the function. That batch fails, Lambda splits it, and now it has a batch with one record (the bad one) and another with two records.

Lambda sends the batch with the bad record and it fails, but there's nothing left to split. So, now the max retry and max age settings come into play. In this example, the function retries the record twice, and when it continues to fail, the function sends the record to the SNS topic configured for the OnFailure destination.

With the erroring record out of the way, Lambda works its way back through each of the smaller batches that it created, always maintaining record order. So, Lambda is going to process the unprocessed batch of two and then the unprocessed batch of five.

At that point, the original batch of 10 is marked as successful, and Lambda moves the pointer on the stream to the start of the next batch of records.

One drawback of this approach is that records can be processed more than once.



As illustrated in the prior slide, when you use BisectBatchOnError, you might incur duplicate processing as batches are split and retried.

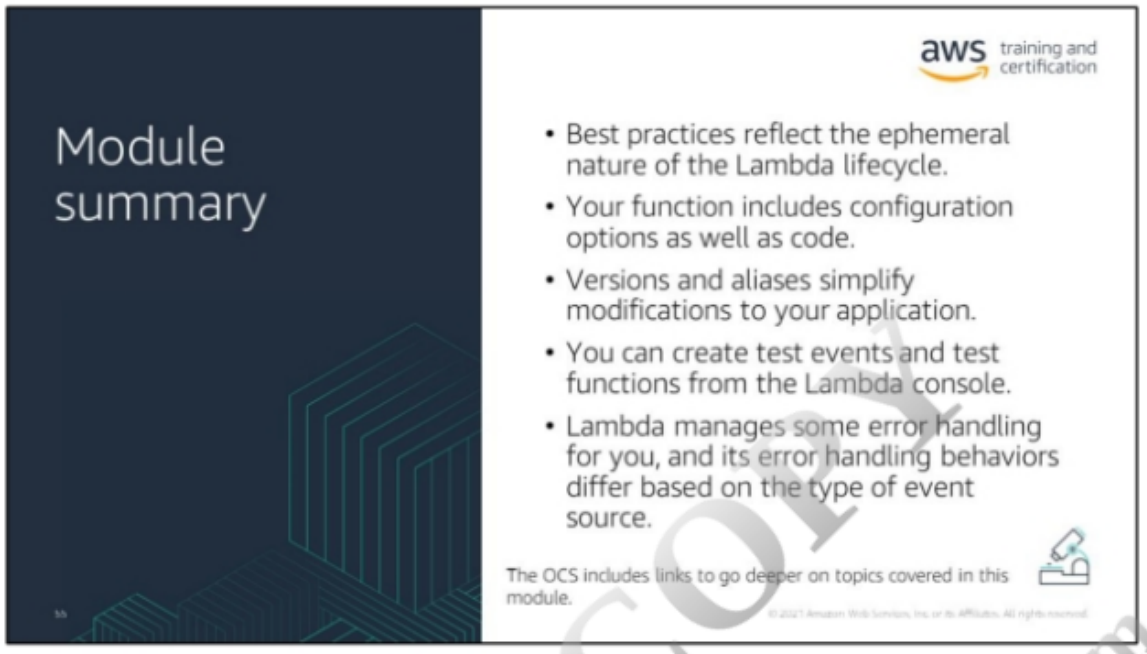
When you set **FunctionResponseType** equal to **Report Batch Item Failure**, if a batch fails to process, only records after the last successful message are retried. This reduces duplicate processing and gives you more options for failure handling. If a failure occurs, Lambda prioritizes checkpointing, if enabled, over other mechanisms to minimize duplicate processing.

