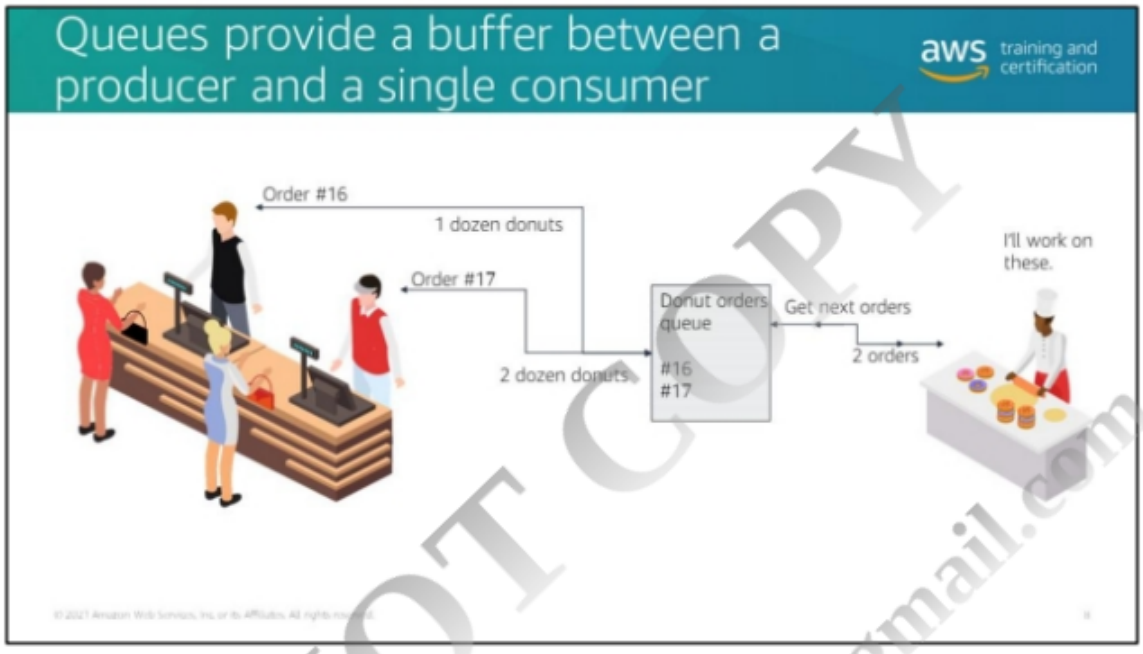
In stream processing:

- A stream of messages is the core entity. Only by looking at many messages together do you get the value of each individual message.
- The stream of messages is also constant in most cases.
- Clickstream data from mobile apps, application logs, and home security camera feeds are common examples where streams provide the responsiveness you need to make data actionable.

For a full list of AWS services that Lambda invokes via event source mappings, see the section titled **"Services that Lambda reads events from"** at https://docs.aws.amazon.com/lambda/latest/dg/lambda-services.html.

This link is also available in the OCS.

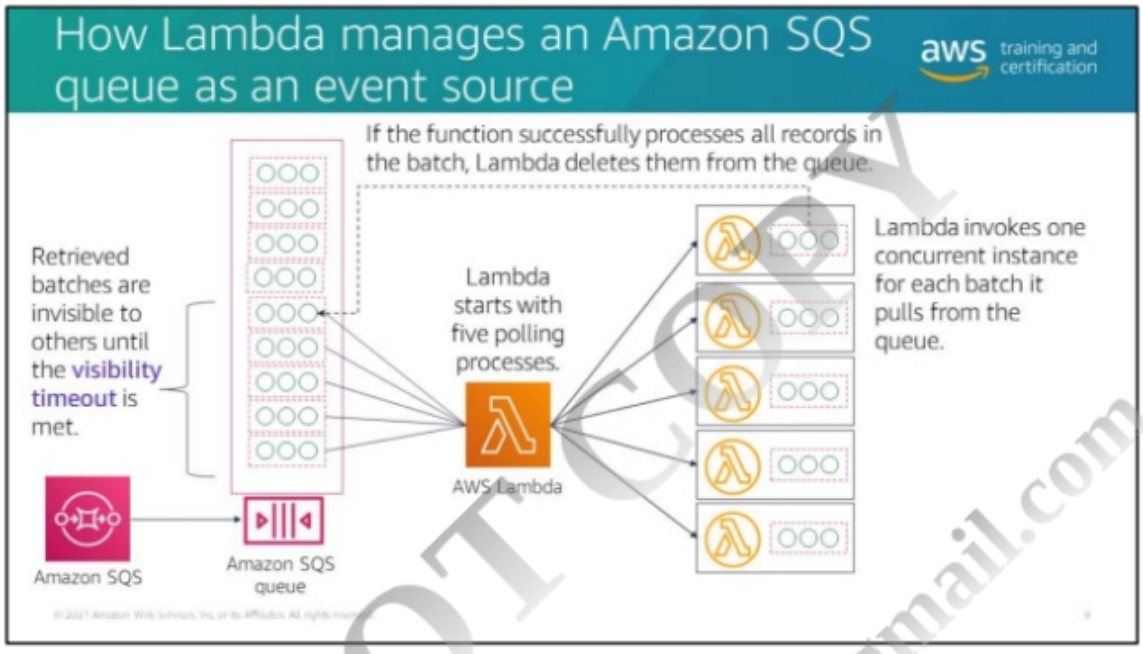
In this course, you hear about Amazon SQS and Amazon Kinesis Data Streams and Amazon DynamoDB streams as examples of services that Lambda reads events from.



In the bakery example, this is like an order queue that the donut maker works from. The donut makers don't have to work as quickly as orders are arriving. The queue provides a buffer that lets them do the work at the pace they need.

Notice that from the customer standpoint, the response is still asynchronous.

The difference is on the backend where there is now a temporary store for events from which things get processed in batches versus as each item arrives.



When Amazon SQS is the event source, Lambda takes care of behaviors that would be your responsibility if you were setting up an SQS queue with another service. Lambda does the following:

- Polls the queue
- Invokes your function synchronously with an event that contains a batch of messages
- Deletes a successfully processed batch of records off the queue
- Scales pollers and Lambda concurrency up and down based on queue depth and error rates

Lambda starts with a default number of pollers and gets one batch for each process. It will use the batch size you set on your function for the queue.

Lambda synchronously initiates one invocation per each batch of records it gets from the queue while polling. In this example, there are five pollers, and five concurrent instances of Lambda.

When Lambda pulls batches of records, those records become invisible to other consumers of the queue (that is, other Lambda invocations) for the duration of the visibility timeout. The visibility timeout is a setting that you control. You also control the batch size that Lambda will use to pull batches off of the queue.

Lambda will increase the number of polling processes if it finds the queue of messages is getting larger, and it will also slow down the number of processers if it is getting a lot of errors back from its invocations, assuming this means there is too much pressure on a downstream consumer.

You hear more about this behavior in the Handling Scale in Serverless Applications module.

If Lambda successfully processes the batch, it deletes those messages off the queue on your behalf so that you don't have to write any code to delete those messages from the queue.

You learn more about retries, error handling, and scaling with Amazon SQS queue event sources later in the course.

What you handle with Amazon SQS as an event source for Lambda

aws training and certification

What you handle	Best practices
On the queue	
Select queue type (standard or FIFO)	Choose a standard queue unless the order of processing is important.
Set visibility timeout on the queue	Set the visibility timeout to at least six times the function timeout.
Set redrive policy on the queue	Configure the dead-letter queue and choose a maximum receive count to handle spikes in traffic, but limit the frequency of bottlenecks.
On the function	
Function batch size	Set the batch size higher for fast workloads and lower for long workloads.
Function timeout	Set the function timeout to complete successfully under most circumstances.
Partial failures	Include code to delete each record as it is processed.
Duplicate messages	Design for idempotency. Include function code that prevents you from processing the same record twice.

 10

When Lambda is the consumer of a queue, it handles some aspects of the queue for you.

For more information about Amazon SQS in general, see https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/welcome.html.

For more information about using AWS Lambda with Amazon SQS, see https://docs.aws.amazon.com/lambda/latest/dg/with-sqs.html.

Let's look briefly at how Lambda works with the queue and how your choices come into play. You'll learn more about failure handling in the Writing Good Lambda Functions module.

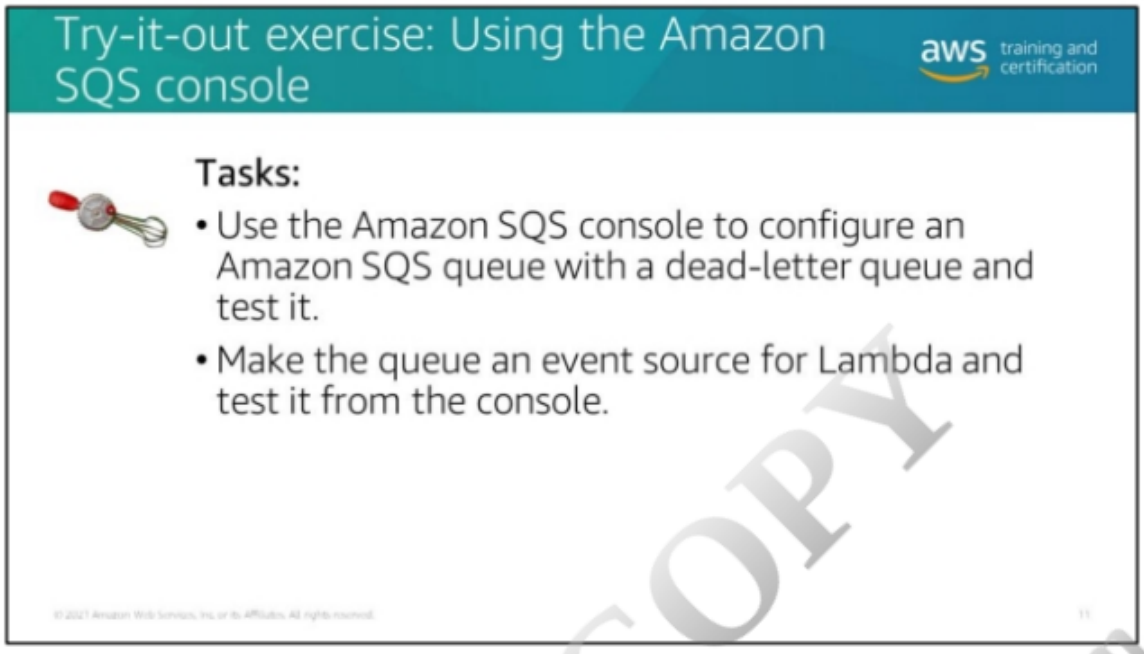
With standard queues, order is not guaranteed, and messages may be delivered more than once. But you can process nearly unlimited messages per second. With FIFO queues, order is strictly preserved per group ID, but throughput is more limited.

It might not be intuitive that you would need such a high visibility timeout, but this is a good example of why you have to test, monitor, and tweak your configurations based on production behaviors to find the right balance.

