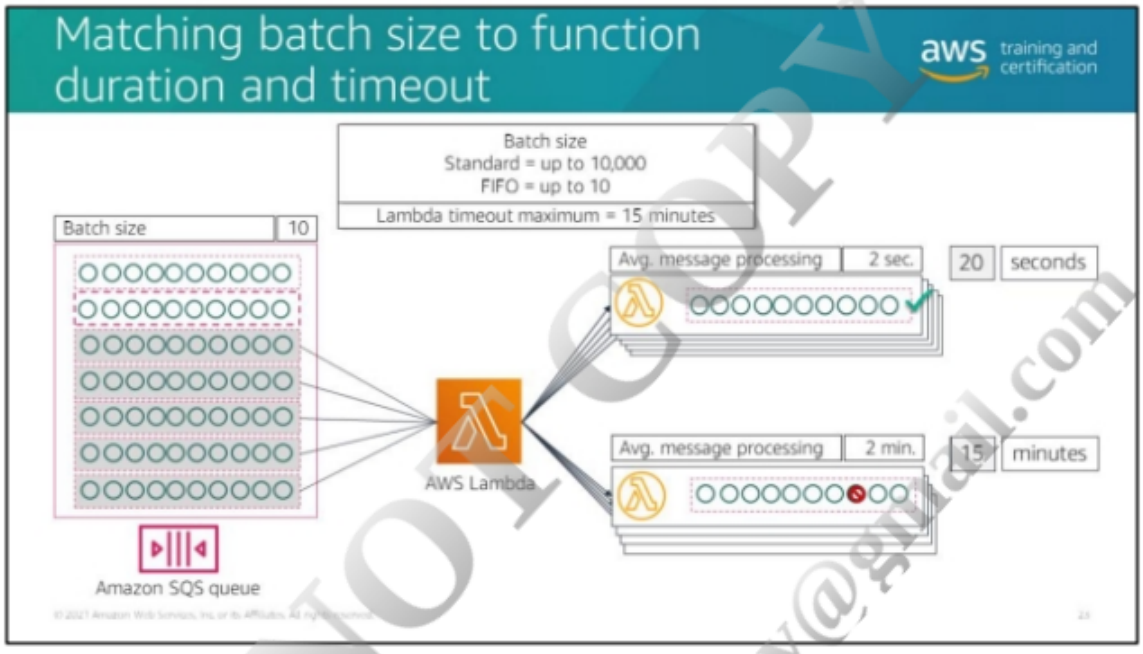


You can set the batch size for a standard queue up to 10,000 messages per batch. For a FIFO queue, the maximum is 10. Using a larger batch size means you can grab more messages off the queue each time, which can reduce polling costs and improve efficiency for fast workloads. For example, a batch size of 10 requires fewer polling processes and fewer invocations than a batch size of 3.

Choose a function timeout that lets you process all of the messages in a single batch before you hit the function timeout so that the batch doesn't error out.

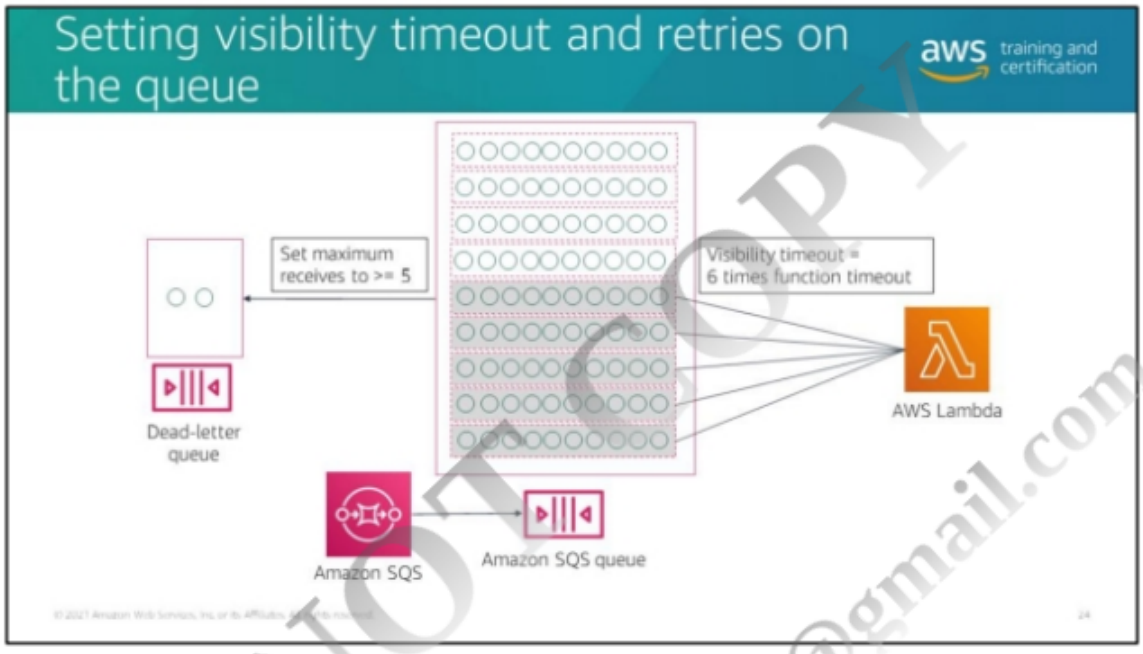
You also want to balance that with preventing stalled invocations from incurring additional costs or creating a bottleneck.

Consider two examples:

- In the first, you have a function that typically processes each message in 2 seconds. With a batch size of 10, the function would successfully process a batch of 10 in about 20 seconds on average. You want a function timeout higher than that to provide some buffer while limiting the impact of a stalled invocation, so maybe you set a function timeout of 40 seconds.
- If the function takes 2 minutes to process each message, a batch of 10 would never successfully complete because the maximum Lambda timeout is 15 minutes.

You either need a smaller batch size or faster per-message processing.

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



On the queue itself, the options that you reviewed as part of error handling also play a part in handling high volumes of requests.

The visibility timeout tells the queue how long to make messages invisible to other consumers once they have been picked up by a consumer. This helps to avoid cases where records are processed more than once. Your visibility timeout should allow a significant buffer to account for situations in which Lambda is throttling invocations and continues trying to invoke the function for some period.

If the visibility timeout expires before your Lambda function has processed the messages in that batch, any message in that batch that your function hasn't deleted will become visible again. Those messages will get processed again, which will also increase the queue depth. The recommended best practice is to set your visibility timeout to at least six times your function timeout.

Sticking with the prior example of a batch size of 10 and batches that process in about 20 seconds each, with a function timeout of 40 seconds, you would want to set the visibility timeout to 40 * 6 or 240 seconds.

For messages that fail processing, it is a best practice to set up a dead-letter queue on the queue and set a redrive policy. The redrive policy includes the maximum receives for a message on the queue. You want to strike the right balance of retrying messages and keeping processing moving to avoid a queue backup.

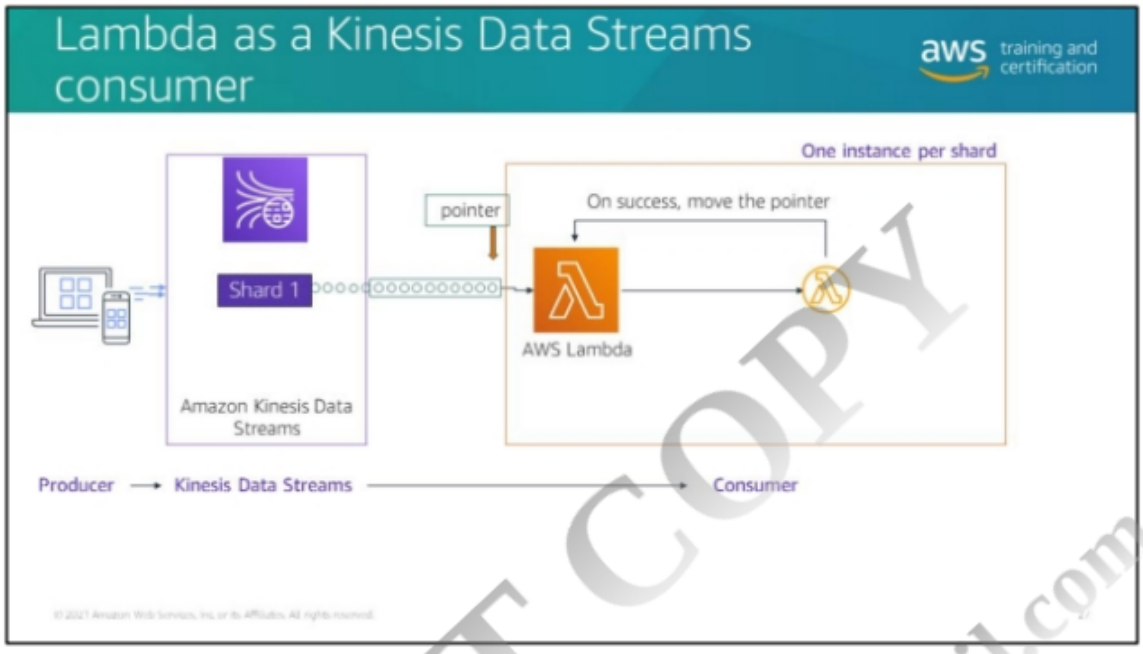
Lambda and Amazon SQS metrics that may indicate a scaling issue

aws training and certification

Service	Metric	What to look for
Lambda	`Duration`	Lambda function duration average is close to or equal to function timeout
Lambda	`Errors`	Lambda function error rate is increasing

Lambda together with Amazon SQS	`Invocations`
	`ConcurrentExecutions`
	`ApproximateNumberofMessagesNotVisible`
	`ApproximateNumberofMessagesVisible`
		Invocations or number of messages in flight is decreasing while the queue depth is increasing

Amazon SQS	`ApproximateAgeOfOldestMessage`	Approximate age of the oldest message on the queue is getting higher
Amazon SQS	`NumberOfMessagesSent` (on the dead-letter queue)	Rate of messages hitting the dead-letter queue is increasing

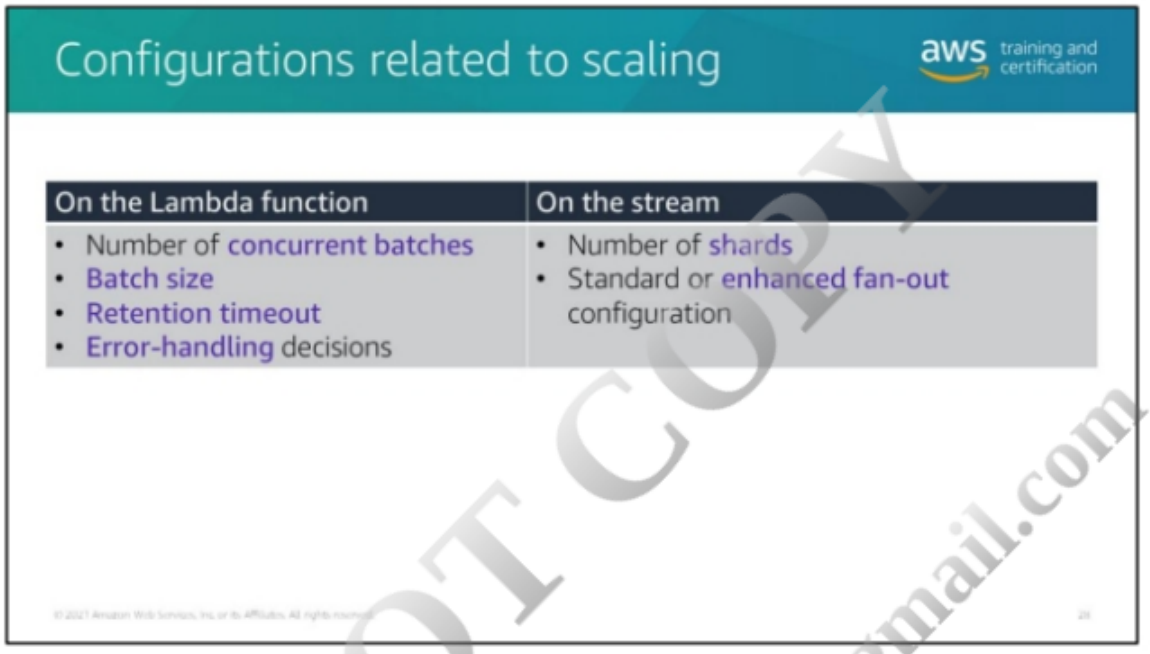


Let's review how Lambda consumes stream records when you use a stream to invoke your function.