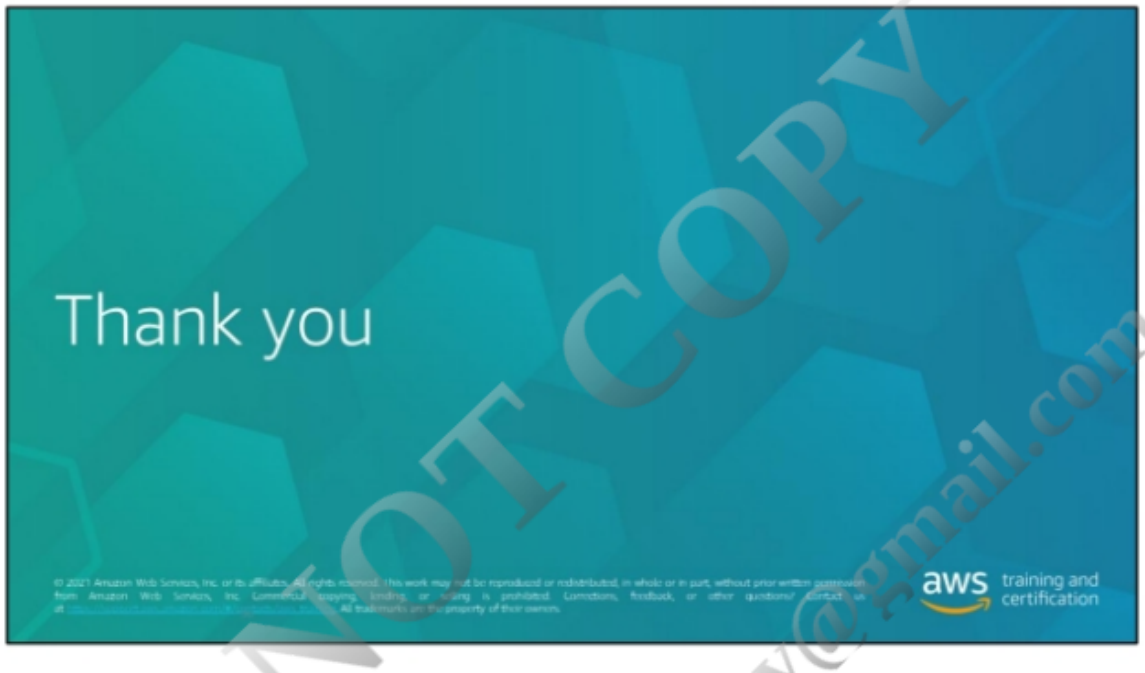
In this example, the first seven records in a batch of ten process successfully, but the eighth record fails. With checkpointing enabled, rather than retrying the entire batch of 10 or splitting the batch into two, Lambda begins retries from the checkpoint. So in this example, Lambda would start retrying with record eight. To use checkpointing, your Lambda function must return a failed sequence identifier. For more information, visit https://aws.amazon.com/blogs/compute/optimizing-batch-processing-with-custom-checkpoints-in-aws-lambda/.

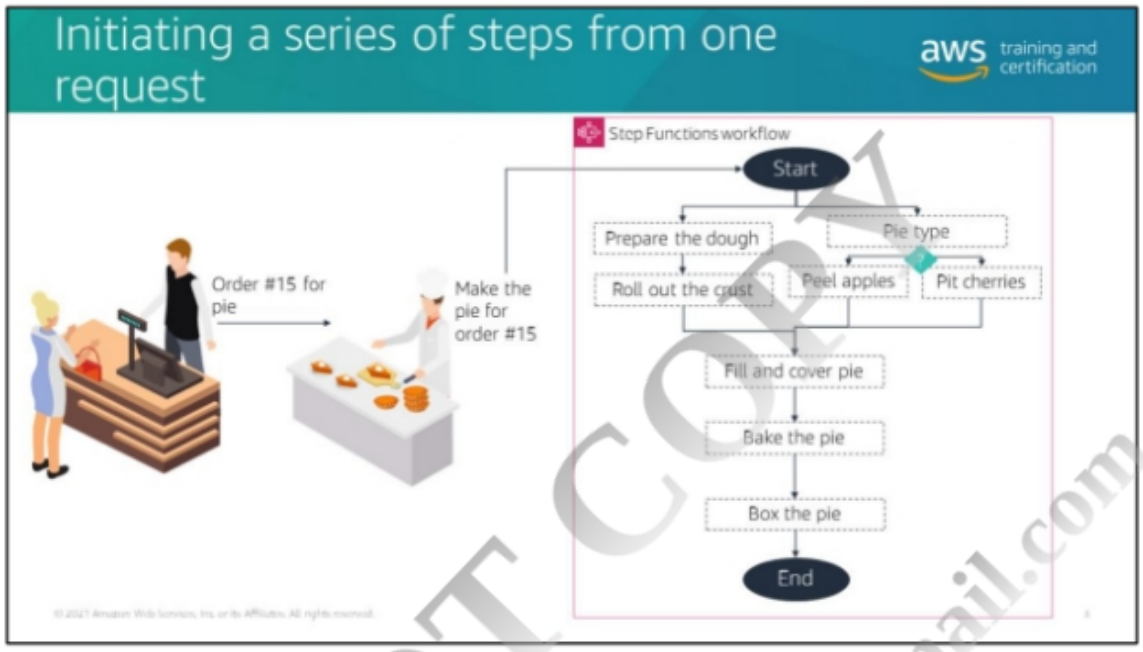
MaximumRetryAttempts still provides the option to limit retries, and you can send records that continue to fail to an OnFailure destination. In this example, record eight continues to fail, so the checkpoint remains at the same point until, after two retries, the record is sent to the OnFailure destination.