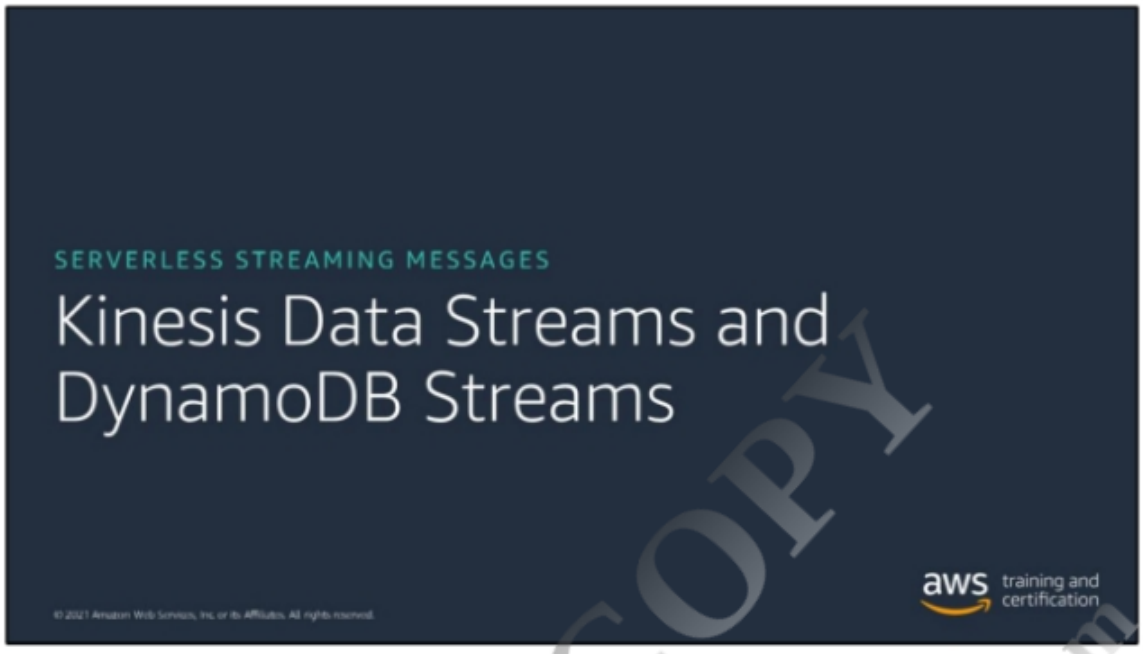
You need to configure the visibility timeout to allow enough time for your Lambda function to complete a message batch. For example, you have a batch size of 10 and a Lambda function that takes 20 seconds to process the batch. You need a visibility timeout that is greater than 20 seconds.

You also need to leave some buffer in the visibility timeout to account for Lambda invocation retries when the function is getting throttled. You don't want your visibility timeout to expire before those messages can be processed.

The best practice is to set your visibility timeout to at least six times the timeout you configure for your function to account for this.



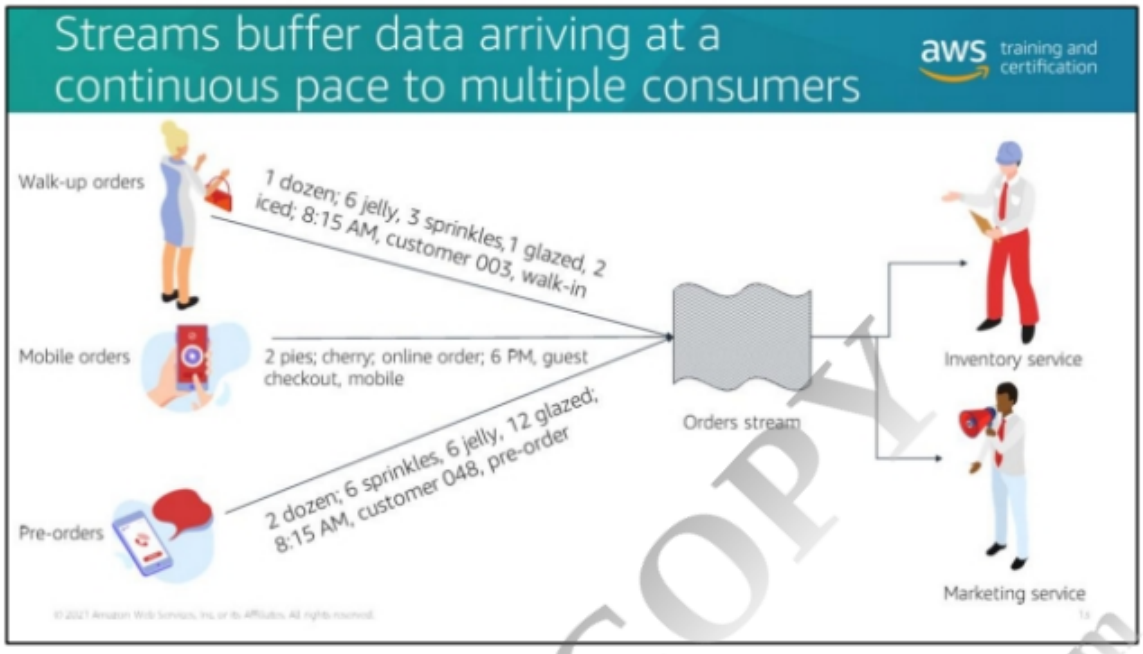
In this quick task, you will use the Amazon SQS console to configure a queue and set up and do some simple testing of a dead-letter queue.



DynamoDB streams provide an ordered set of records that other services, including Lambda, can consume.

Kinesis Data Streams is part of the Amazon Kinesis family of streaming services that allows you to push producer record sources onto the stream and then consume data from the stream with Lambda or other consumers.

DynamoDB Streams provides a stream of item changes that you can read from to get a change log of activities on your database.



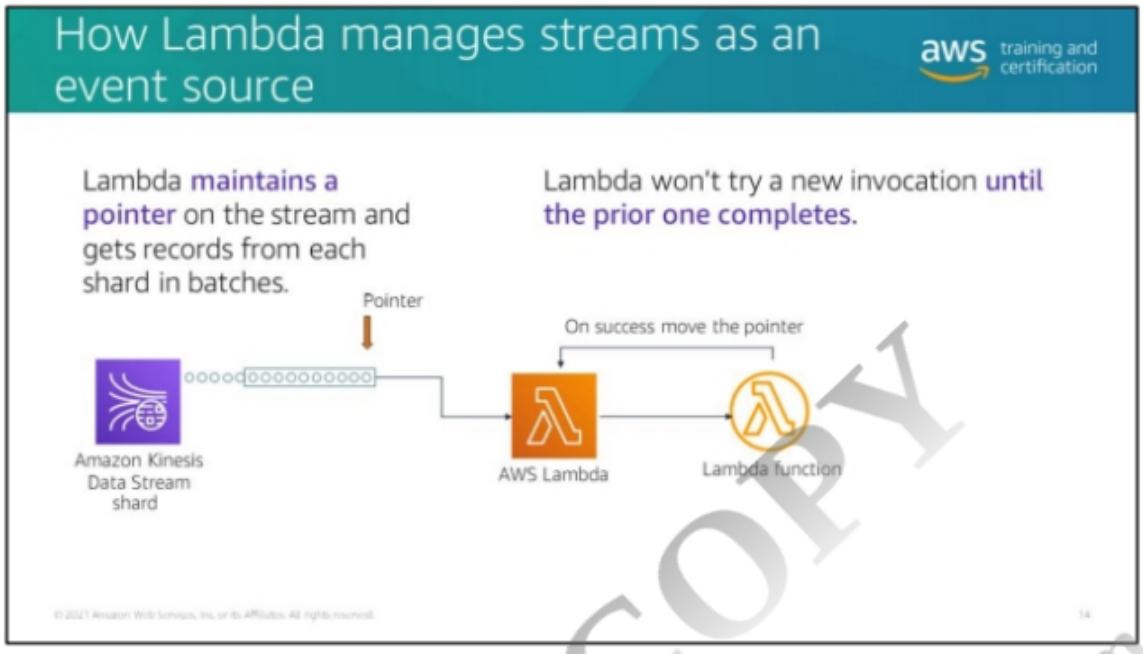
Streams can be fed from multiple sources and can have multiple consumers.

In the bakery example, this could be a stream that collects data about all the orders. The order service or several services are the producers adding data to the stream.

The inventory service might be a consumer of the stream who is using it to dynamically offer discounts on items that are not selling as well that day.

The marketing service might also consume this data to make personalized recommendations to users who are shopping online.

Streams let you process data much closer to real time than if you were to process the data in batches at the end of the day.



Lambda polls the stream and invokes your function with an event that contains a batch of messages. Lambda maintains a pointer of the last record processed on the stream. The Lambda service gets one batch of records per shard and invokes one Lambda function per shard. If that batch is successfully processed, Lambda will move on to the next batch of records in the shard based on the pointer position.

Lambda will not increase the concurrent invocations unless you increase the number of shards or use concurrent batches per shard.

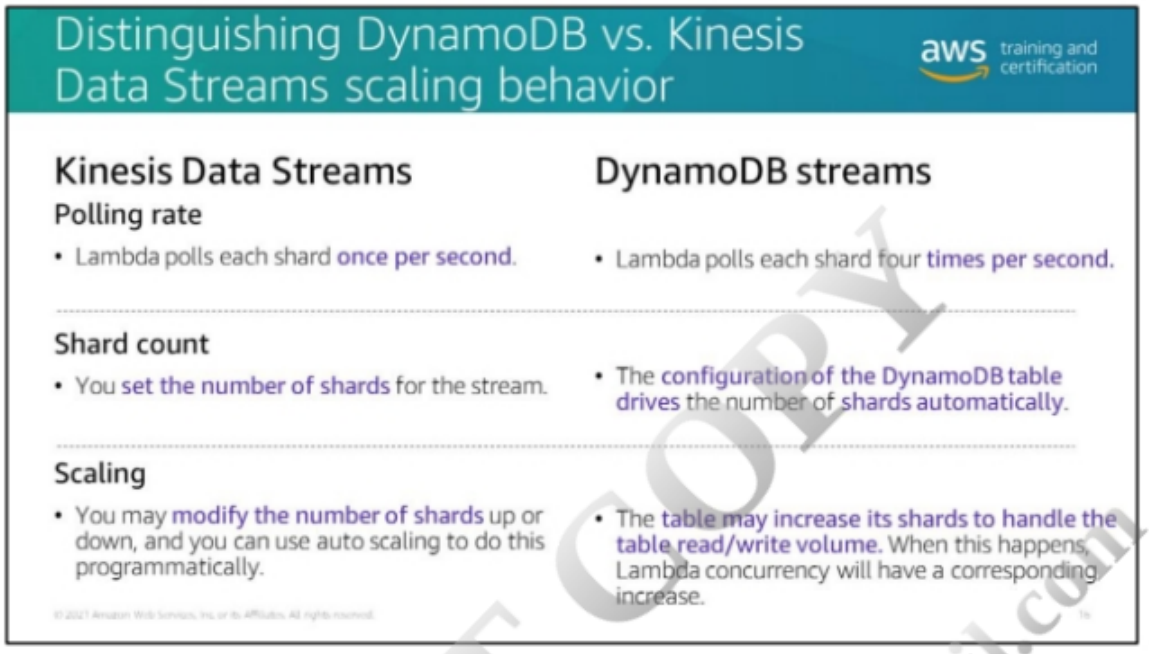
If you don't configure error handling for the stream, a failed record will completely block the shard until it succeeds or until the records reach the end of their retention period.

You learn more about options for handling stream errors in the Writing Good Lambda Functions module.

What you handle with a stream as an event source for Lambda

aws training and certification

What you handle	Best practices
On the stream	
Kinesis Data Streams: The number of shards	
DynamoDB: Read/write throughput and scaling behavior of the table	
On the function	
• Function batch size • Concurrent batches per shard • Batch window	• To optimize for speed, increase the concurrent batches per shard. • To optimize for cost, use a higher batch size or increase the batch window setting.
Error handling options	If the order is important, use checkpointing (requires coding). If the order is not important, use bisect batch on error.
Duplicate messages	Design for idempotency. Include function code that prevents you from processing the same record twice.