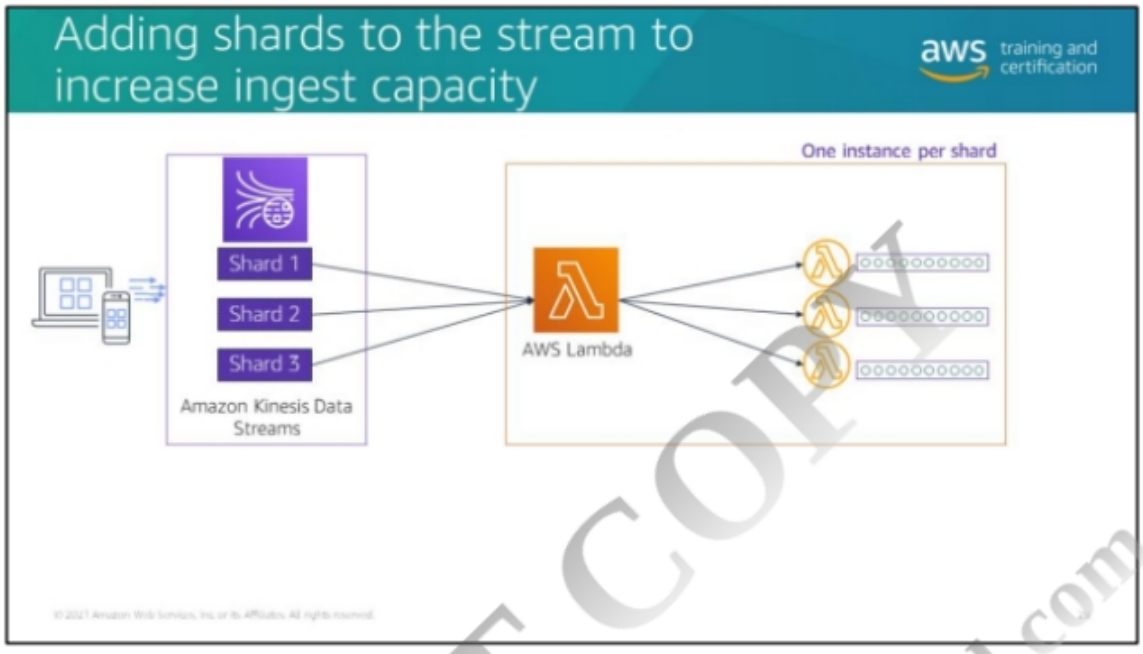
Producers write data to a Kinesis data stream. Lambda polls the stream and gets batches of records to process. In contrast to queues, Lambda (and other consumers) don't delete records from the stream. Records remain on the stream for a period of time and are available to multiple consumers. Lambda must maintain a pointer on the stream and move the pointer after a batch of records has been successfully processed.

If you don't configure any additional error handling, Lambda keeps trying a failed batch until the records expire off of the stream.

By default, Lambda uses only one invocation per shard in your stream. From a scaling perspective, this gives the Lambda function the maximum available throughput for consumers of the stream (assuming the Lambda function in the example is the only consumer of the stream).



You are using Kinesis Data Streams in the examples of these features, but the same considerations apply when Amazon DynamoDB is the source of the stream. As noted earlier in the course, the exception is that you don't set the number of shards on the DynamoDB table; the table configuration indirectly determines the number of shards.

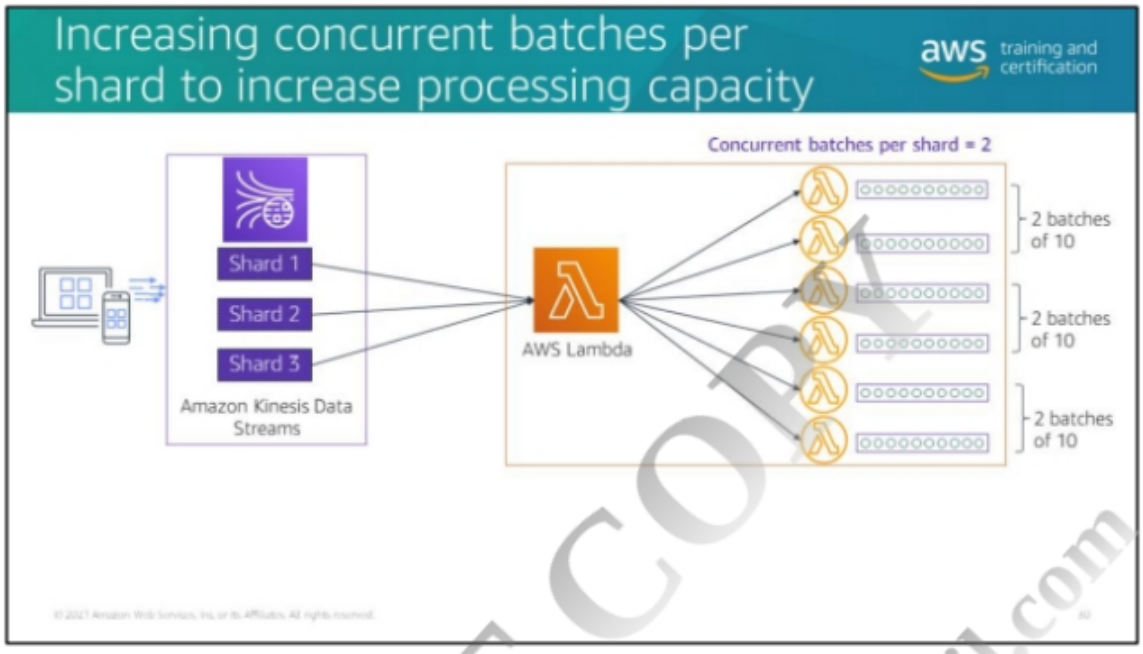


If you need to increase the speed at which your stream can ingest data being written by producers that feed the stream, you need to increase the number of shards in the stream.

A *shard* is a uniquely identified sequence of data records in a stream. A partition key is used to group data by shard within a stream.

The data capacity of your stream is a function of the number of shards that you specify for the stream. The total capacity of the stream is the sum of the capacities of its shards. In this example, the stream invoking this Lambda function has three shards, so three invocations of your Lambda function are invoked, one per shard.

For additional details about sharding, visit https://docs.aws.amazon.com/streams/latest/dev/key-concepts.html.

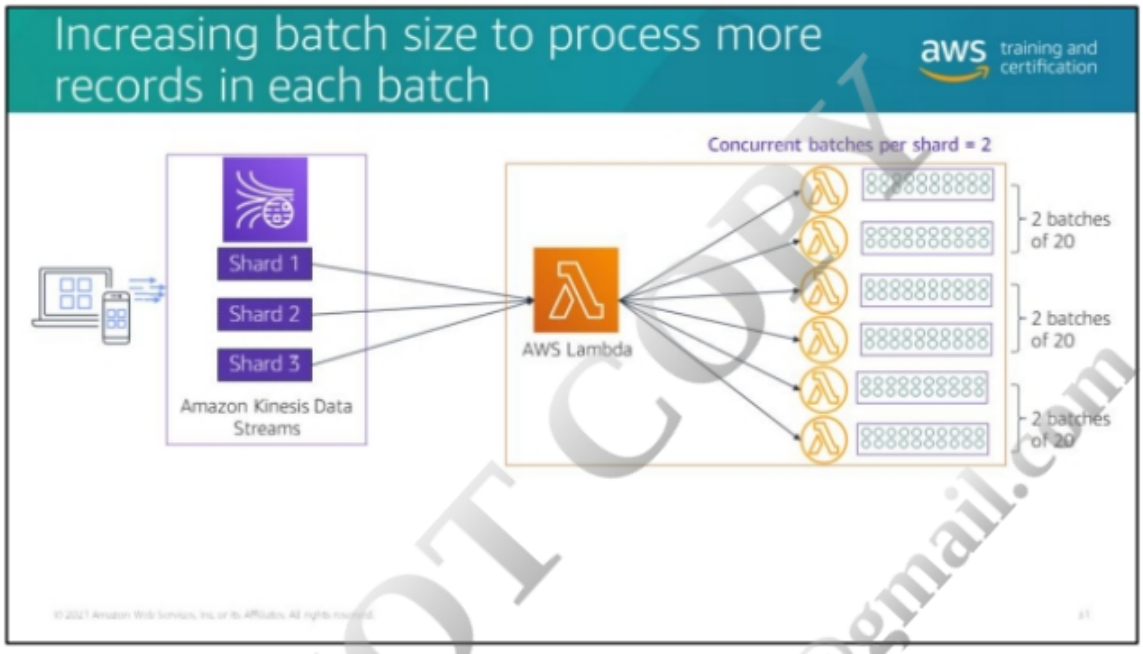


When you are using your stream to invoke Lambda, you have a few options within the configuration that impact scaling.

The first is increasing concurrent batches per shard. If the nature of your workload is such that you want to increase the number of function invocations running in parallel per shard, you can increase the concurrent batches per shard from the default of one and run more invocations in parallel.

In this example, with concurrent batches per shard set to two, Lambda pulls two batches off of each shard in parallel and processes them at the same time.

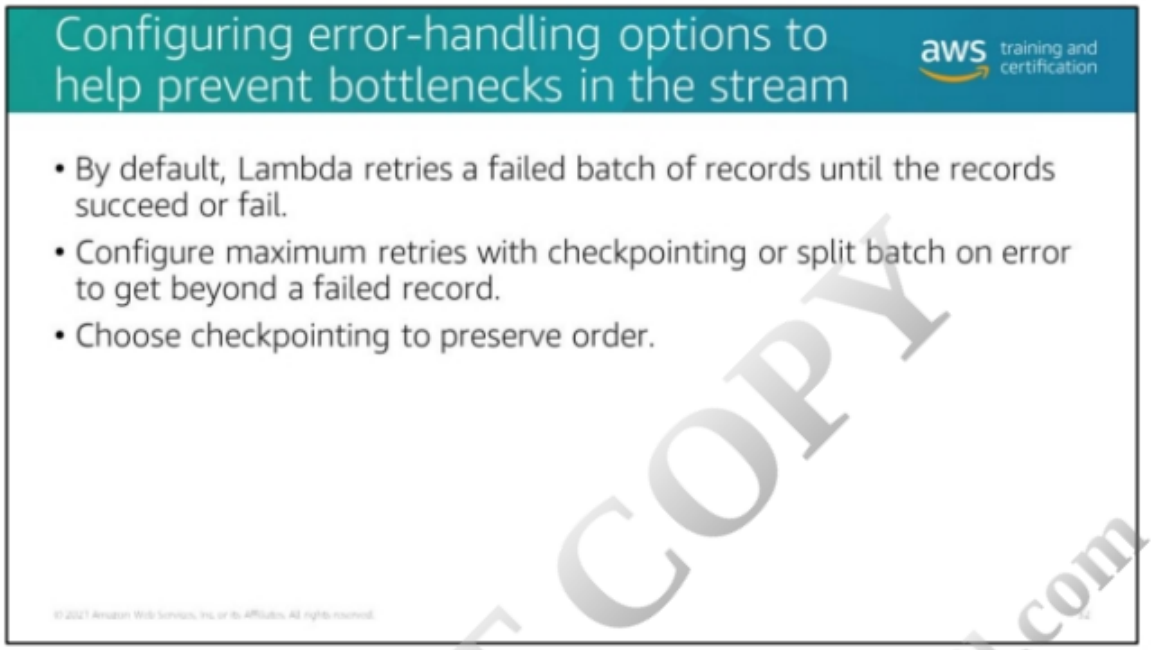
You can access this option via API using ParallelizationFactor. For more information, see https://docs.aws.amazon.com/lambda/latest/dg/API_EventSourceMappingConfiguration.html.



Depending on your workload and the business logic that your function is performing, you might try different batch sizes to find the optimal size for your workload. Analyze the nature of the processing you need to do at the record level to help you determine the optimum batch size.