The checkpoint now finds record nine to be the next unprocessed record in the batch and starts processing from there. Records nine and ten process successfully, and processing order for records 1-7 and 9-10 is preserved.

For more information about checkpointing for Kinesis and DynamoDB Streams, visit https://aws.amazon.com/about-aws/whats-new/2020/12/aws-lambda-launches-checkpointing-for-amazon-kinesis-and-amazon-dynamodb-streams/.







Let's get back to our baker. When an order to make a pie comes in, that's actually a series of related but distinct steps. Some steps must be completed first or in sequence, and some can be worked in parallel. Some steps take longer than others. Someone with expertise in each step performs that step.

You could have each team member hand off to the next, and make each team member responsible for knowing what comes before and after and what to do if there's a problem. This means that each staff member needs some knowledge of and spends some effort outside of their expertise. On a small scale, you would be OK with this approach, but you plan to make a lot of pies.

The other potential issue is that you're paying for your employees' time, so you don't want one of them to stand idle waiting on a long task to complete. For example, you don't want the person responsible for baking the pie to stand at the oven looking through the window until the pie is done. You want the person to come back and take the pie out when it's finished.

To make things go smoothly and let the experts work in their area of expertise, you could add a way to manage the flow of steps and keep others informed of the status.

Step Functions keeps your Lambda functions focused on business

aws training and certification

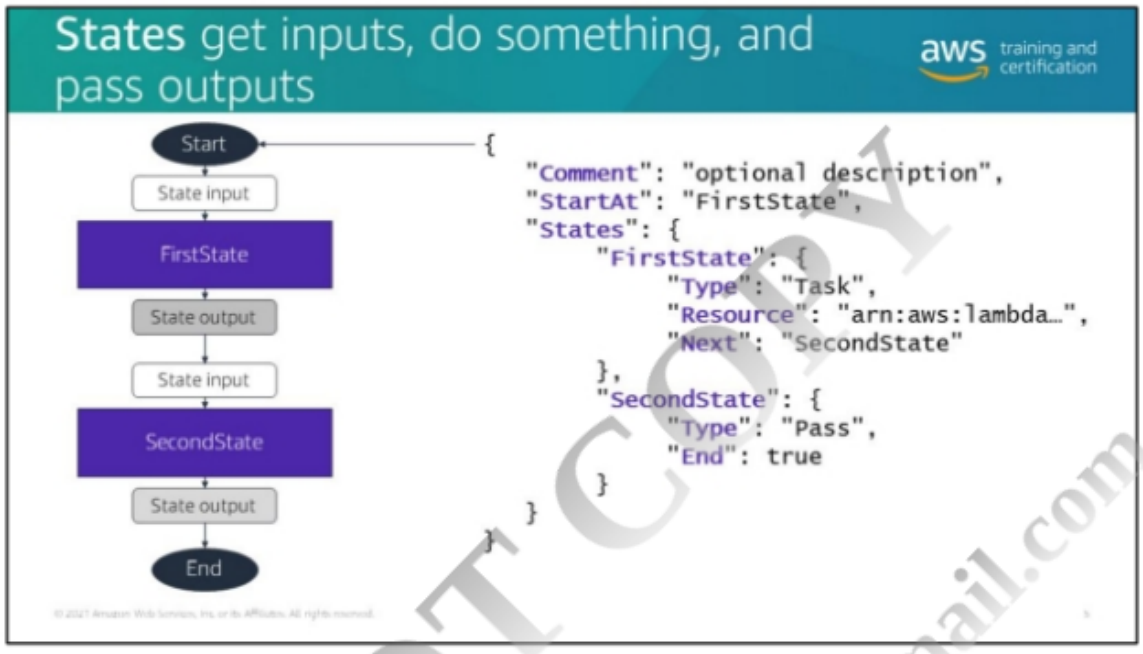
Developer features

- Define **workflows** as **state machines**
- Use the States language to handle orchestration logic and **maintain state** outside of your code
- Start from templates and code snippets available on the console
- Monitor progress via an **autogenerated flow diagram** and **run history**

References

- States language: https://states-language.net/spec.html
- Templates: https://docs.aws.amazon.com/step-functions/latest/dg/concepts-templates.html
- Code snippets: https://docs.aws.amazon.com/step-functions/latest/dg/concepts-code-snippets.html

The AWS Step Functions console also provides access to templates and snippets that you can start from.



States are the elements that perform functions in your state machines. The state name is a string that you use to identify it.

A Step Functions execution receives JSON text as input and passes that input to the first **state** in the workflow.

The state machine always has a **StartAt** field, which indicates by name the state where processing starts. The state machine also always has a **States** section, which defines all of the states within the state machine.

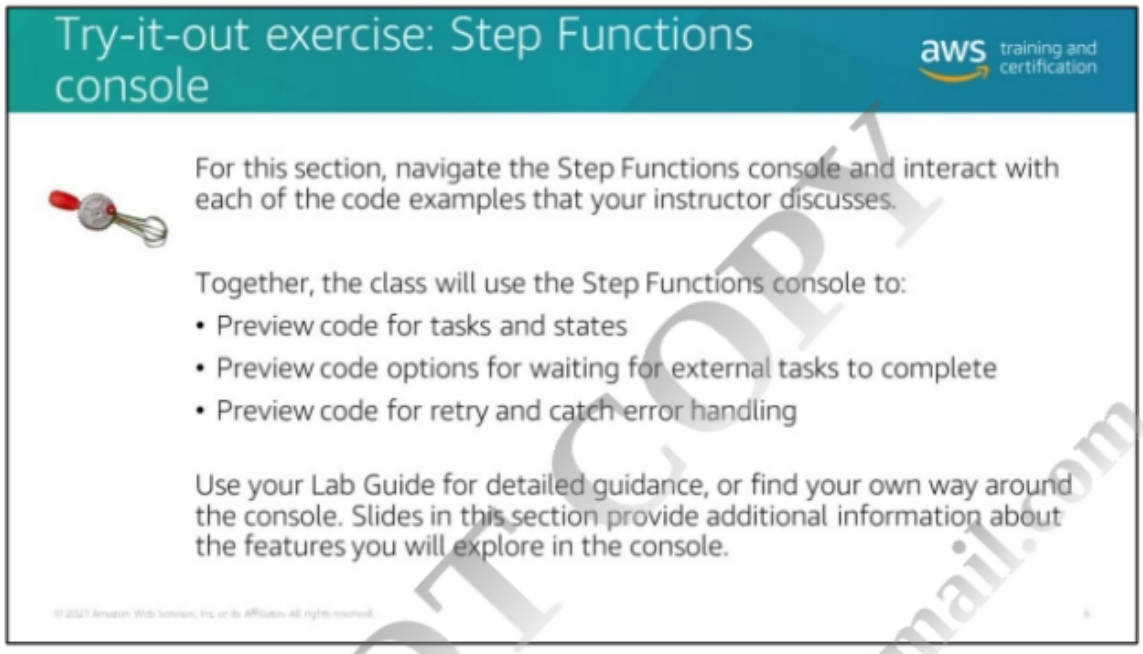
Each state must have a **Type** field indicating what type of state it is. Each state type is designed for a different type of logic, and some fields are relevant to specific state types.