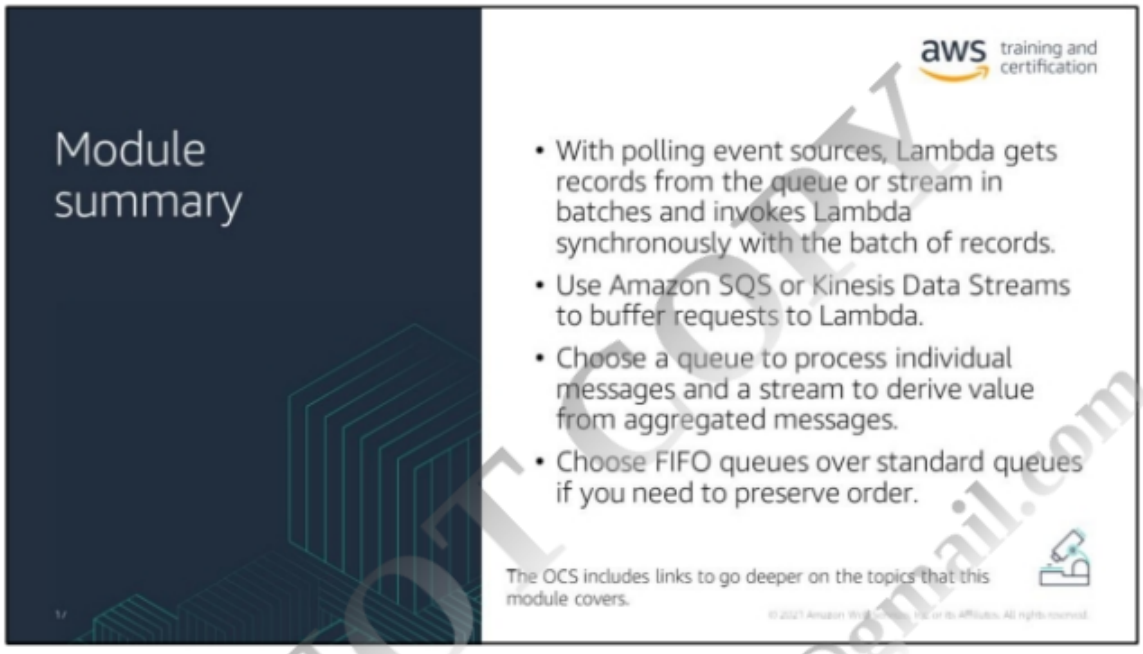


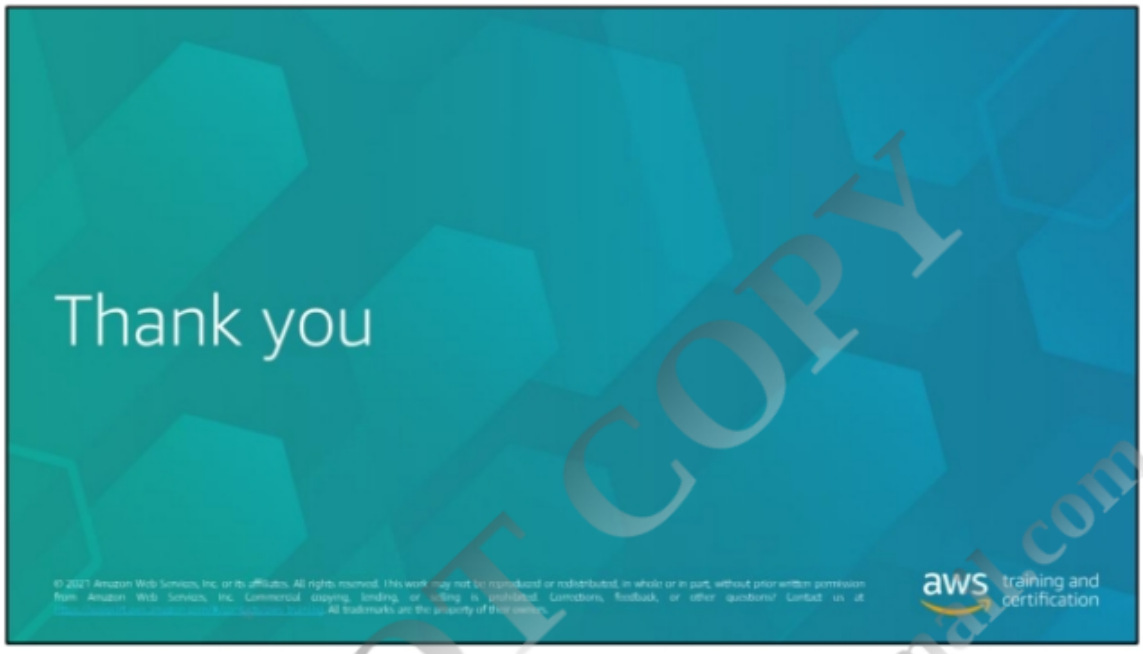
The features you've just discussed apply to both DynamoDB and Kinesis Data Streams.

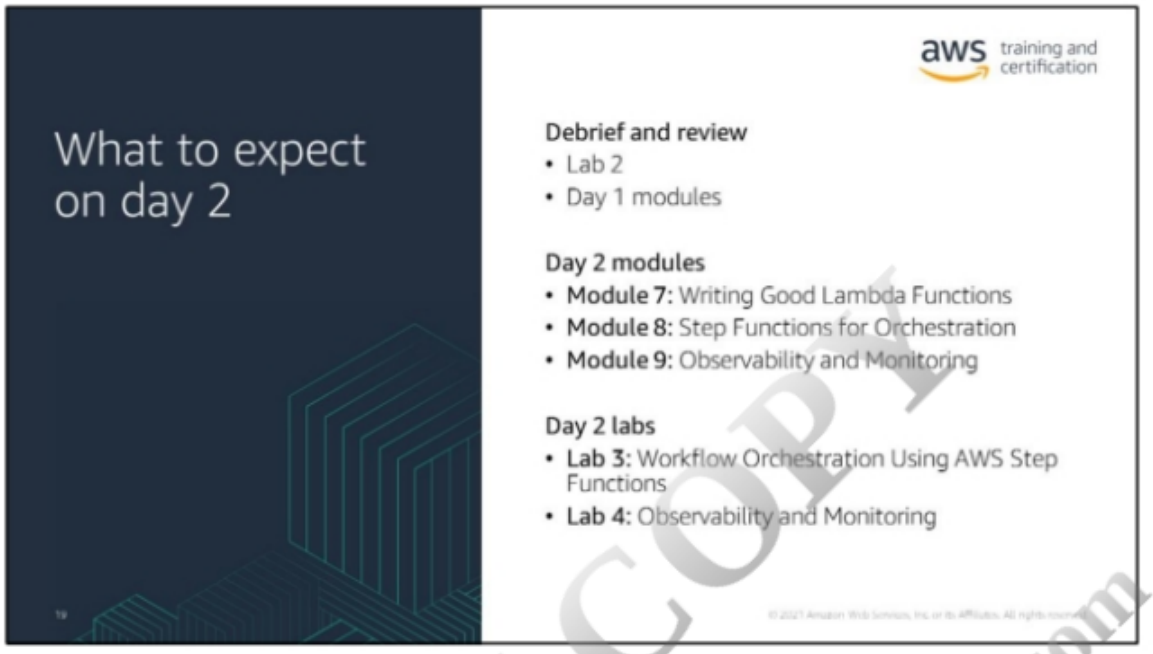
As you've just seen, increasing the number of shards increases the number of concurrent Lambda invocations that are running. It's important to distinguish how your service configuration drives that number.

With Kinesis Data Streams, you control the number of shards, and you control when that number changes.

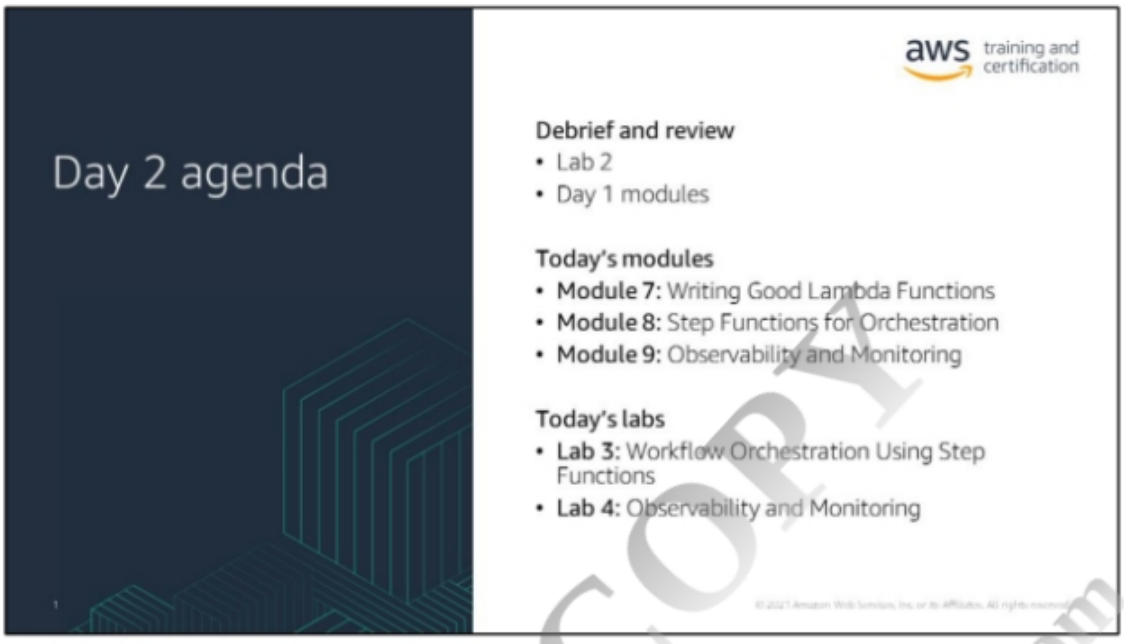
With DynamoDB streams, the way you provision read/write capacity and your scaling decisions for the DynamoDB table you are using drive the number of shards. DynamoDB will automatically adjust the number of shards needed based on the way you've configured the table and the volume of data.







Tomorrow, you start the day with a debrief of lab 2 and a review of material from today's modules.



As you saw yesterday, when you create your function and its associated resources with AWS CloudFormation or the AWS Serverless Application Model (AWS SAM), the function is created as an application in AWS Lambda.