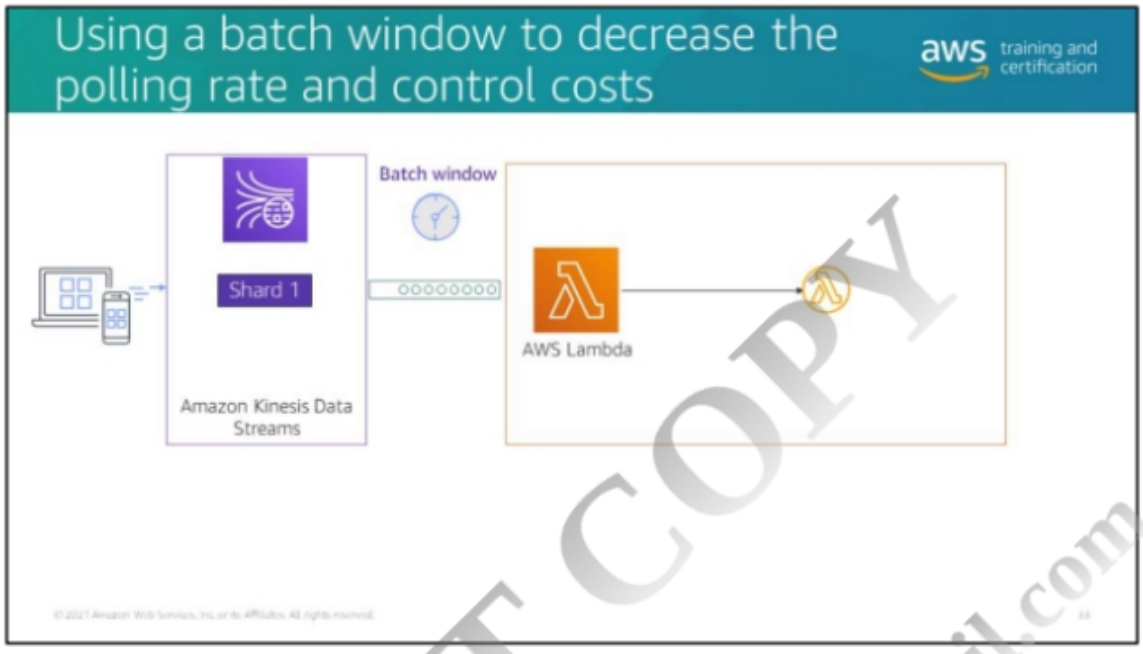
If you pull more records each time Lambda polls the stream, you can decrease costs and might improve performance. In the example, your Lambda function went from processing 10 records at a time per shard (one concurrent invocation per shard) to 40 records at a time (two concurrent functions, each processing a batch of 40). A higher batch size make sense when your function is doing quick processing on the records in the batch.

However, a larger batch size means more records to process in each batch, and if there's an error in the batch, the shard will be blocked until your error-handling conditions are met, and it may take a bit longer to isolate the error with a larger batch. Longer-running functions probably need smaller batch sizes. If it takes too long to process a batch of records, the records might hit the retention timeout, and you won't have the opportunity to try to process them. You can increase the retention timeout, but there is a cost to doing so.



The error-handling options you reviewed previously can help to keep the records flowing and keep Lambda moving through the stream to avoid bottlenecks.

You need to do some analysis on your own stream and connected Lambda functions to get a feel for how many retries you want and what batch size works best. For example, maybe you know that a failed record almost always points to an issue that a separate process needs to manage, so you might set retries low to prevent spending time retrying records.



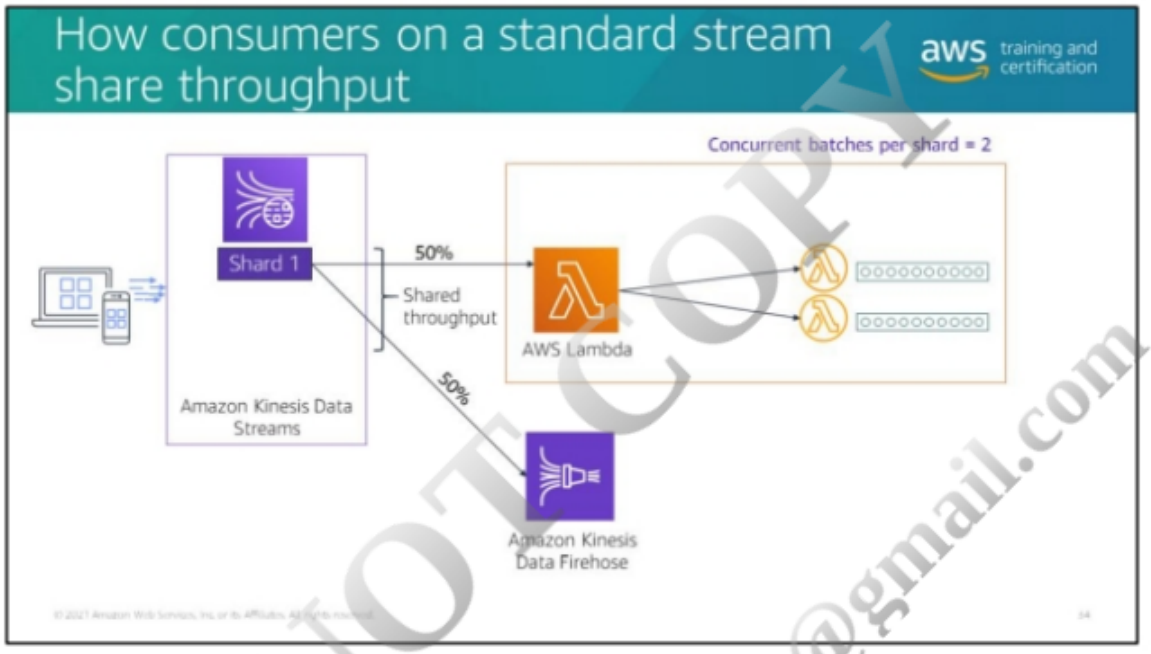
Looking from a different scale perspective for a minute: What if you want to slow things down a little to let the stream gather more records before invoking Lambda?

Lambda reads records from a stream at a fixed cadence, and by default Lambda invokes your function as soon as records are available in the stream. If the batch that Lambda reads from the stream has only one record in it, Lambda sends only one record to the function. Although the per-invocation cost is small, if you do this thousands of times, you incur costs for processing small numbers of records versus the batch size that you want.

To avoid invoking the function with a small number of records, you can tell the stream to buffer records for up to 5 minutes by configuring a *batch window*. Before invoking the function, Lambda continues to read records from the stream until it has gathered a full batch or until the batch window expires. This option lets you increase the average number of records passed to the function with each invocation.

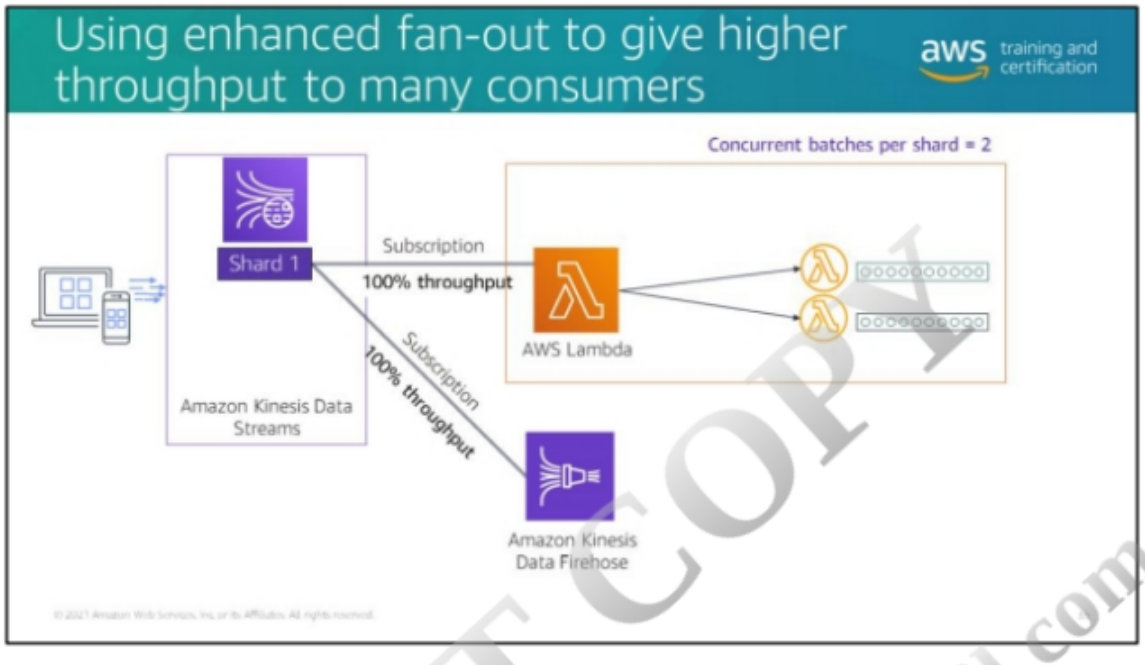
In this example, the batch size is 10, and a batch window lets Lambda collect almost a full batch (eight in this example) before the window expires and Lambda sends the batch of eight to an invocation.

This option is helpful when you want to **reduce the number of invocations** and optimize cost.



Let's go back to general characteristics of the stream itself, unrelated to when Lambda is a consumer.

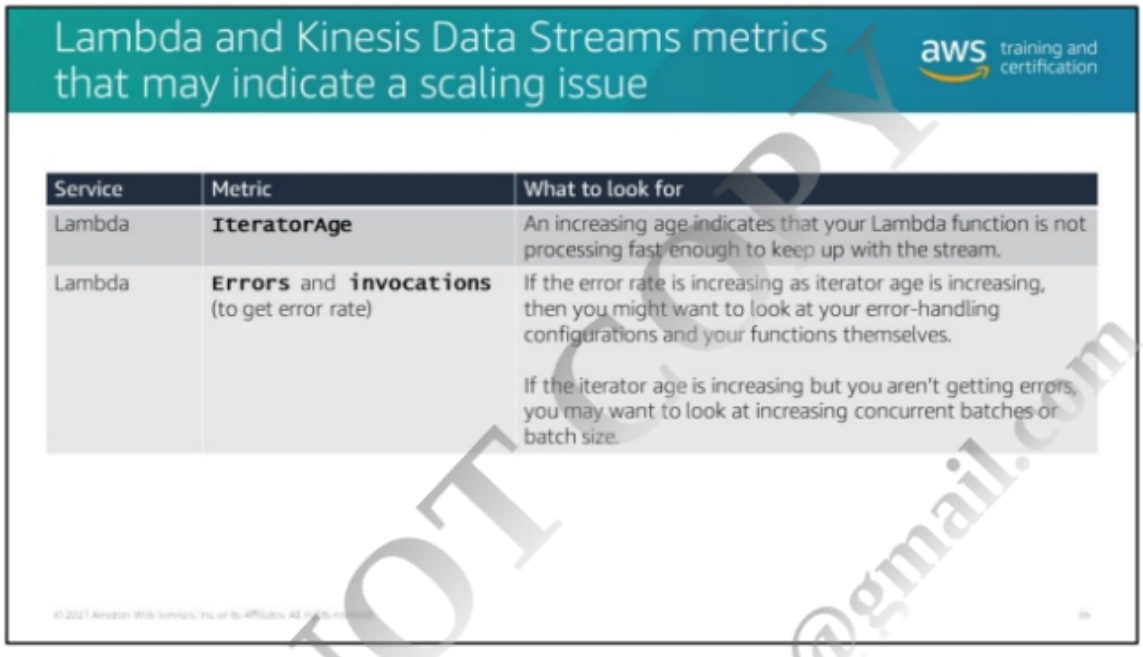
With standard streams, consumers of the stream share the read throughput of the stream. So if you added a second function polling the stream or added a different type of consumer (for example, Amazon Kinesis Data Firehose), all of the consumers share the read throughput of the stream for each shard. In this example, both the Lambda function and Kinesis Data Firehose would get 50 percent of the capacity of the stream. The illustration shows only one shard, but this would be true across all shards on the stream.