Most states have a **next** or **end** field, which tells the state machine what to do upon completion of that state. States are not processed in the order listed; the next field drives the order of operations.

In this example, FirstState is a Task state that runs an AWS Lambda function and indicates that the next step is SecondState. SecondState is a Pass state type and indicates that processing ends after it completes.

For more information about state machine structure, visit https://docs.aws.amazon.com/step-functions/latest/dg/amazon-states-language-state-machine-structure.html.

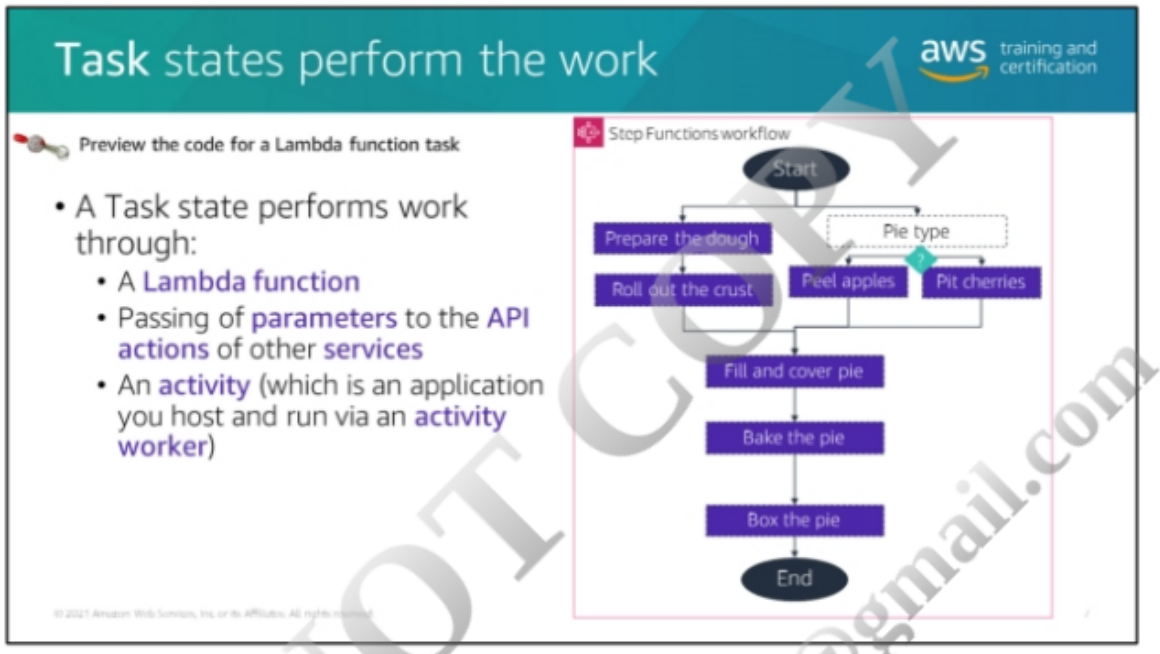
For more information about common state fields, see https://docs.aws.amazon.com/step-functions/latest/dg/amazon-states-language-common-fields.html.



In this section, you will use the Step Functions console to review state types.

Start by selecting **State Machines** in the Step Functions console navigation menu. Choose **Create state machine**. Select **Author with code snippets**. Select **Standard** for the type.

The remaining steps are listed in the OCS. Slides marked with the egg beater icon used on this slide are associated with steps in the exercise.



In the pie example, most of the listed items are Task states as described; work is happening in each step. These steps could each represent individual Lambda functions but could also represent interaction with another service or application code that you have written and hosted somewhere else.

Supported service integrations:
- Optimized integrations (17 services)
- AWS SDK integrations (over 200 services)

https://docs.aws.amazon.com/step-functions/latest/dg/connect-supported-services.html