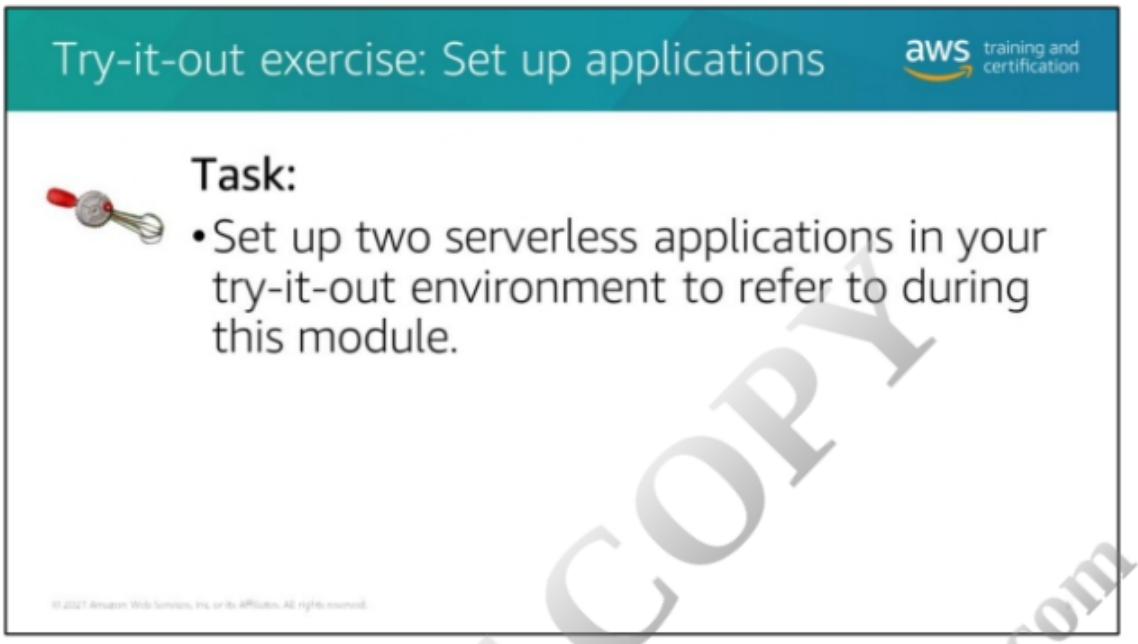
The application page in the Lambda console provides a list of available applications. When you select one, the console provides a logical representation of the resources that are part of the underlying CloudFormation template. This is the same type of representation you would find via the CloudFormation console, and it can be helpful for summarizing what's happening in the template. The application page also provides options for looking at the deployment history and for monitoring each of the resources in the application.

You have three options for creating a Lambda application, all of which directly or indirectly use CloudFormation:

1. Lambda creates an application whenever it is using CloudFormation on the backend. Both AWS SAM and Amplify are transformed to CloudFormation for the actual deployment of your resources.

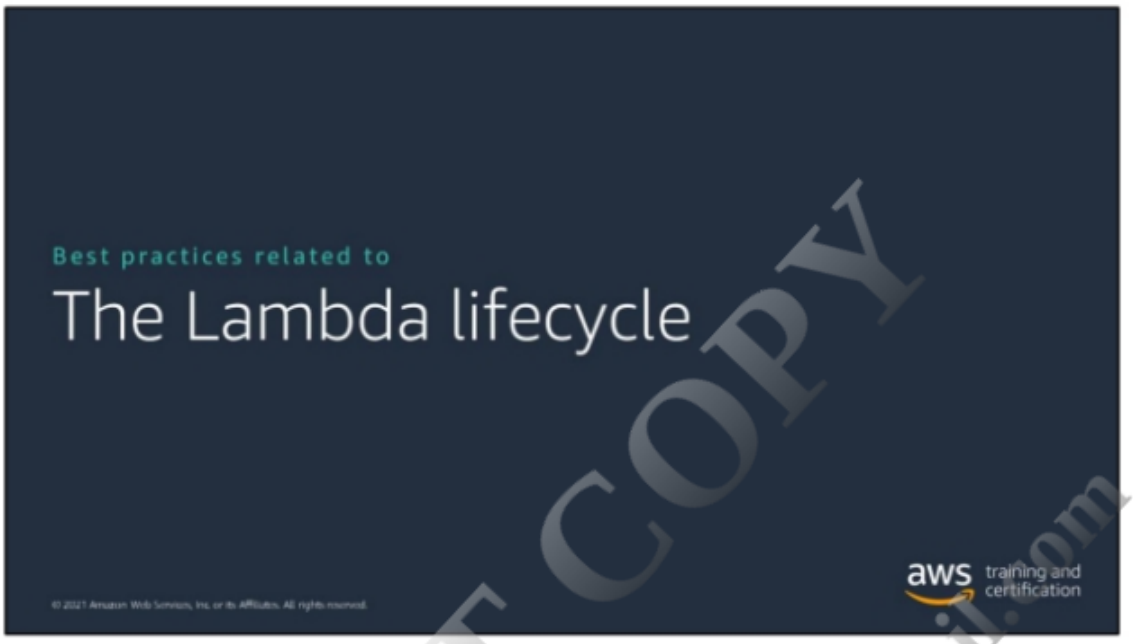
When you select one of these options, a repository and deployment pipeline is created as part of creating your application.

2. If you deploy something you find in the AWS Serverless Application Repository (https://serverlessrepo.aws.amazon.com/applications), this will also create an application in Lambda. These applications use AWS SAM to deploy.
3. You can also select **Create application** from within the Lambda console, and then select from one of the samples, or select **author from scratch**. Your own development environments may use tools outside of the AWS pipeline, but you may find this option helpful for getting started and understanding how all of the pieces fit together.

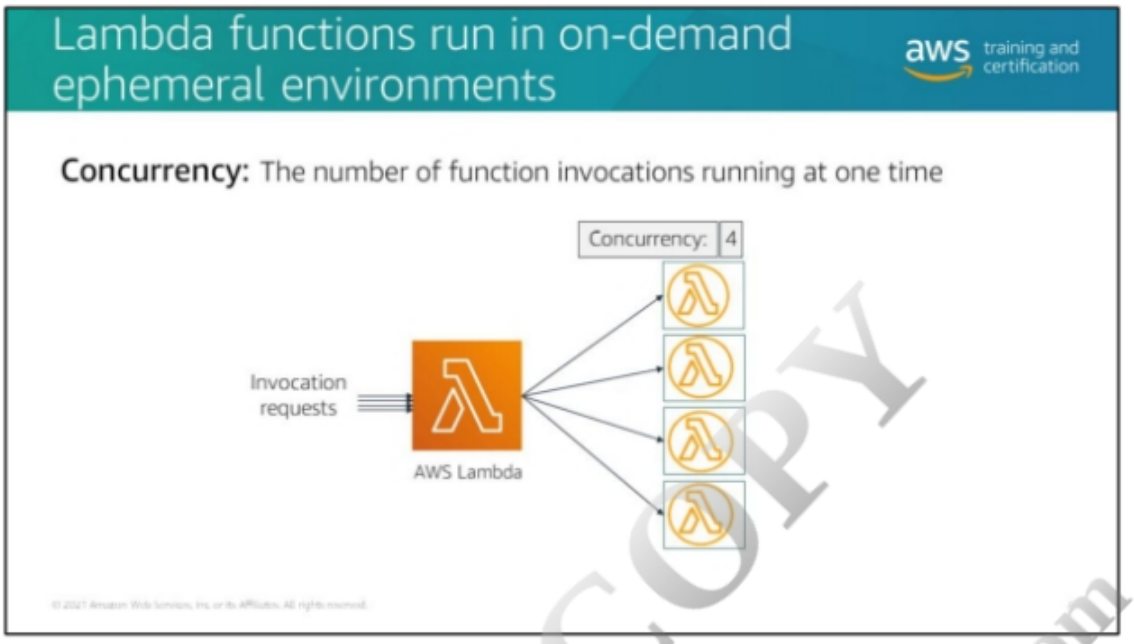


To get started in this module, deploy an application that you can use to illustrate topics that this module discusses later.

This task sets up an example, which you want to have in place for the rest of the module.



While the sample application is being created, let's do a quick review of the Lambda lifecycle and how that contributes to the best practices you will employ when writing your functions.



Each invocation of a Lambda function runs in an ephemeral environment—one that spins up on demand, lasts a brief time, and then is taken down. Lambda manages creating and tearing down these environments for your functions. You don't have direct control over these environments.

Concurrency is the number of Lambda function invocations running at one time. Lambda provides the environments on demand as it receives requests. Through the eyes of the cook in our diner, it's like the number of burners that are going at once. This is how Lambda scales horizontally quickly.

The concurrency that Lambda uses is determined by multiple factors, some of which you have control over, but all of which you need to be aware of as a serverless developer. You will hear more about concurrency in the scaling module.

Factors that influence concurrency include:

Reserved concurrency on a function: This is an optional value set per function that both reserves a subset of the Regional quota for the function and also establishes the maximum concurrent instances allowed for the function. For more information about managing reserved concurrency, visit https://docs.aws.amazon.com/lambda/latest/dg/configuration-concurrency.html#configuration-concurrency-reserved.

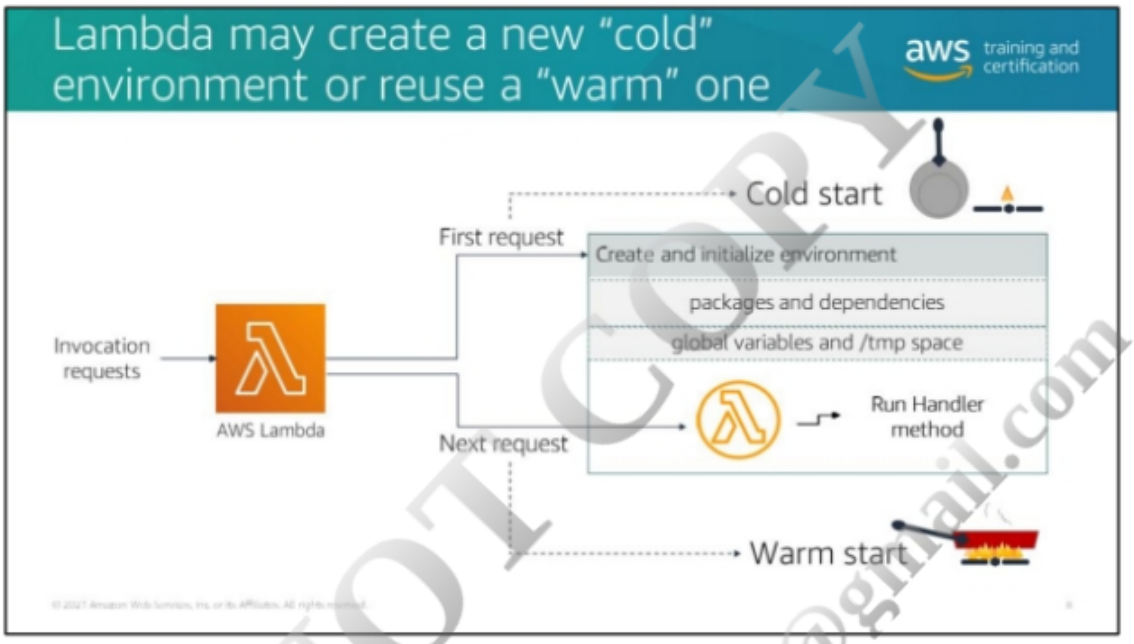
Regional quota: This is the total number of invocations that can run concurrently across all Lambda functions within an account by Region. AWS sets this as a soft quota on the account. For more information about quotas, visit https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html.

Burst quota: Each Region has a burst limit, which prevents concurrency from increasing too quickly in the event of a large spike of requests in a very short time period. You cannot modify this limit. We'll talk about this more in the scaling module. For more information about bursts, visit https://docs.aws.amazon.com/lambda/latest/dg/invocation-scaling.html.

Request rate and function duration: This is the rate at which new invocation requests for the function arrive, in conjunction with how long it takes Lambda to run the function.

The event source:

- Lambda adds concurrency up to applicable quotas to keep up with requests for synchronous and asynchronous event sources. Multiply your request rate by average function duration to determine the concurrency your function will use.
- Lambda increases and decreases polling rates/concurrency when Amazon Simple Queue Service (Amazon SQS) is the event source based on queue depth and error rates. Lambda uses one concurrent invocation for each parallel process that it uses to retrieve messages from the SQS queue, starting with a default of five concurrent invocations.
- Streaming sources set concurrency based on the number of shards on the stream and the parallelization settings on the function.