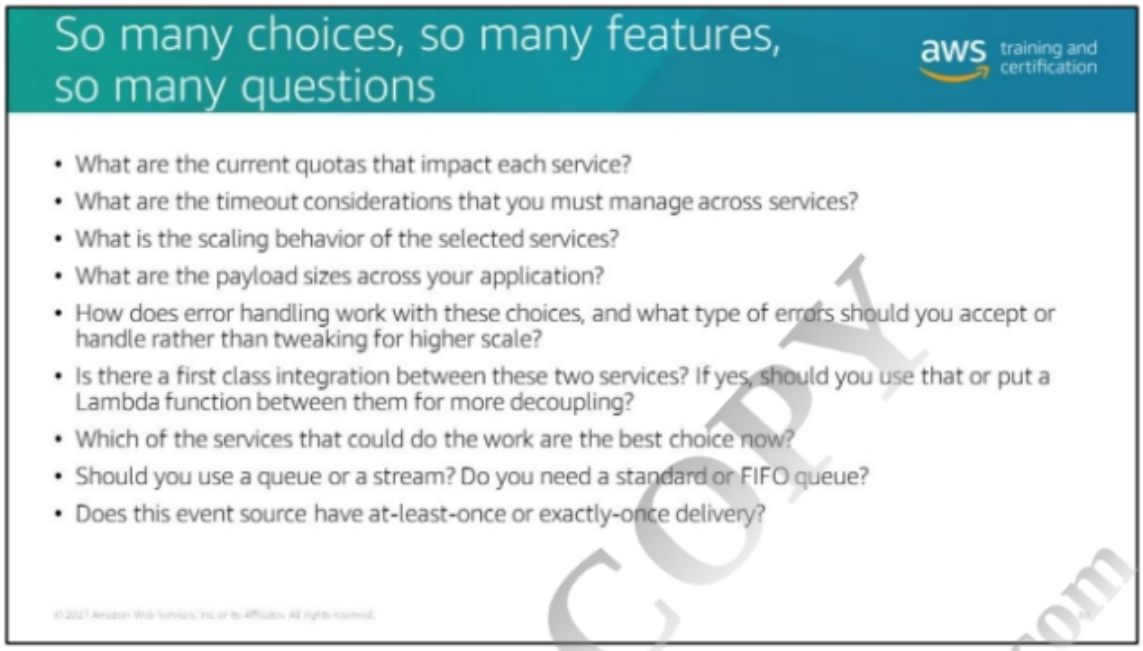


Enhanced fan-out might make sense if your workload requires a high number of consumers with very high throughput. Enhanced fan-out consumers subscribe to the stream. Once they are subscribed, data from the shard is pushed out to the consumer using an HTTP/2 request that can run for up to 5 minutes. Data continues to get pushed out to the consumer as data comes in.

This reduces the latency and increases throughput. Any consumers that are using enhanced fan-out get their own pipe, so they get 100 percent of the throughput reading from the stream.

Using enhanced fan-out incurs additional cost, so consider what your traffic will look like and whether the latency of a standard consumer is acceptable. Generally speaking, if you have three consumers or fewer and latency isn't critical, you probably want to use a standard stream to minimize the cost.





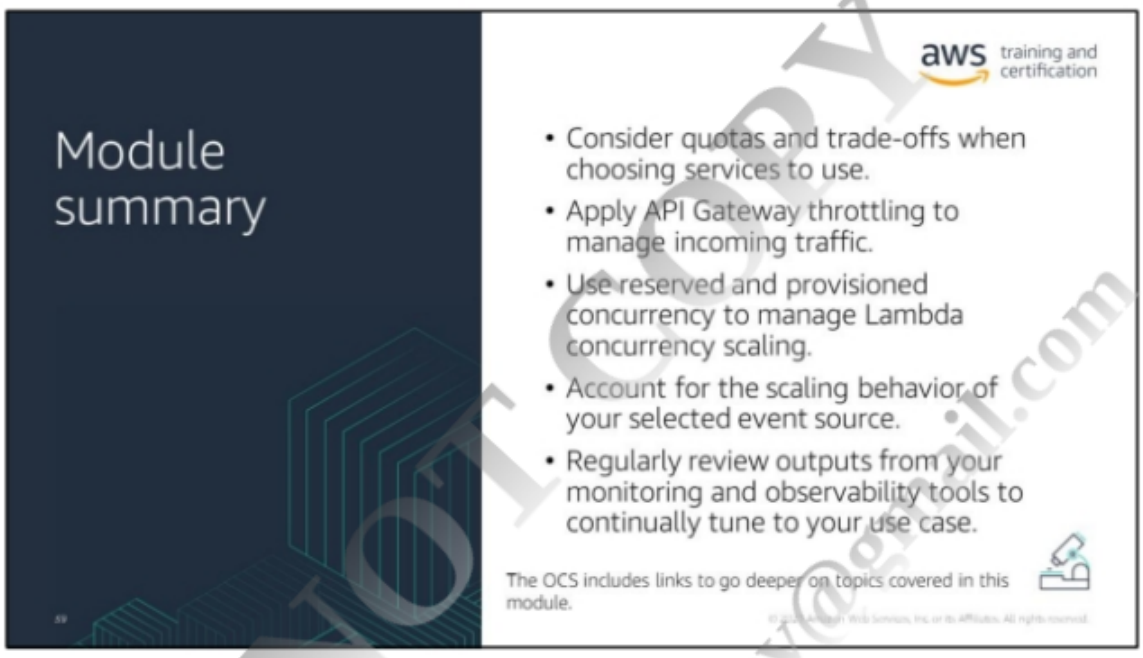
The point of this slide is not to overwhelm you, capture all of the scaling options that you might consider in your application, or provide a finite list of things to consider; rather, this slide reinforces what you have been learning about throughout this section.

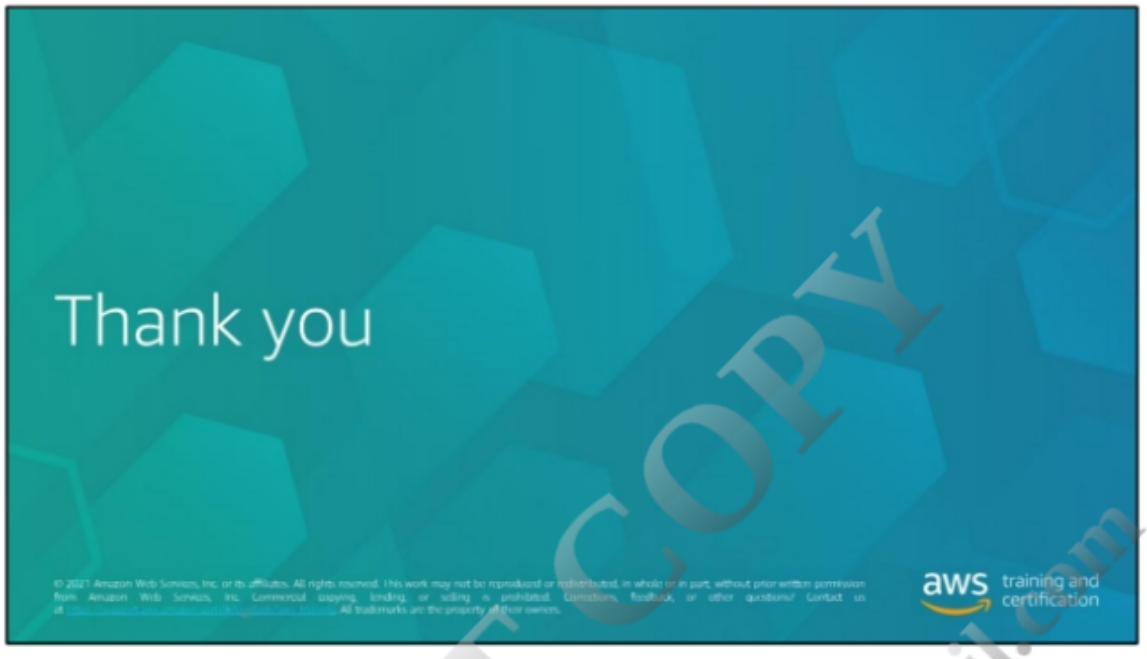
Use AWS managed services to decrease the undifferentiated lifting that you perform as a developer, but understand each of the building blocks and make sure you test each integration point with data patterns that are similar to production.

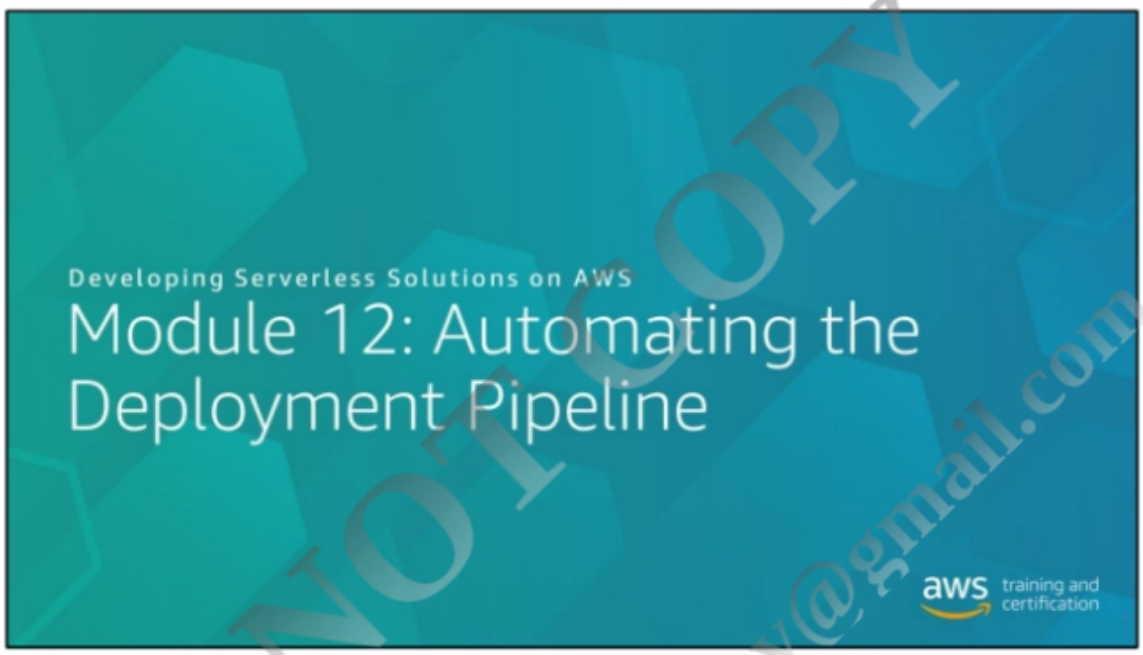
Use monitoring and observability tools to build ways to understand the patterns of how consumers of your application actually use it. Use X-Ray to identify slow points in a request's end-to-end journey through your system.

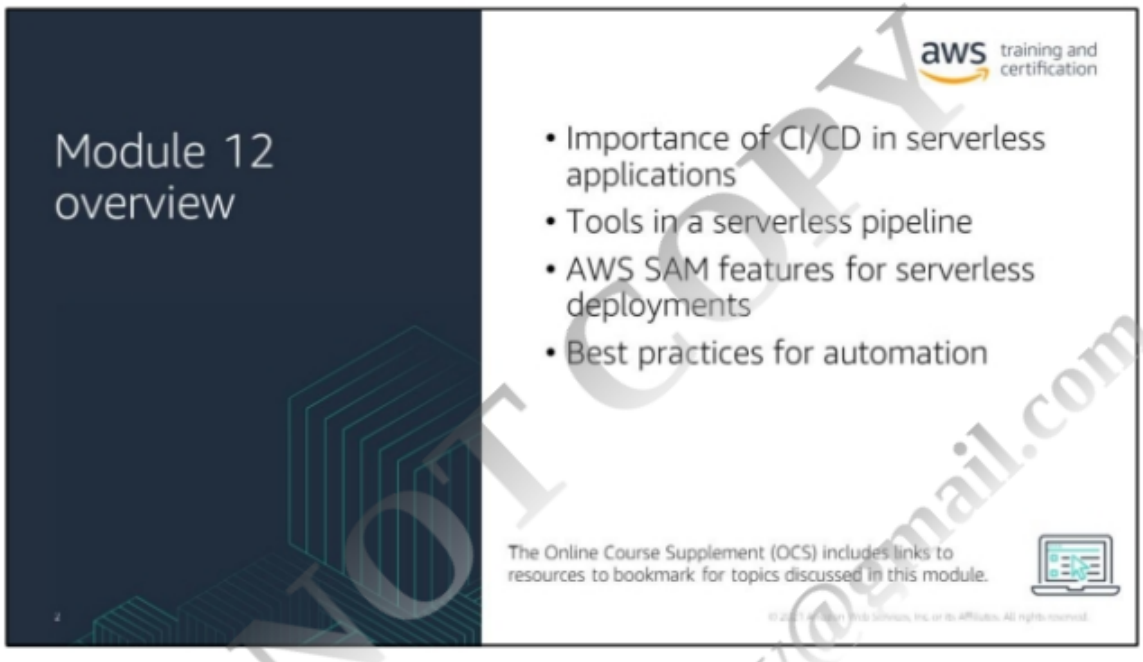
Going back to what you learned on day one about being able to scale individual elements, review your production application and think about when it might be time to replace a component because of new features or services, which might be able to do that job more efficiently or at a lower cost.