The first time your function is invoked, Lambda creates an execution environment in which to run it. Lambda downloads your code into the environment and initializes your runtime, as well as any packages, dependencies, or global variables that your function includes. The environment also initializes temp space (/tmp) for your function. When the environment is ready, Lambda runs your function starting from the **Handler** method.

When that invocation is complete, Lambda can reuse that initialized environment to fulfill the next request, if the next request comes in close behind the first one. So, the second request skips all of the initialization steps and goes straight to running the handler method of your function, using the /tmp space and the global variables as they were set during the prior run.

In this example, the first request got a "cold" start, and the second one got a "warm" start.

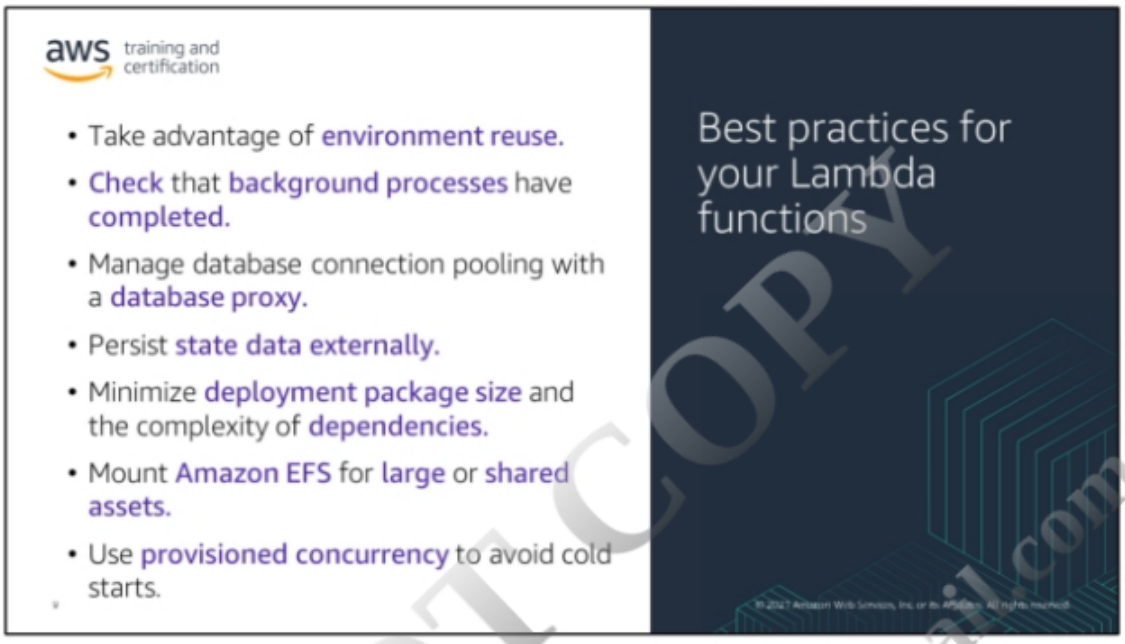
What does this look like in the diner analogy?

- The first time someone orders eggs in the morning, the cook has to turn on the burner and heat the cold pan before cooking the eggs.
- If another order comes in just after the first order is finished, the cook can use the already-warm pan to cook the second order. The eggs will be ready faster than the first ones because the cook didn't need to wait for the pan to warm up.
- A cold start could happen if the function hasn't been run in a while. Lambda "breaks down" an environment when it's not receiving additional requests. (It's 2 PM and there haven't been any orders in the last hour, so the cook turns the burner off.)
- A cold start can also occur when Lambda needs to add concurrency to keep up with requests. (The cook needs to turn on a second burner/pan because there is another order for eggs before the first ones are finished.)

In the pictured example, the concurrency is 1 because the first invocation is complete before the second request begins. But if the second request had come in while the first one was still running, Lambda would have to spin up a second environment, making the concurrency 2, and both would have received a cold start.

If you have a steady, predictable request rate, you can go a long time without a cold start. New requests get routed to environments that have just been used by a previous request. But if your request rate is prone to spiky or unpredictable traffic, you need to accept or mitigate the potential for cold starts.

Another potential challenge of this model is providing everything your function needs within the ephemeral environment with a limited amount of temp space and no ability to use external resources without loading them into the environment as part of initializing it. We'll talk more about the primary way to mitigate cold starts—provisioned concurrency—in a few minutes.

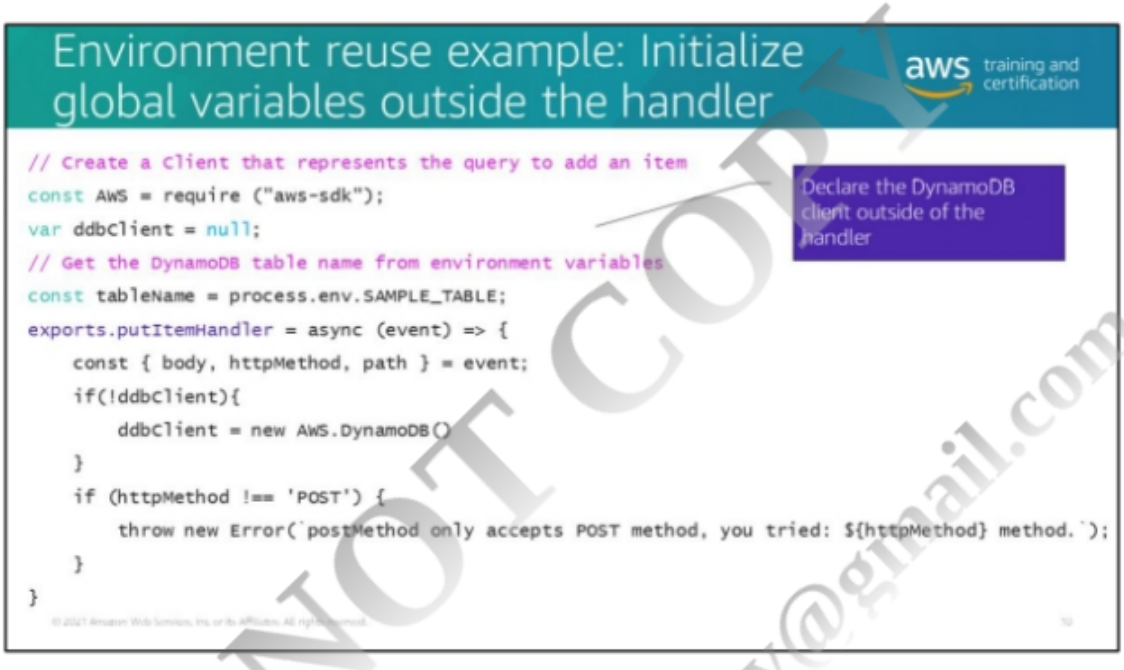


This model presents a few new wrinkles for developers starting to think serverless. You need to:

- **Code for ephemeral environments:** You don't have direct access to the environment in which your code runs, and Lambda manages creating and breaking down new environments. You can't directly route invocation A to a warm environment, for example.
- **Write stateless functions:** Each Lambda function runs in its own environment, so you can't share information directly between function invocations.
- **Optimize the speed to initialize and run your code and minimize the impact of cold starts:** The size and speed of function resource initiation is much more important than in a server-based setup. You need the startup time of a new environment to be as quick as possible so that your functions run as quickly as possible. This is important for both cost and performance.

Six best practices that you should apply to your functions based on these characteristics:

1. Write your functions to take advantage of a warm start, when you get one, so that you don't waste time re-initializing resources that are already there. But be sure not to leave a process running from one invocation to another, because the warm environment will keep running that process when a new invocation comes in.
2. Configure a database proxy to manage database connections.
3. Make your functions stateless, using external data stores like Amazon DynamoDB to persist state data.
4. Make your deployment packages as small as possible, and limit your dependencies, so your functions initialize as quickly as possible.
5. Attach Amazon Elastic File System (Amazon EFS) storage to functions where you need to work with large binaries that can't fit into the /tmp space limit. You can also use Amazon EFS to share large libraries or other resources across invocations where it isn't practical or possible to load them into the execution environment.
6. To reduce or possibly eliminate cold starts, use a feature called provisioned concurrency to keep a desired number of environments "always warm."

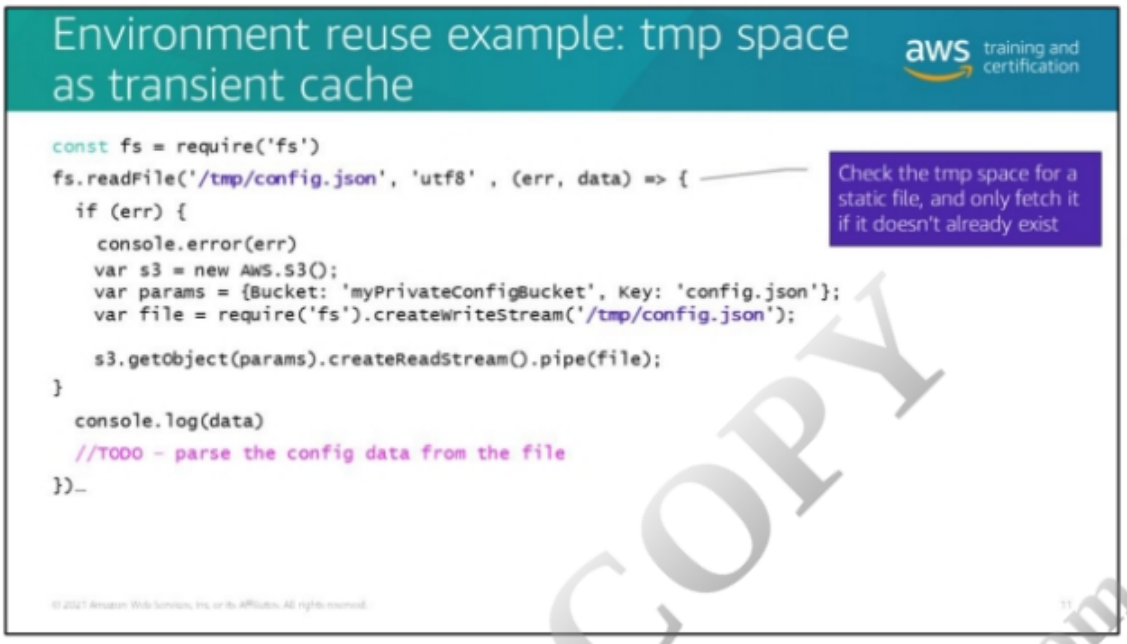


One example of environment reuse is to put globals outside of the handler.

If your code retrieves any externalized configuration or dependencies, make sure they are stored and referenced locally after initial run. Limit the re-initialization of variables or objects on each invocation. Any declarations in your Lambda function code (outside the handler code) remain initialized when the function is invoked again.

You should also add logic to your code to check whether a connection already exists before creating one. It a connection exists, reuse the existing one.

In this Node.js example, the DynamoDB client is declared outside of the handler, making it available to the current invocation as well as to any that are routed to the same environment while the invocation is still warm.



Another way to take advantage of environment reuse is to add code to check whether the local cache has the data that you stored. Each invocation environment provides a small amount of disk space in the /tmp directory, which remains in the environment.

This is a Node.js example where the function looks for config.json in the tmp directory. If this is a warm environment, the file does not need to be reloaded. If there is an error finding the file in the tmp directory, the function loads it.