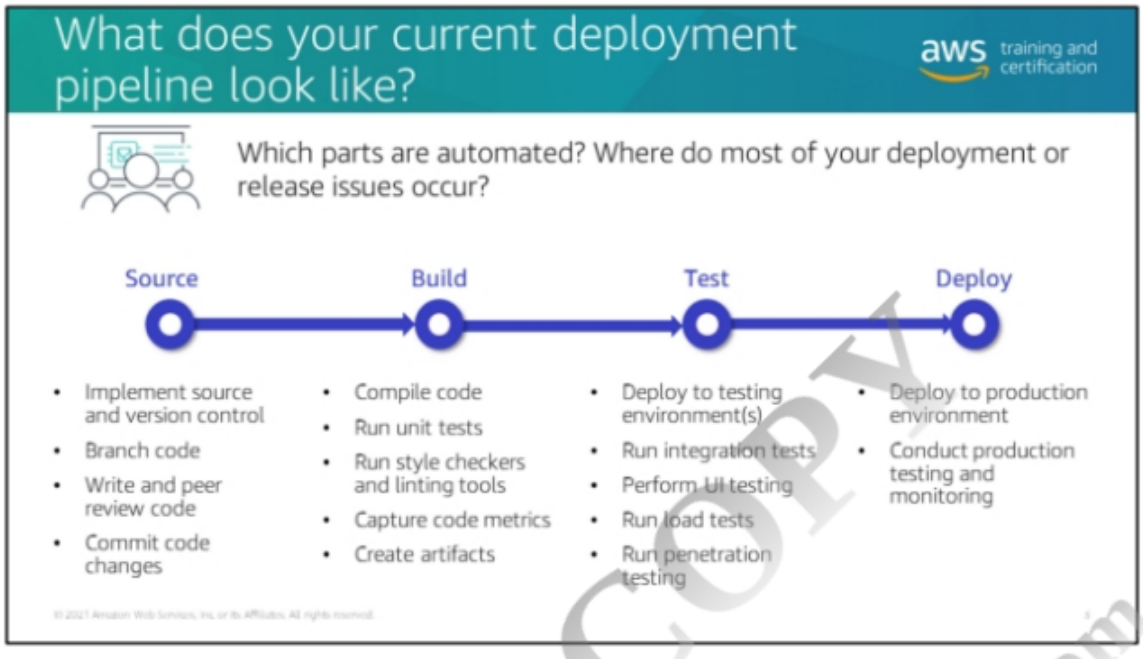
The key takeaway is one that this section opened with: Scaling is a continual process, and you want to observe and tweak on a regular basis.

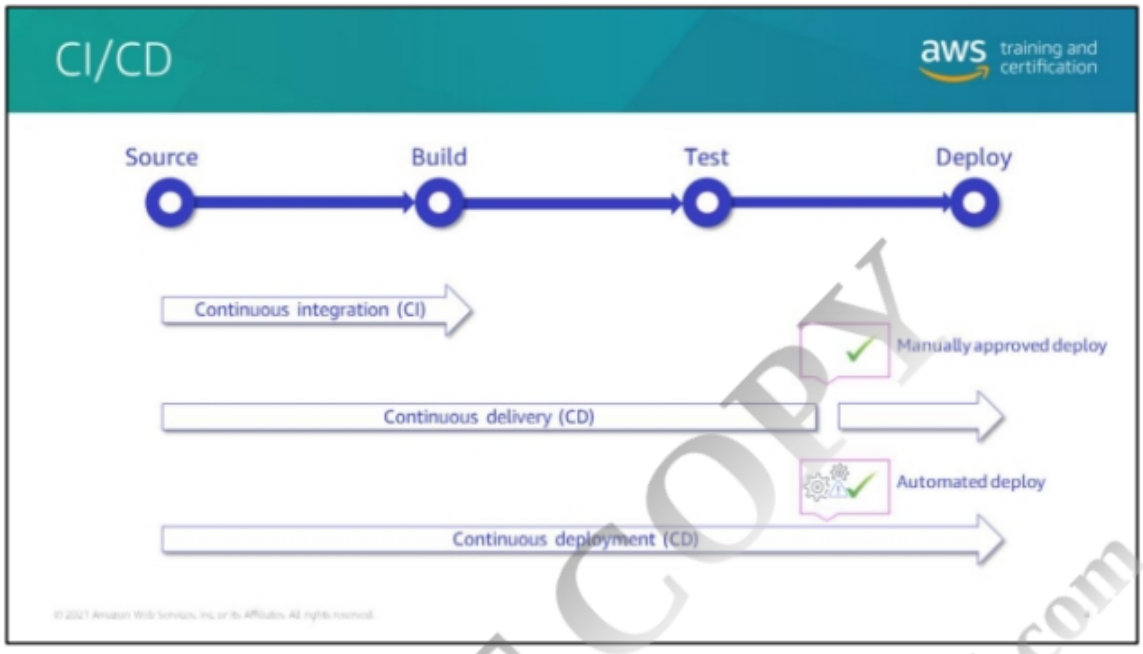












Automating the pipeline and implementing continuous integration/continuous delivery (CI/CD) has a couple of key sections:

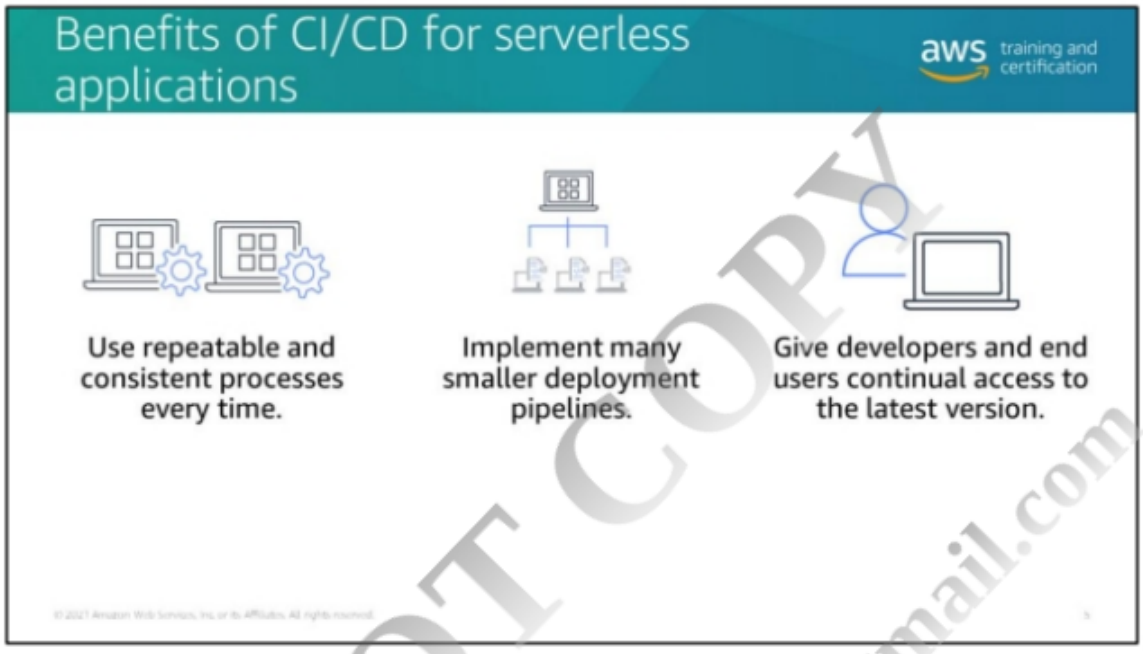
Continuous integration (CI) is the practice of automatically incorporating new code commits into a build and running unit tests and other automated quality gate checks before creating a new artifact for deployment.

Continuous delivery (CD) takes this further, automatically taking the build artifact, deploying it to test or staging environments, and running automated tests on the deployed build.

After the deployment and testing of the artifacts in your test and staging environments and before deployment to production, an individual or group provides manual approval. The group responsible for the production deployments deploys the tested artifacts into the production environments, where any post-deployment tests are performed.

With **continuous deployment (CD)**, your pipeline also automates the step of deploying to production. Your pipeline gets an acknowledgement from the testing step that the build artifact is ready for production, and the process of deploying and validating in production completes automatically. That doesn't mean that approval isn't required in the automated pipeline, but it does mean that the request for approval is built into the pipeline. Upon receiving approval, the deployment commences automatically and includes automated checks to verify the deployment and roll it back if an issue occurs.

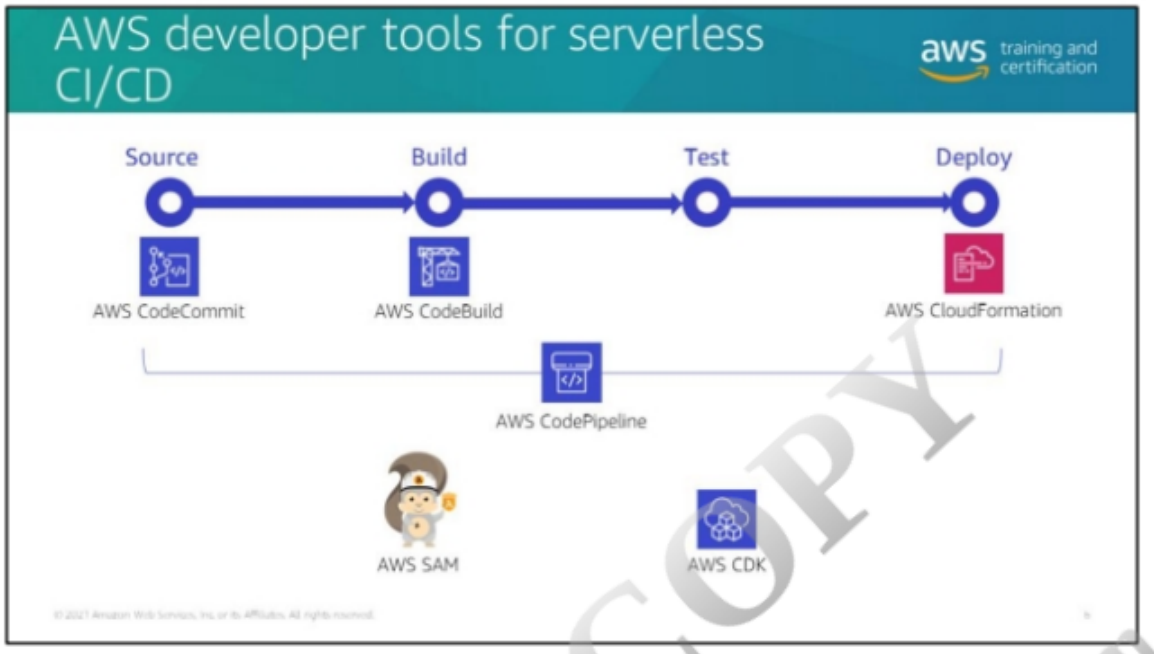
Whether you are doing only continuous delivery or both continuous delivery and continuous deployment as part of your pipeline, you may find it referred to as CI/CD.



The first benefit of an automated process is evident if you reflect on what has gone wrong with your deployments. With a repeatable process, you get consistent results and greatly reduce the chance of manual misses by someone on the team. You can continue to improve upon your processes to increase velocity and the levels of automation as your processes mature. When the process of building and deploying new releases isn't a resource-intensive process, it's much easier to release frequent, incremental updates.

Automation is especially important with serverless applications. Having many distributed services that can be deployed independently means more and smaller deployment pipelines that each build and test a service or set of services. With an automated pipeline, you can incorporate better detection of anomalies and more testing, halt your pipeline at a certain step, and automatically roll back a change if a deployment were to fail or if an alarm threshold were met.

An added benefit is the speed with which you can move new code and new features from source to test. Now you can build and deploy in very small chunks, which also gives end users the ability to review smaller updates on a regular basis in the test environment. This makes it easier to correct a change or unexpected issue with the requirements and increases visibility into work that has been completed.



Your pipeline may be a mix of AWS or third-party components that suit your needs, but the concepts apply generally to whatever tools your organization uses for each of the steps in the deployment toolchain.

This module references the AWS tools that you can use in each step in your CI/CD pipeline. More specifically to serverless, you will also look at features within the AWS Serverless Application Model (AWS SAM) templates that support your automated pipeline and the creation of AWS CloudFormation stacks for each of your environments.

As the diagram notes, you might also use the AWS Cloud Development Kit (AWS CDK) to define your application resources.

CloudFormation transforms and provisions both AWS SAM and AWS CDK definitions into your desired stack.