Example: Check for background processes

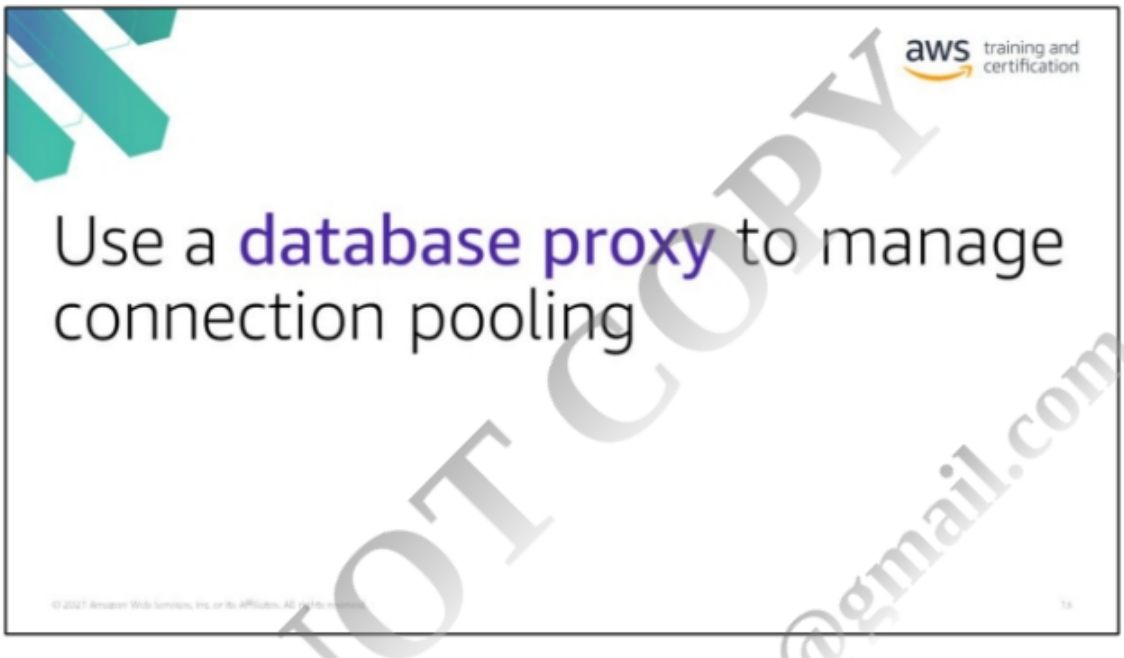
aws training and certification

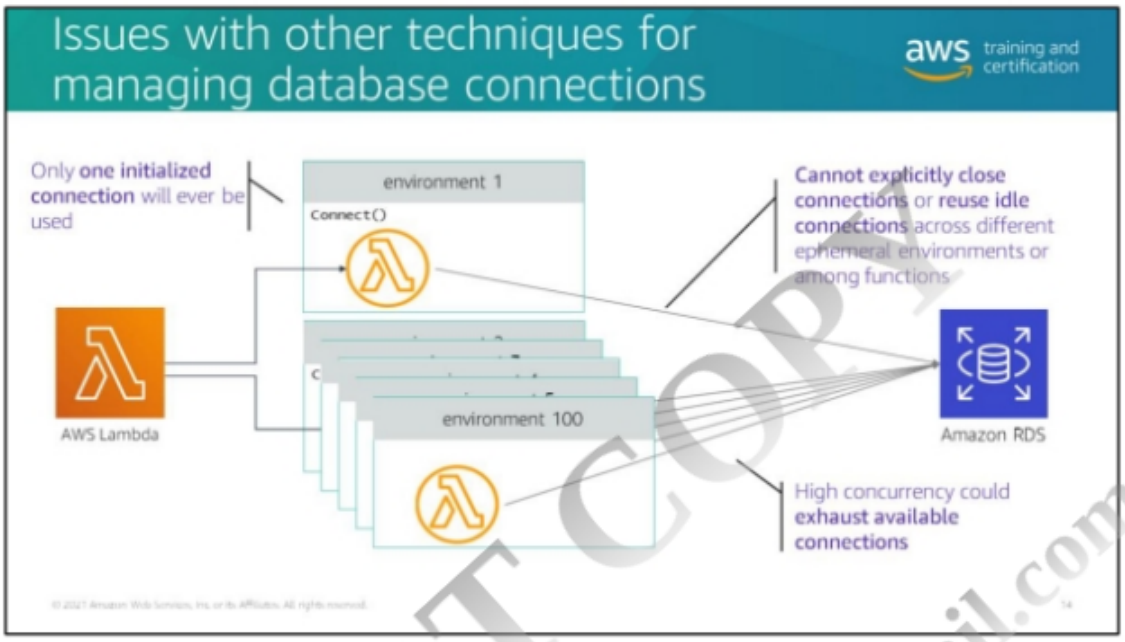
```
const AWS = require ("aws-sdk");
const ddbClient = new AWS.DynamoDB();
…
let tables = ddbClient.listTables();
// CALLBACKS ddbClient.listTables({}, function(err, data){ console.log
(data.TableNames) })
// PROMISES let tablesPromise = ddbClient.listTables().promise();
tablesPromise.then((data) => { console.log(data) })
// Async/Await getTables() async function getTables(){ let tableNames = await
ddbClient.listTables().promise(); console.log(tableNames); }

console.log(tables);…
```

Node.js options to check for completion

If you don't wait for a background process to end before exiting your function, the background process continues to run while that environment exists. So an invocation that follows and gets that warm environment could get unexpected results from a process that began running during the previous invocation.

Use the constructs within your chosen language to check for completion. This example lists three options available in Node.js.



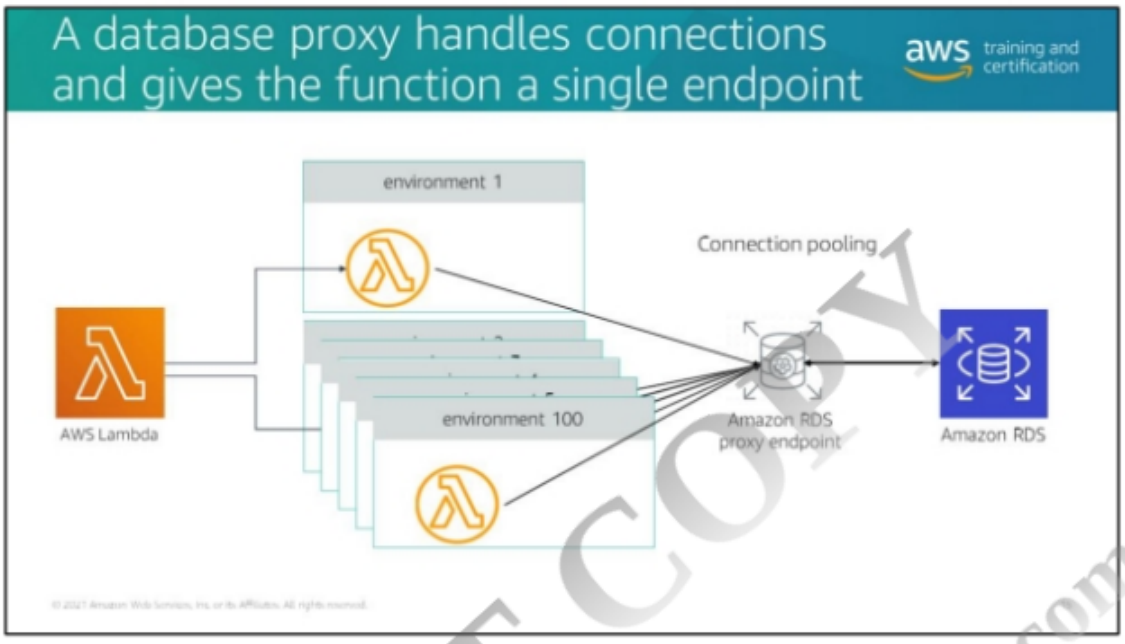


Techniques you might use to manage database connections have limited value when applied to the Lambda lifecycle and its ephemeral environments.

For example, you might want to initialize a number of connections on startup for your application to use. With Lambda, you could initialize a connection outside the handler and then check for it in your code, making it available to subsequent invocations that get a warm start in that environment. But Lambda never runs more than one invocation at a time within that environment. So, you have no reason to open anything more than a single connection as part of initializing your function.

A separate challenge: You can't explicitly close connections when an environment gets recycled because there is no hook to let you indicate destruction of a Lambda environment. You can use the database Time to Live (TTL) as a fallback to clean up connections, but this can still lead to session leakages. Because you have no control over the lifecycle of the invocation environments, you could have connections sitting idle. And because you can't share environments with two different Lambda functions, you can't reuse idle connections across functions.

You also have the potential of Lambda's concurrency increasing beyond the capacity of available database connections. You would need to find a way to throttle the concurrency to match the available connections. You can use reserved concurrency at the function level to limit the number of potential connections that Lambda would attempt to create, but it may be difficult to plan the right level to set, especially if you need to segregate connections for different applications.



A database proxy can provide connection pooling in addition to other benefits. This allows your functions to interact with the Amazon Relational Database Service (Amazon RDS) proxy endpoint, and the proxy manages the incoming connections to the database as needed. This is much more efficient for functions where you anticipate high levels of concurrency. AWS offers database proxy on Amazon RDS MySQL and Postgres, as well as on Amazon Aurora. For more information about connection pooling, visit https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/rds-proxy.html#rds-proxy-connection-pooling.

Like other features, weigh the cost versus benefit to your workload. If you won't get high levels of concurrency competing for your database connections, you may want to rely on the strategies noted on the previous slide—initialize and reuse a connection, using reserved concurrency to limit connections and TTL settings to close idle connections.

If you use database proxy, there is a setting for what percentage of connections you want the proxy to consume. The proxy will check the database to see the maximum connections it's set to handle. For example, if the database is set to handle 200 connections, the proxy will take all of them if you don't adjust the percentage. The proxy scales the number of connections it opens to Amazon RDS. It could be more than one, but it won't be more than one per Lambda function.

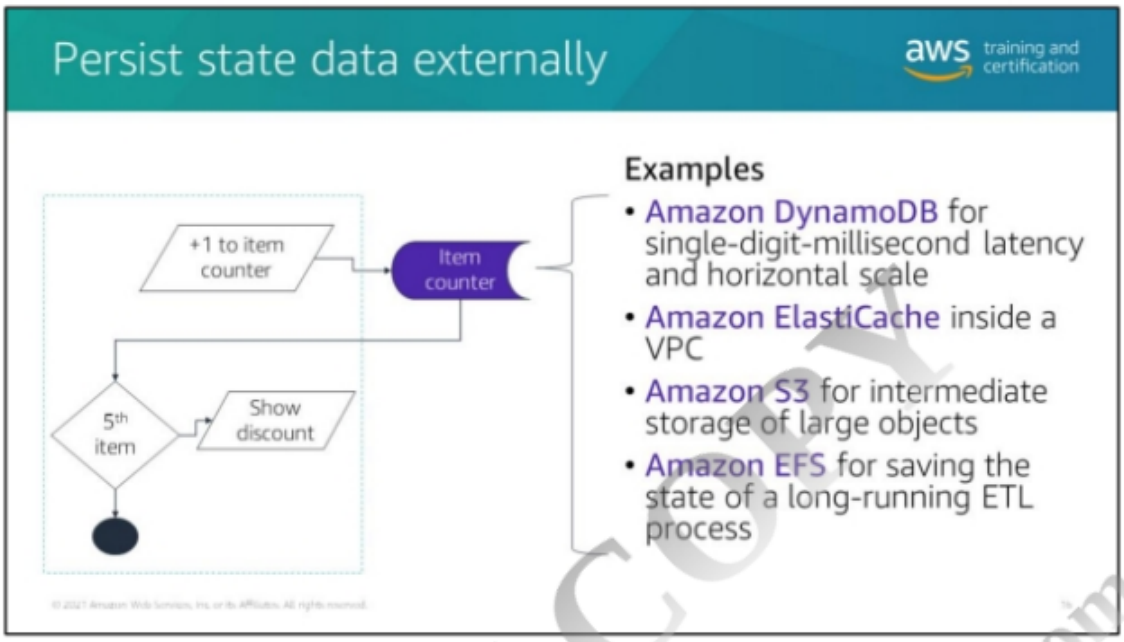
Don't let the pool take everything. You need to leave a percentage for other activities, such as management. Think about what kind of backends you are using. If your database is shared by other applications, you need to consider the connections those will need.