You might choose another serverless framework, but the goal is to take advantage of features that simplify the repeatability of building and deploying your serverless applications into multiple environments and accounts.

AWS CodeCommit is a fully managed source-control service that hosts secure Git-based repositories

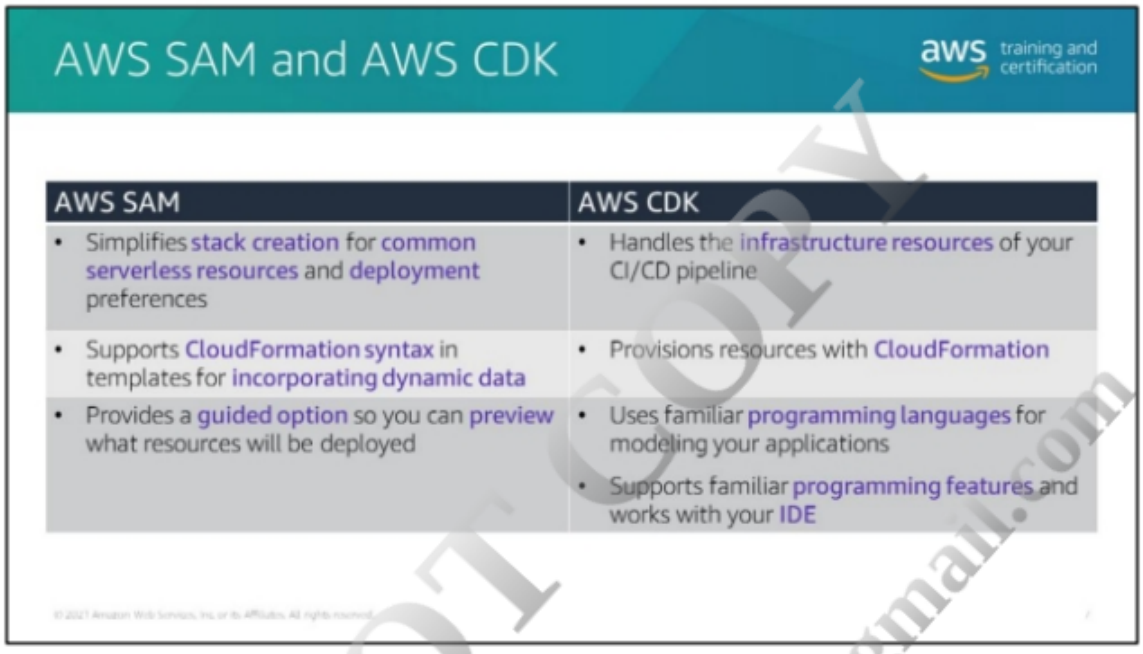
- Allows teams to collaborate on code in a secure and highly scalable way
- Automatically encrypts your files in transit and at rest
- Is integrated with AWS Identity and Access Management (IAM)

AWS CodeBuild is a fully managed build service that can compile source code, run tests, and produce software packages

- Scales continuously and processes multiple builds concurrently
- Can consume environment variables from AWS Systems Manager Parameter Store
- Supports dependency caching

AWS CodePipeline provides continuous delivery for fast and reliable application updates

- Lets you model and visualize your software release process
- Builds, tests, and deploys your code each time there is a code change
- Integrates with third-party tools and AWS



This course covered AWS SAM and AWS SAM templates earlier, but a few things are particularly relevant for deploying your serverless applications.

AWS SAM simplifies the CloudFormation code that you would need to write for deploying common serverless resources. AWS SAM can also include deployment preferences that let you include pre- and post-deployment hooks and specify your choice of safe deployment type (linear or canary).

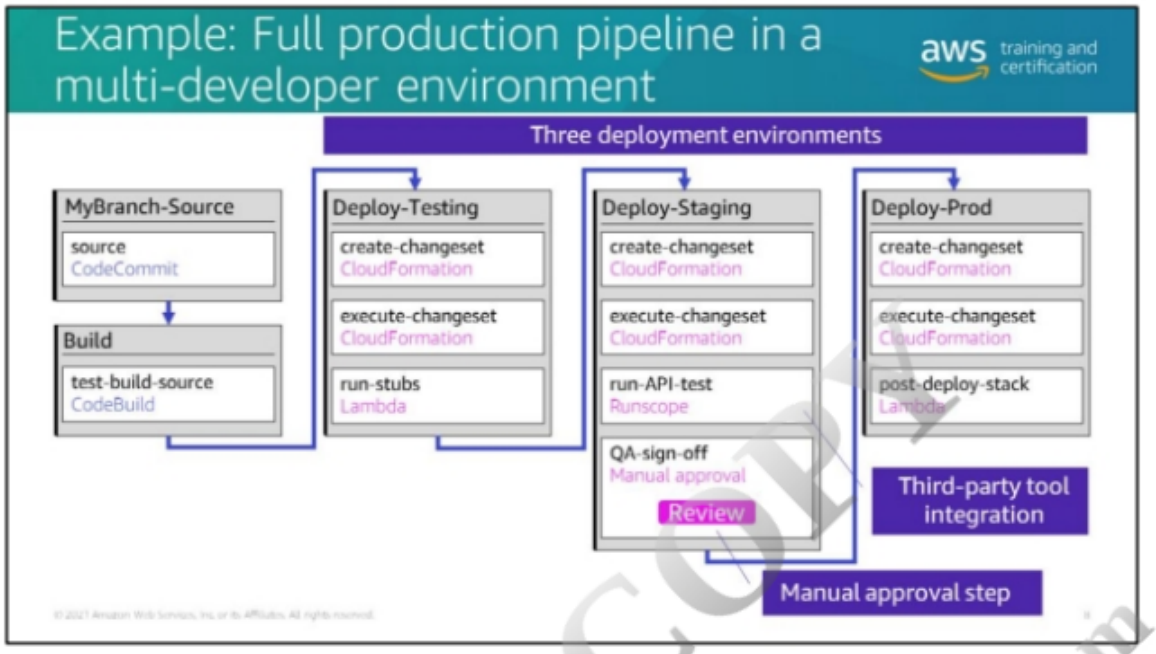
AWS SAM templates support CloudFormation syntax that supports the use of dynamic data in a template so that you can use one template across environments. Examples include string manipulation, parameters, reference substitutions, and mappings.

The guided option in AWS SAM is a great way to preview the resources that will be deployed before you actually deploy the stack. Use this option to understand what you're building and verify that you're getting what you intended.

You can use the AWS SAM framework to define your application stack and deployment preferences and then use AWS CDK to provision any infrastructure-related resources, such as the CI/CD pipeline.

This lets you use AWS SAM to simplify the creation of your serverless resources and use a familiar programming language to build your pipeline.

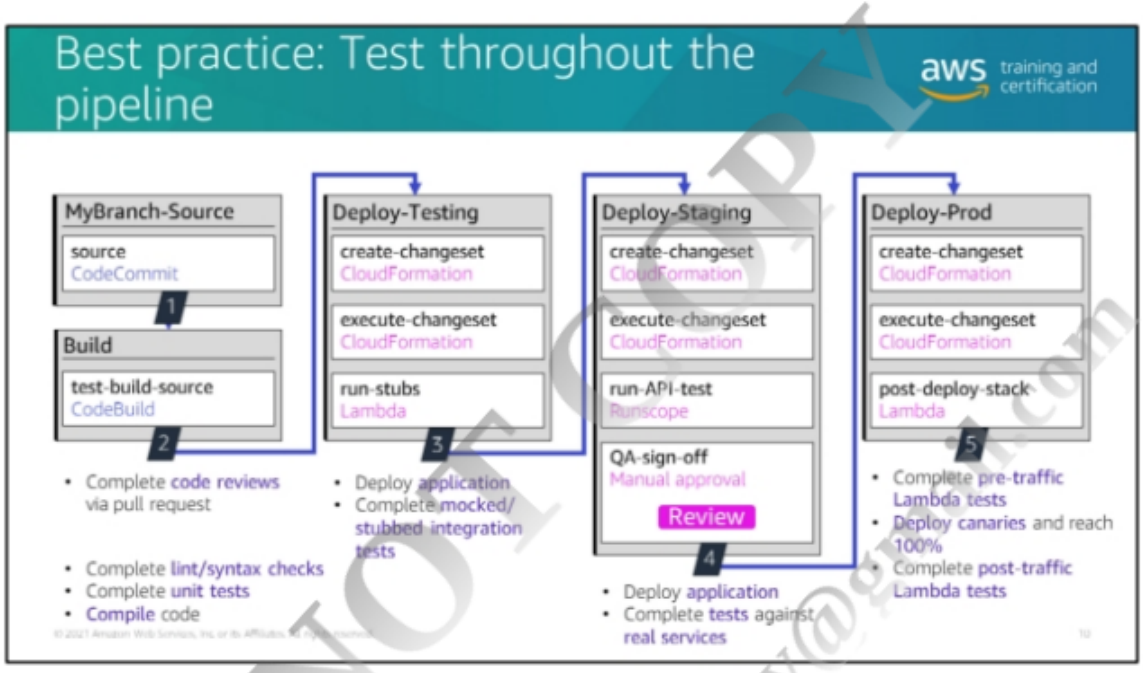
You will use AWS SAM and AWS CDK together to deploy your applications in lab 6.



In this example, after the build step is successful, the build is deployed to three consecutive stacks, which function how different environments would in a server-based pipeline.

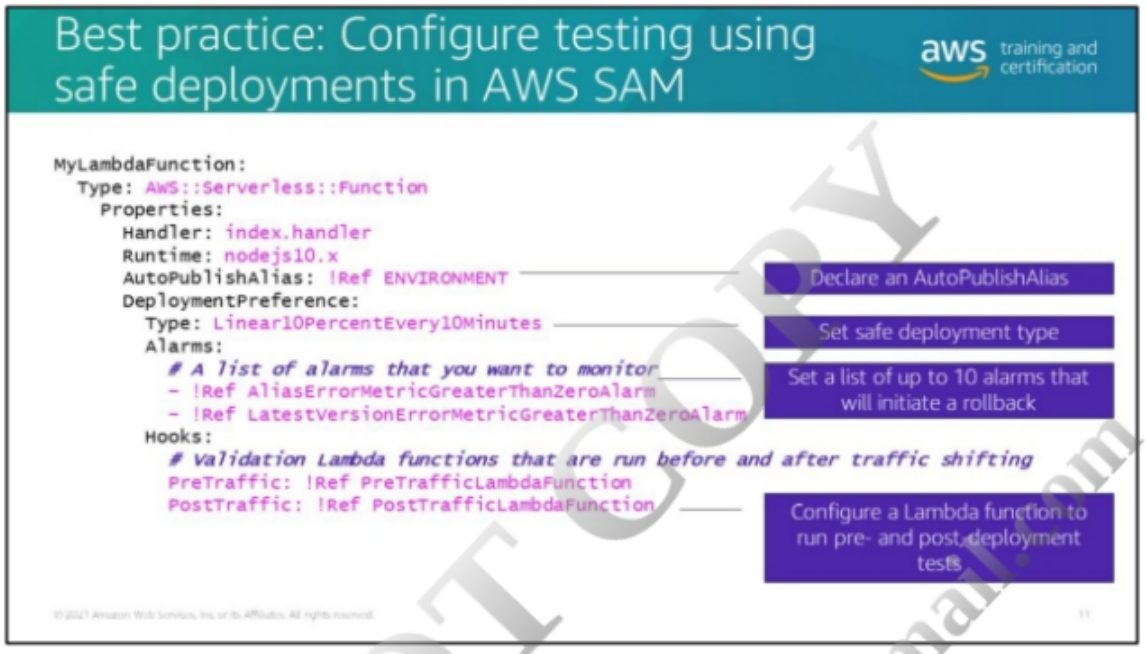
You include automated tests that are appropriate to each step in the deployment before deploying to the next environment.

In this example, the staging environment includes a manual sign-off, which must be complete before automation continues to deploy the code to production.



Using an automated pipeline and a serverless framework like AWS SAM, it's possible to test across the pipeline and automatically stop or roll back when tests aren't successful.

Your AWS SAM template can include AWS CodeDeploy preferences such as AWS Lambda functions that will be used for testing, and a linear or canary deployment option to perform blue/green testing as part of a deployment.



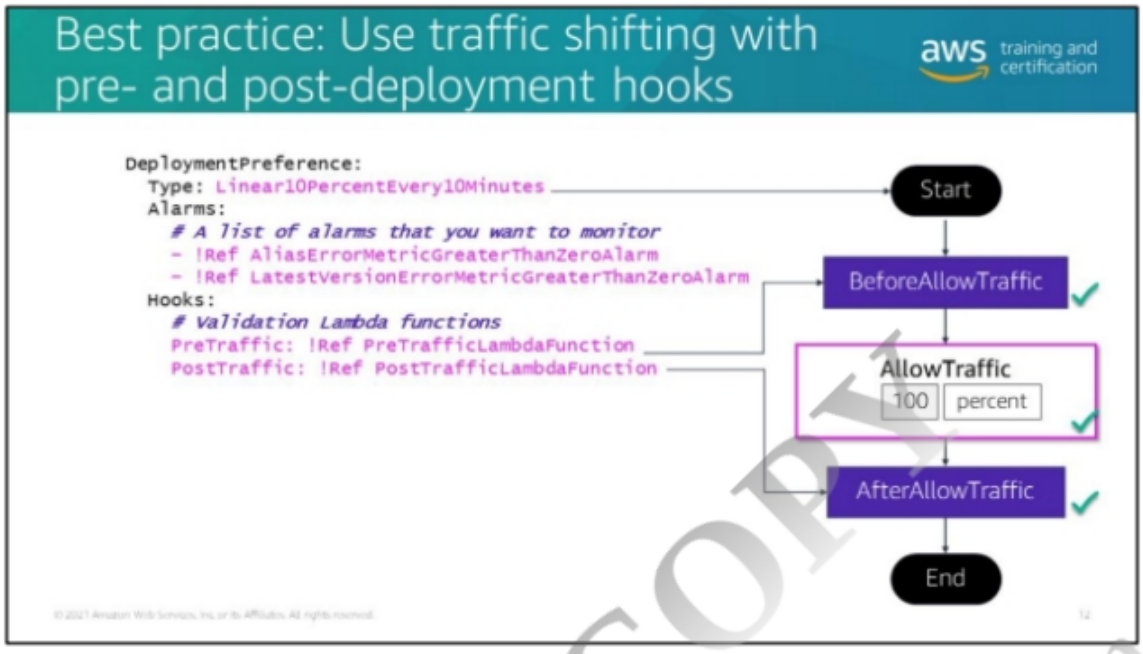
By adding the AutoPublishAlias property and specifying an alias name, AWS SAM will do the following:

- Detect when new code is being deployed based on changes to the Lambda function's Amazon Simple Storage Service (Amazon S3) URI
- Create and publish an updated version of that function with the latest code
- Create an alias with a name you provide (unless an alias already exists) and point to the updated version of the Lambda function

The deployment preference sets the type of traffic shifting that you want to use between the previous and latest versions of your code.

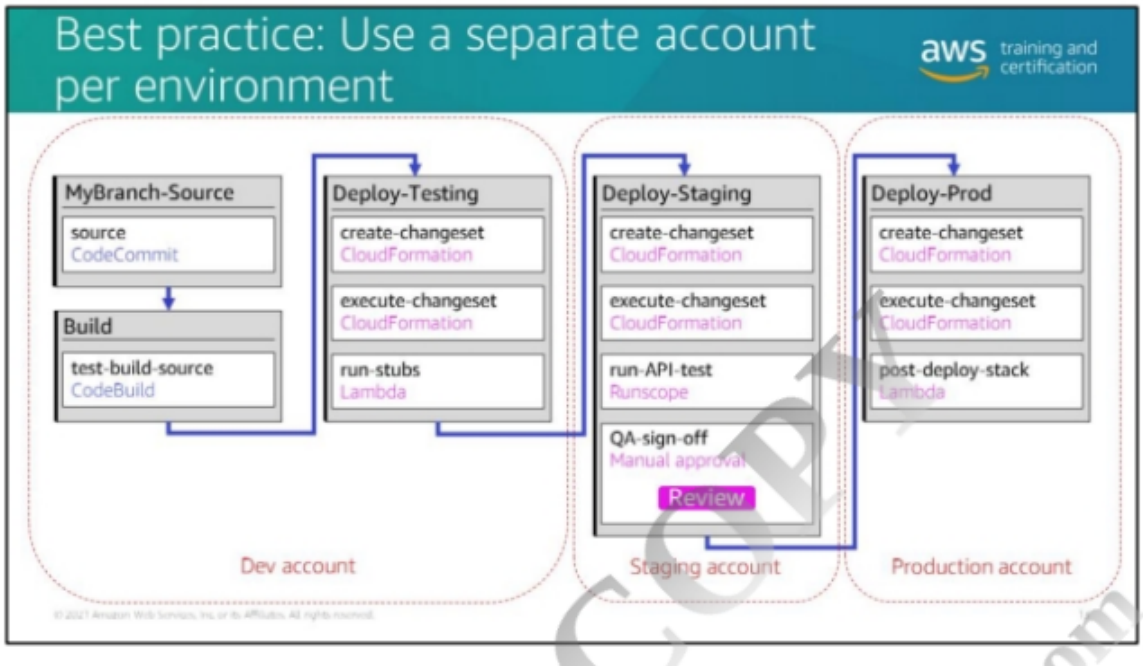
This example is using a linear deployment in which 10 percent of the traffic moves to the new version every 10 minutes.

For more information about options for deploying serverless applications, visit https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/automating-updates-to-serverless-apps.html.



Let's look a bit closer at how the deployment preferences work.

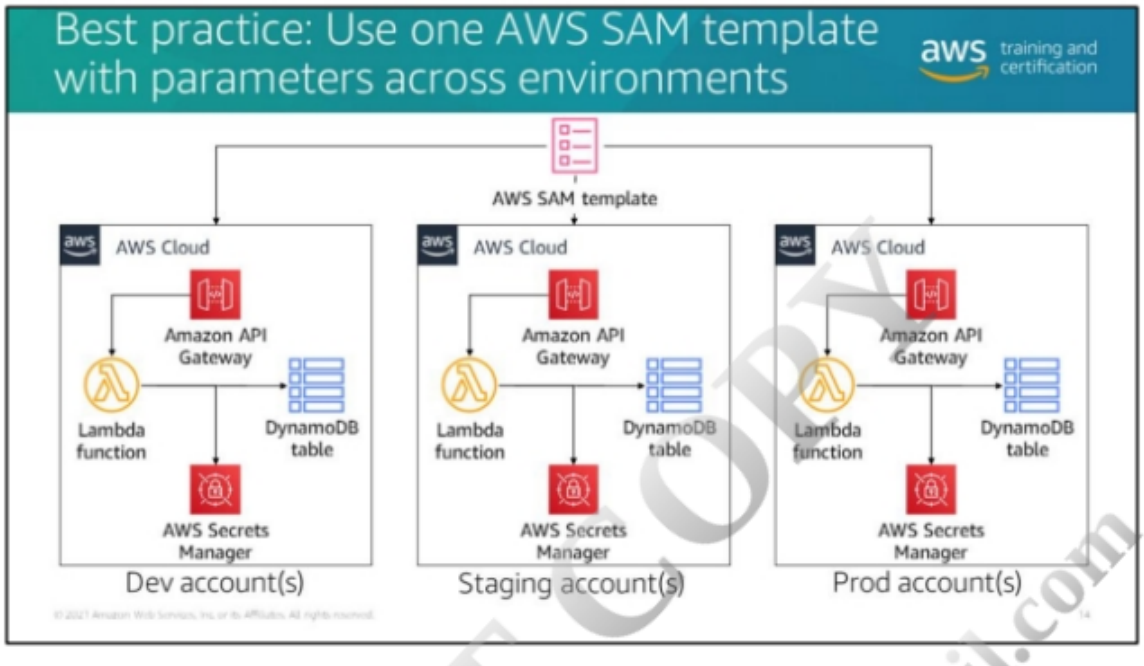
1. When the application is deployed, the PreTraffic Lambda function runs to determine if the deployment should continue. If that function completes successfully (that is, returns a 200 status code), the deployment continues. If the function does not complete successfully, the deployment rolls back.
2. Assuming a successful response from the PreTraffic function, the linear traffic that shifts 10 percent of traffic every 10 minutes to the designated Lambda alias begins. If any of the alarms are met during this progression, the deployment rolls back.
3. If the traffic successfully completes the progression to 100 percent of traffic to the new alias, the PostTraffic Lambda function runs. If the function returns a 200 status code, the deployment is complete. If the PostTraffic function is not successful, the deployment rolls back.



A best practice with serverless is to use separate accounts for each stage or environment in your deployment. Each developer has an account, and the staging and deployment environments are each in their own accounts.

This approach limits the blast radius of issues that occur (for example, unexpectedly high concurrency) and allows you to secure each account with IAM credentials more effectively with less complexity in your IAM policies within a given account. This practice also makes it less complex to differentiate which resources are associated with each environment.

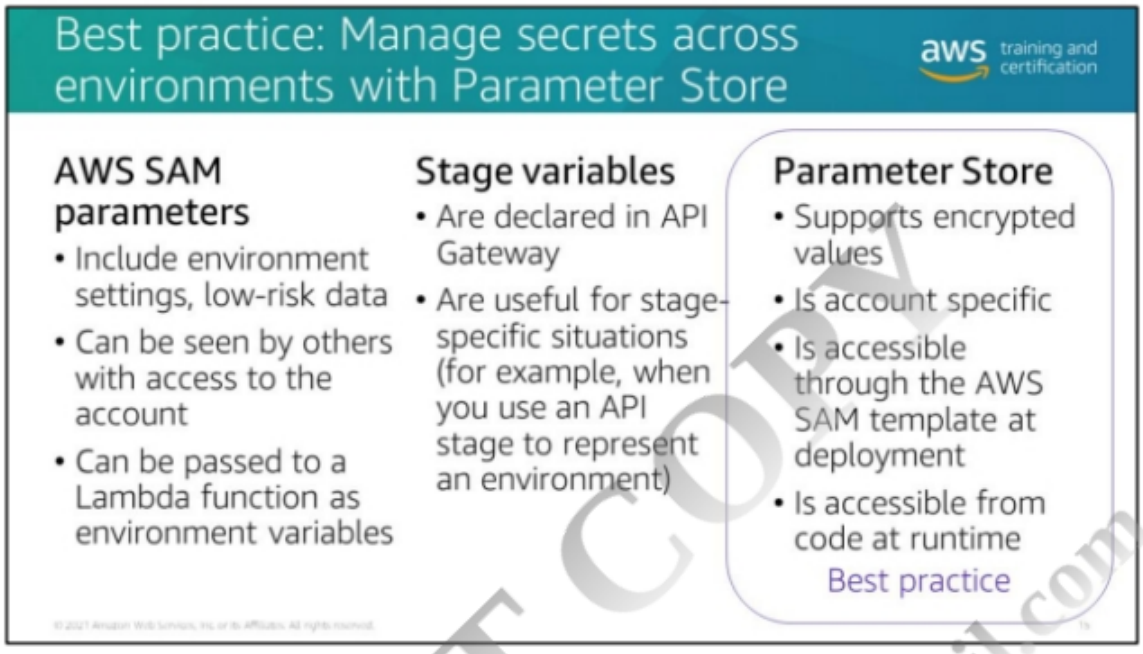
Because of the way costs are calculated with serverless, spinning up additional environments doesn't add much to your cost. Other than where you are provisioning concurrency or database capacity, the cost of running tests in three environments is not different than running them in one environment because the cost is mostly about the total number of transactions that occur, not about having three sets of infrastructure.