You might also choose to set up multiple proxies, for example, to use separate credentials or balance out use cases.

Understand the percent of available connections that make sense. Key monitoring considerations include the Amazon RDS Proxy metric, database connections, and client connections.

For more information about Amazon RDS Proxy, visit https://aws.amazon.com/rds/proxy/.

For more information about monitoring Amazon RDS Proxy using Amazon CloudWatch, visit https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/rds-proxy.html#rds-proxy.monitoring.



As noted earlier, take advantage of the warm start when available, but you can't assume that you will get a warm start. Therefore, you can't assume that any data you write to the /tmp directory will remain available.

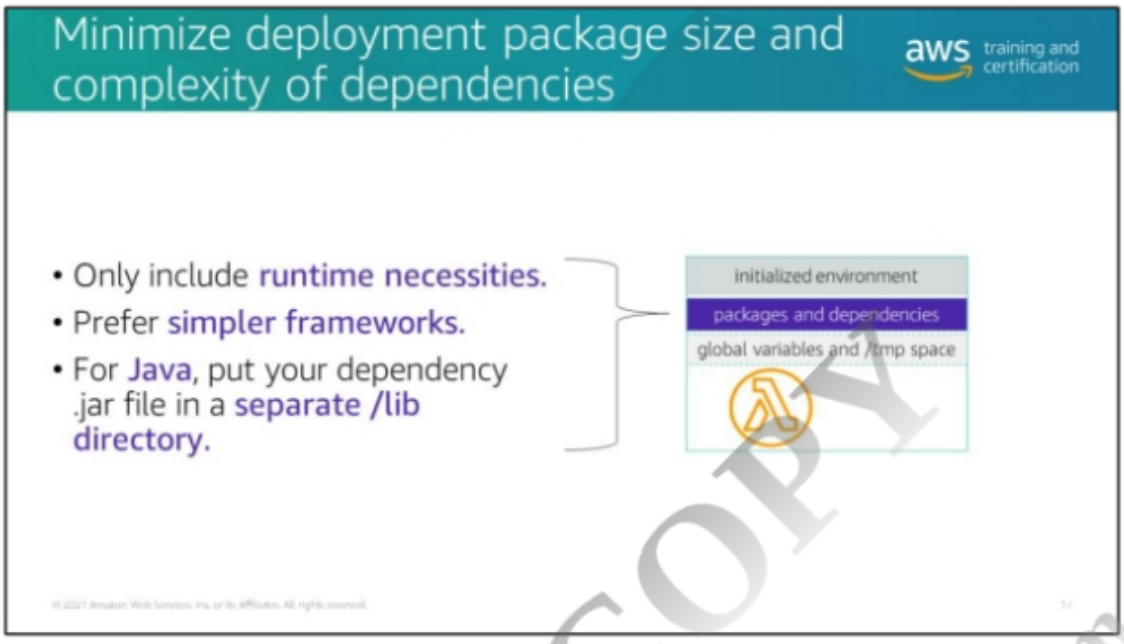
Let's say you have a function that adds an item to a shopping cart and increments a counter for items in the cart. If you store the value of that counter locally and the next invocation gets a cold start, the local counter will have been reset, and you would not be able to maintain state properly. If you need to maintain state, persist data using an external store.

DynamoDB is serverless and scales horizontally to handle your Lambda invocations. DynamoDB also has single-digit-millisecond latency, which makes it a great choice for storing state information.

If you have to put your Lambda function in a VPC, you might consider using Amazon ElastiCache, which may be less expensive than DynamoDB and lets you use features like sorted sets.

You can use Amazon Simple Storage Service (Amazon S3) as an inexpensive way to persist large objects that your function uses; for example, intermediate storage of images that you are processing through a set of steps.

You can use Amazon EFS for saving state and sharing it across function invocations; for example, a long-running ETL process. (More about Amazon EFS in a moment.)



Your function's code consists of scripts (Python) or compiled programs (Java, C#) and their dependencies. When you author functions in the Lambda console or with a toolkit, the client creates a ZIP archive of your code. This is the deployment package. When you manage functions with the Lambda API, command line tools, or SDKs, you create the deployment package.

You need to create a deployment package manually for compiled languages and to add dependencies to your function.

When you invoke your function, the deployment package is downloaded, unpacked, and loaded into your invocation environment. Minimizing the package helps shorten cold starts and reduces your cost because you are billed from the point at which Lambda initializes the runtime in your environment.

Minimize your deployment package size to its runtime necessities. This reduces the amount of time it takes for your deployment package to download and unpack ahead of invocation. This is particularly important for functions authored in compiled languages.

Avoid including the entire AWS SDK library as part of your deployment package. Instead, selectively depend on the modules that pick up components of the SDK that you need, such as DynamoDB, Amazon S3 SDK modules, and Lambda core libraries.

We'll talk more about Lambda layers in a bit, but it's a best practice to bundle the SDK in a layer with your function.

For Java, put your dependency .jar files in a separate /lib directory to reduce the time it takes Lambda to unpack deployment packages.

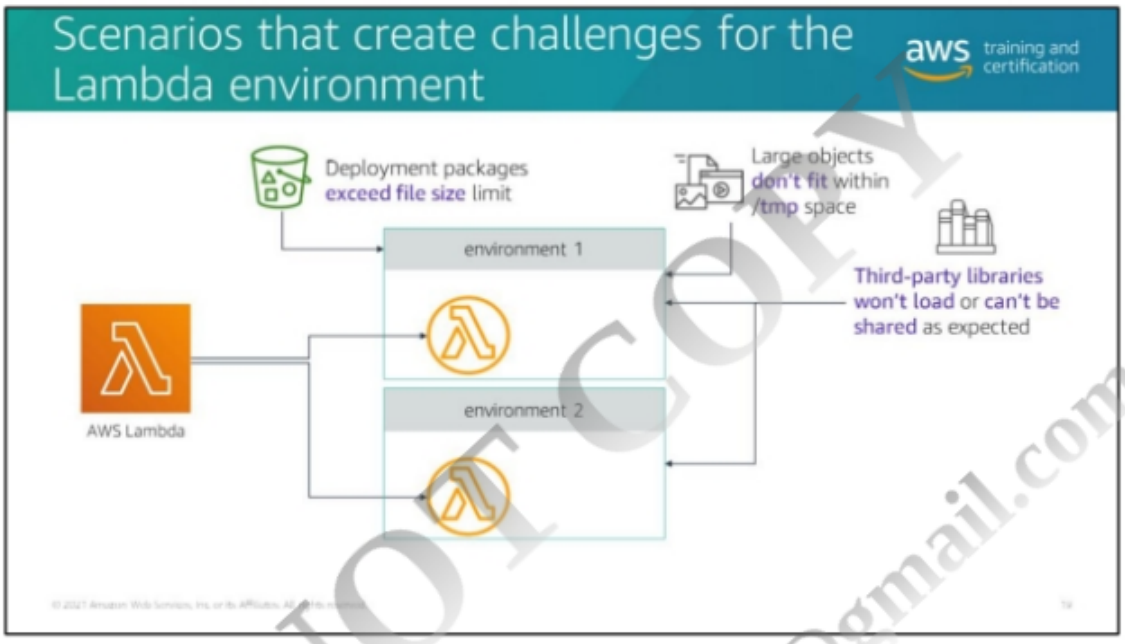
Minimize the complexity of your dependencies. Prefer simpler frameworks that load quickly on startup. For example, prefer simpler Java dependency injection (IoC) frameworks like Dagger or Google Guice, over more complex ones like Spring Framework.

This best practice is most important when using heavier languages like Java or .NET.

For details about deployment package size quotas on the Lambda quotas page, visit https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html.

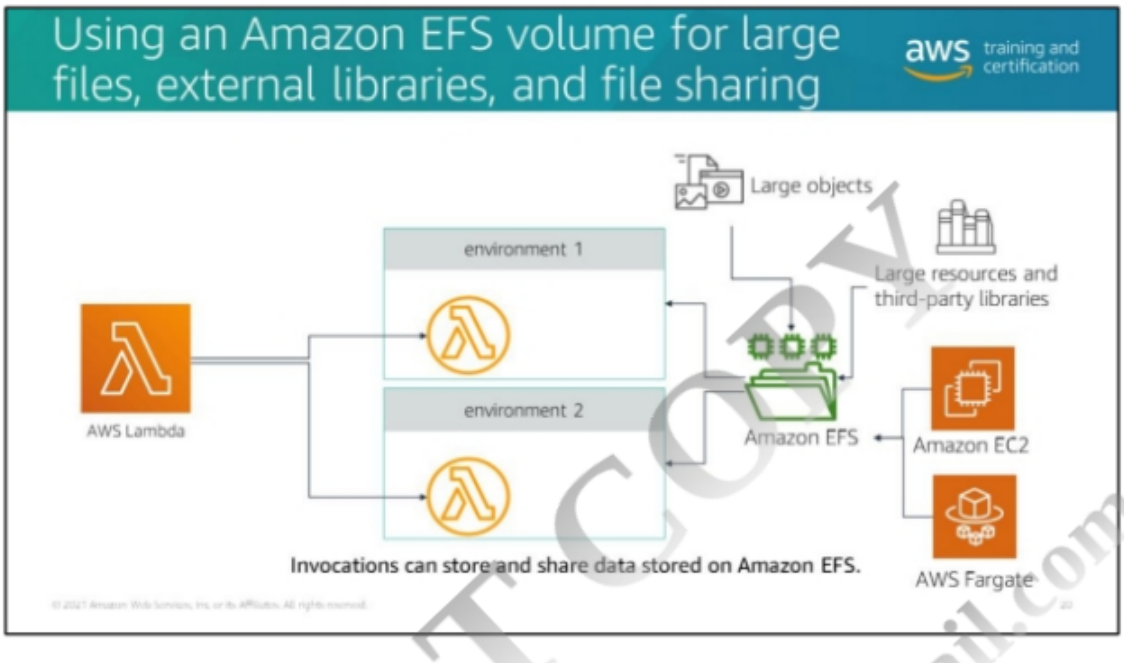
For more information about deployment packages and links to specific instructions for each runtime, visit https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-features.html#gettingstarted-features-package.

For tips on minimizing your startup time with Java, visit https://aws.amazon.com/blogs/developer/tuning-the-aws-java-sdk-2-x-to-reduce-startup-time/.



The size and nature of the Lambda invocation environment makes some use cases very difficult.

For example, you can't get your deployment package small enough to meet the size limit or you need to work with very large objects like media files or machine language models that can't be loaded into the temp space. There are also third-party libraries that can't be loaded and shared in a Lambda invocation environment the way they would if you ran things from a server. And, as noted earlier, you can't directly share data.