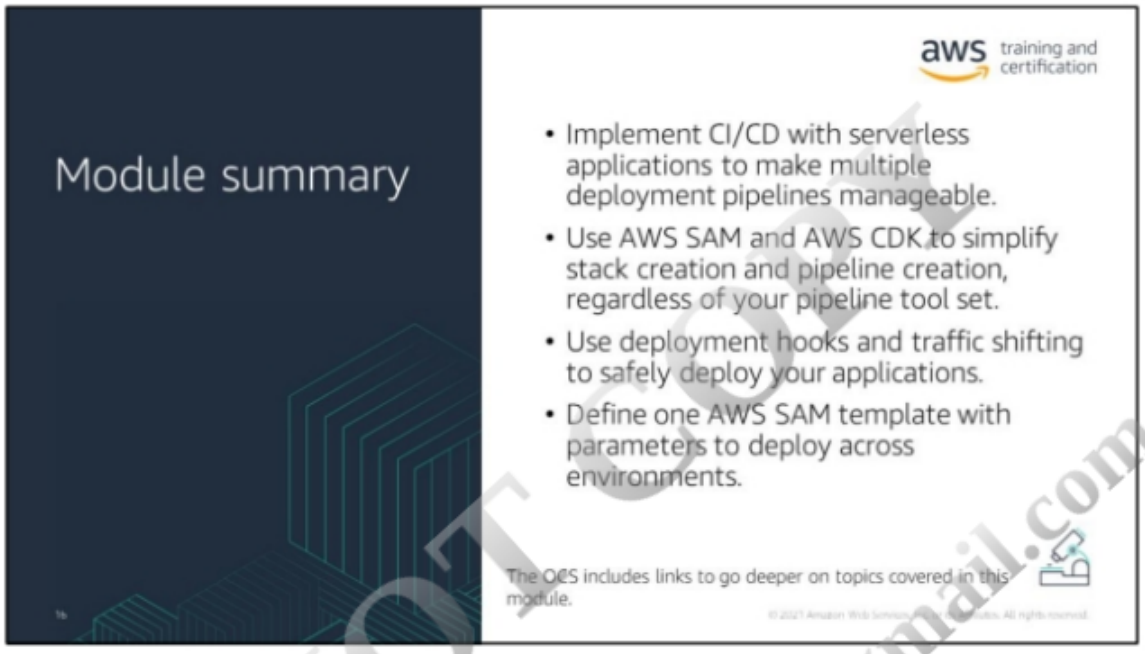


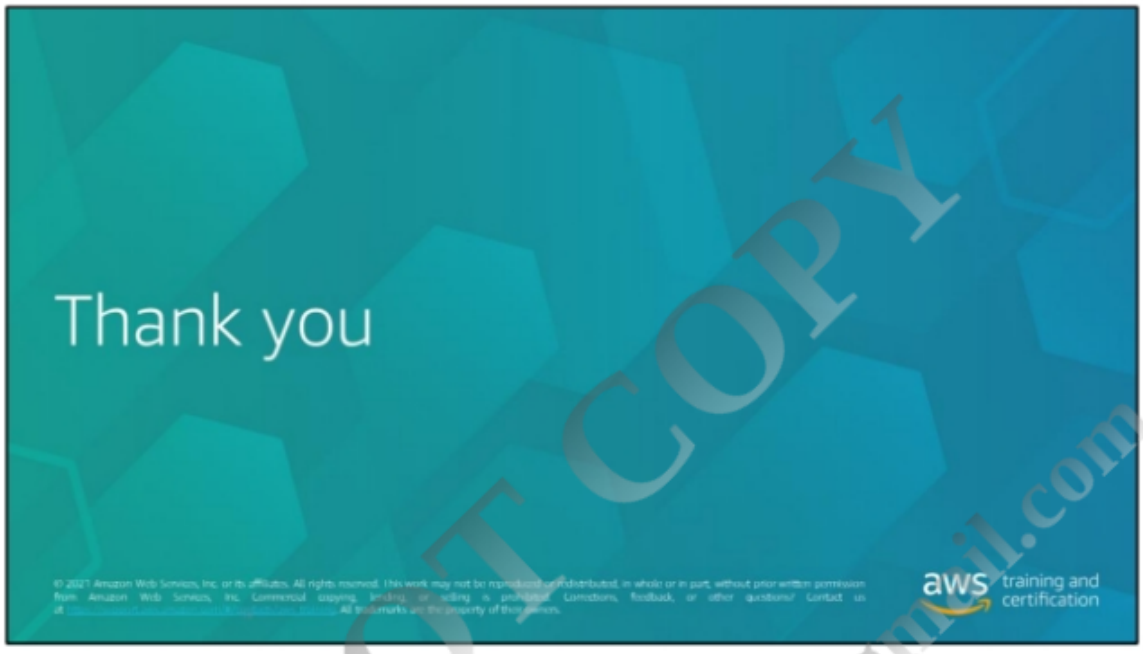
As noted earlier, AWS SAM supports CloudFormation syntax so that your AWS SAM template can be the same for each deployment environment with dynamic data for the environment provided when the stack is created or updated. This helps you make sure that you have parity between all testing environments and aren't surprised by configurations or resources that are different or missing from one environment to the next.

With AWS SAM, you can build out multiple environments using the same template, even across accounts:

- Use parameters and mappings when possible to build dynamic templates based on user inputs and pseudo parameters, such as AWS::Region.
- Use the Globals section to simplify templates.
- Use ExportValue and ImportValue to share resource information across stacks.

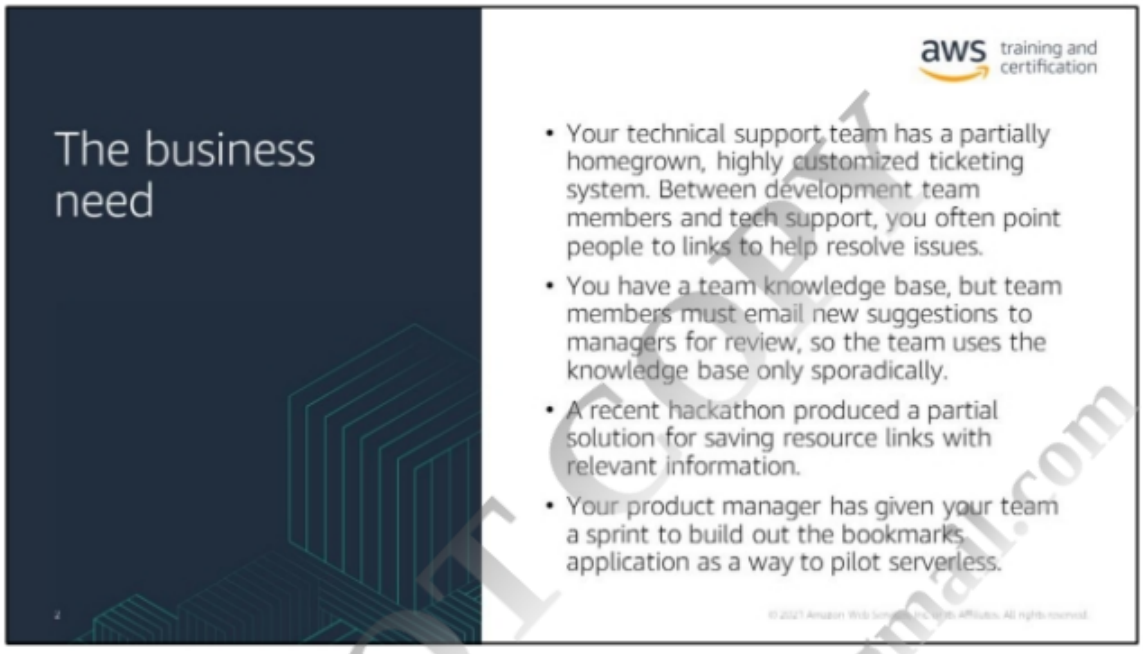




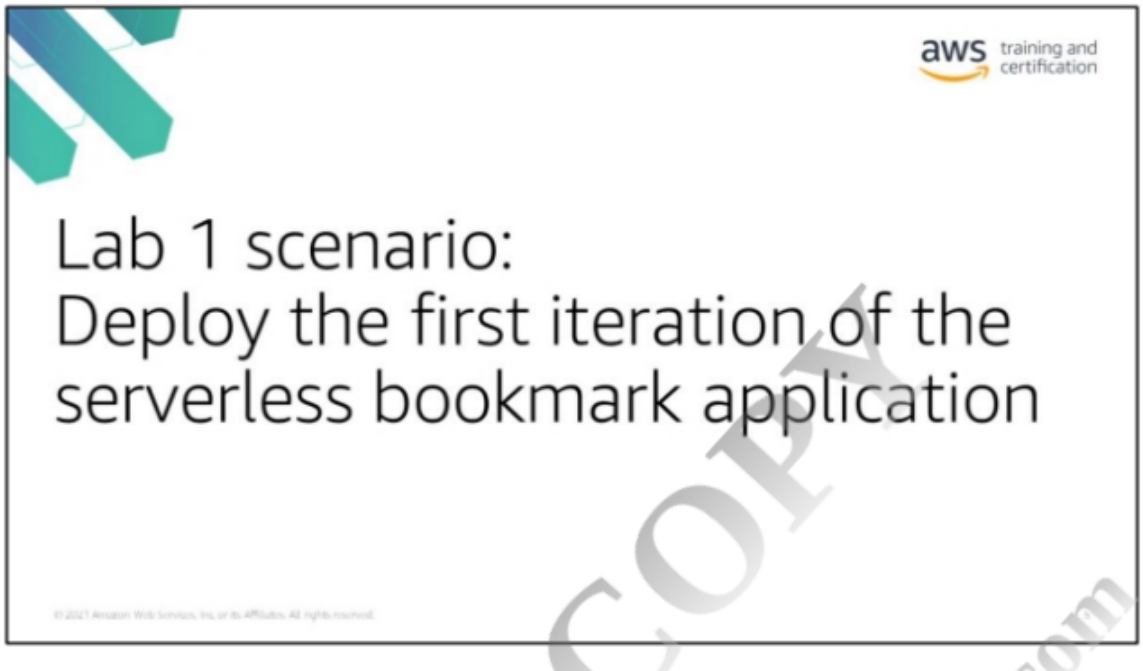


In today's labs, you will use AWS Step Functions to add functionality to your application and then improve the observability of your application.

These labs pick up your bookmarks application where you left it yesterday.



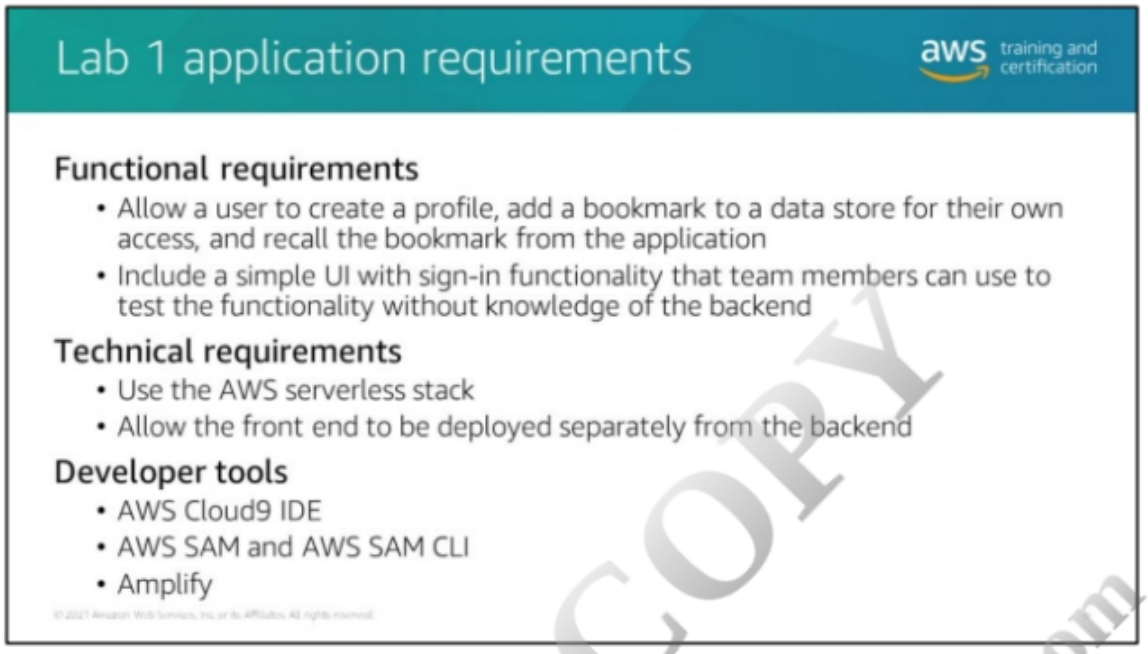
Lab 1: Deploying a Simple Serverless Application
Lab 2: Message Fan-out with Amazon EventBridge
Lab 3: Workflow Orchestration Using AWS Step Functions
Lab 4: Observability and Monitoring
Lab 5: Securing Serverless Applications
Lab 6: Serverless CI/CD on AWS



Let's return to the business need presented in the course introduction: Your technical support team has a partially homegrown, highly customized ticketing system.

Between development team members and tech support, you are often pointing people to links to help resolve an issue, but there are no real strategies for managing the resources. You have a team knowledge base that the development managers maintain, but team members must email new suggestions to a shared email inbox for review before they are published. So submissions, reviews, and updates happen sporadically.

A recent hackathon produced a partial solution for saving resource links with relevant information. Your product manager has asked your team to build the first iteration of this application as an opportunity to address the business need while learning about and piloting serverless.



In lab 1, you will deploy the first serverless iteration of the application based on the existing code to incorporate the requirements noted.