For more information about using Amazon EFS with Lambda, visit https://docs.aws.amazon.com/lambda/latest/dg/services-efs.html.

Mounting an Amazon EFS volume addresses the challenges listed on the previous slide. You can store large objects and third-party libraries on Amazon EFS and access them from within your Lambda function.

Using Amazon EFS has another advantage. It gives you the ability to write data to Amazon EFS and share it between invocations. This lets you use Amazon EFS for persisting data that is updated in one invocation and used on the next invocation. Writing to Amazon EFS requires no special coding. You just use the same command you would use to write a file to block storage.

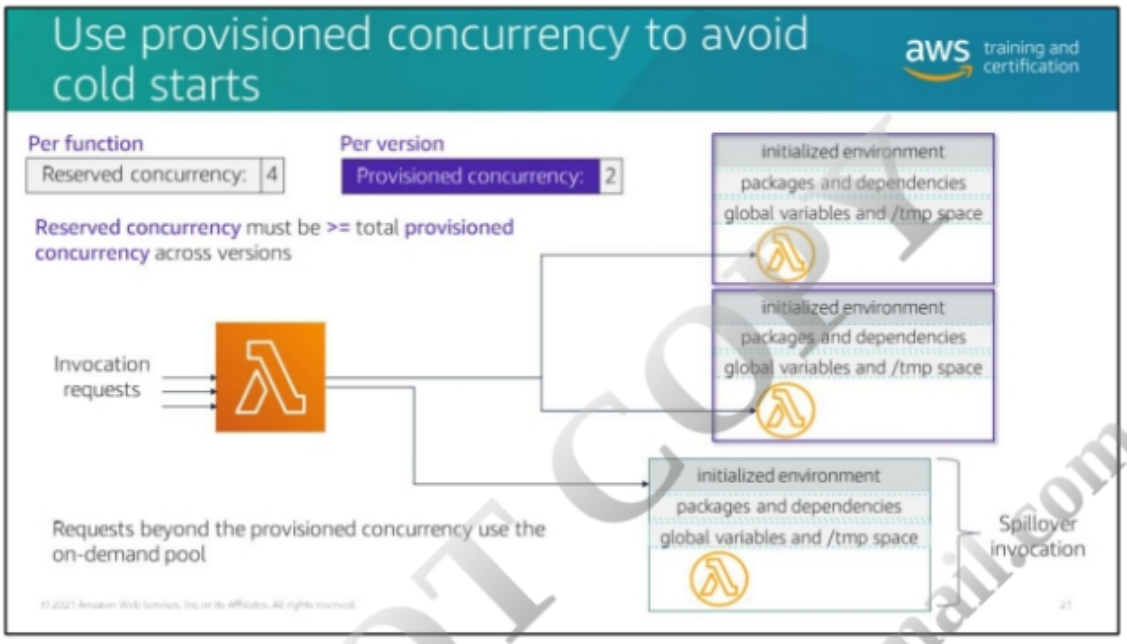
You can use other sources to update the data on Amazon EFS as well (AWS Fargate or Amazon Elastic Compute Cloud [Amazon EC2], for example). This makes hybrid scenarios much easier and extends the workloads you can run with Lambda.

For a quick primer on Amazon EFS, visit https://www.aws.training/Details/Video?id=30976.

The ability to attach Amazon EFS volumes and share them among invocations can simplify architectures and may reduce the cost of processing these workloads. For example, it's much easier to deal with very large files for manipulating data or processing video when you can access block storage. It's easier to break up parts of a file for parallel processing using Amazon EFS versus an object store like Amazon S3. You could also combine compute sources; for example, using Fargate to break the file up and then having Lambda complete processing.

With artificial intelligence (AI) and machine learning (ML) workloads, the models you're training are large, and even if you could get them to fit within Lambda's memory or /tmp space, you would spend runtime while they are downloaded. With Amazon EFS, you save time and can access much larger packages.

With real-time apps, without Amazon EFS, a typical workflow would be to store data in Amazon S3 and move it to ElastiCache for faster read access. This pattern works, but replacing the Amazon S3 and ElastiCache storage with Amazon EFS storage reduces the complexity and lowers the cost.



You can configure some portion of your function's **reserved concurrency** as **provisioned concurrency** rather than on-demand concurrency. Lambda initializes the number of environments that you have configured, and they will always be "warm," even if the function hasn't been invoked for a while.

In our diner analogy, this is like keeping a couple of pans warm on the stove in anticipation of future orders.

In this example, with provisioned concurrency set at 2, the first two invocations of that function both get warm starts.

Provisioned concurrency can alleviate the burden of handling cold start latency in your application, but there is a cost to keep the environments provisioned. Therefore, you want to find the right balance based on the workload that you have and the criticality of having a warm start for your function.

If Lambda requests more invocations than the provisioned concurrency that you set, those requests go to the on-demand pool and are handled as described earlier. The request might get a warm start if an environment is available, or they might get a cold start.

If you have a provisioned concurrency of 2, that means there will always be two provisioned environments for your function.

A third request that comes in in before the first two invocations are completed would still spin up a third concurrent environment. Assuming this represents all of the recent invocations, the request would get a cold start. The third request is invoked in an on-demand environment.

These invocations, routed to on-demand environments, are reflected in the **spillover** metric **ProvisionedConcurrencySpilloverInvocations.**

If Lambda receives more than four invocation requests while the initial four are still running, the additional requests are **throttled** based on the **reserved concurrency** of 4.

We'll talk more about how to decide the reserved and provisioned amounts in the scaling module, but you will want to track these Lambda function metrics:

- Invocations
- Throttles
- ProvisionedConcurrencyInvocations
- ProvisionedConcurrencySpilloverInvocations

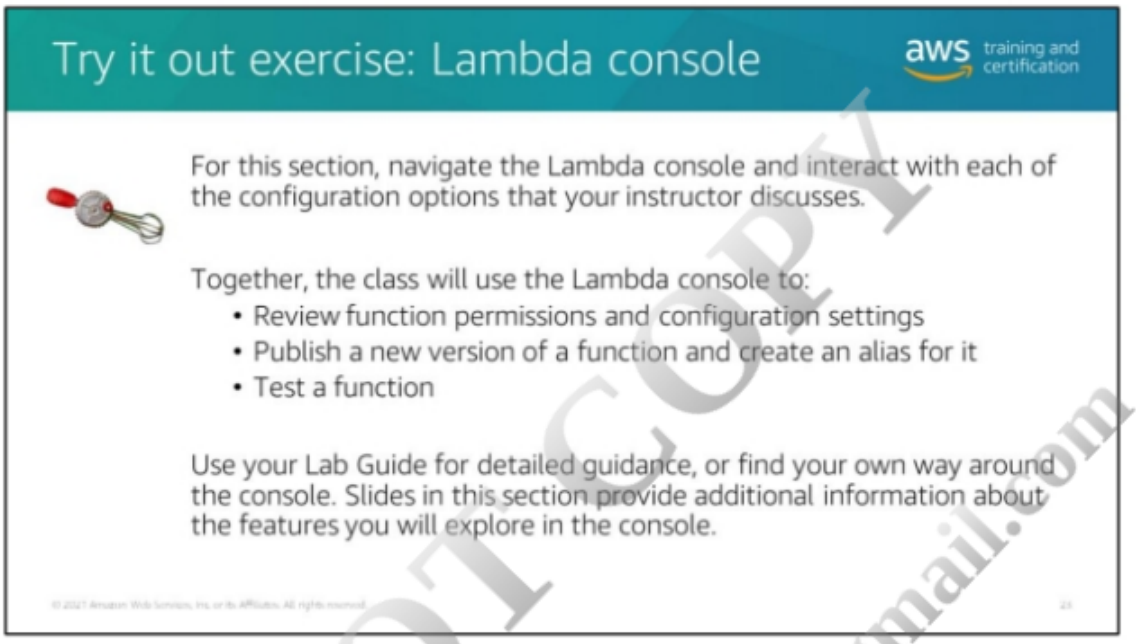
For more information about working with Lambda function metrics, visit https://docs.aws.amazon.com/lambda/latest/dg/monitoring-metrics.html.

It's important to keep in mind how reserved concurrency and provisioned concurrency are handled per function. Reserved concurrency is set at the function level and applies **across all versions** of that function. Provisioned concurrency is associated with a **particular version of the function**.

For example, if you set reserved concurrency to 4 and are running requests against two versions of a function in a traffic-shifting scenario, the total provisioned concurrency between those two versions cannot be more than 4. More information about versions and aliases is coming in this module.

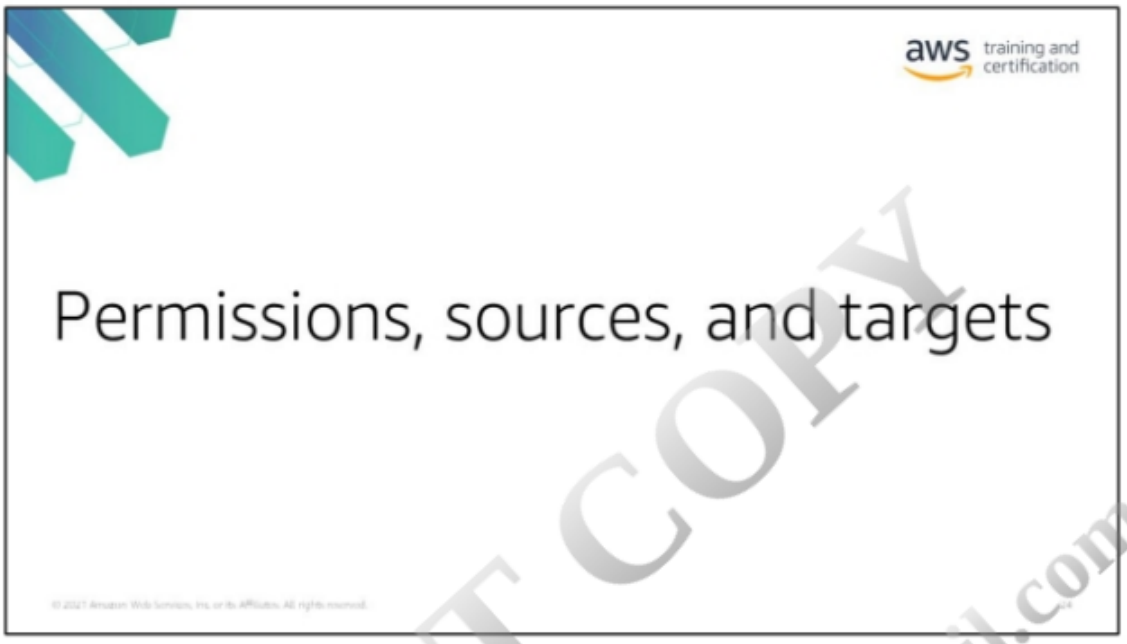
The "spine" of your Lambda function runs from the event sources that trigger it, to the function code that runs when its triggered, to the target resources that the function interacts with. You also configure other attributes of the function that define how it behaves. Together, your configurations and code are your Lambda function.

You can use the Lambda console to configure your functions, or you can use the Lambda API to update your settings.



In this section, use the try-it-out (TIO) environment to navigate through the Lambda console and review the settings that your instructor is demonstrating. Select the Lambda service from the AWS Management Console to get started.

Steps are listed in the Online Course Supplement. Slides marked with the egg beater icon used on this slide are associated with steps in the tour.



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.

You need to set up permissions when you create a function.