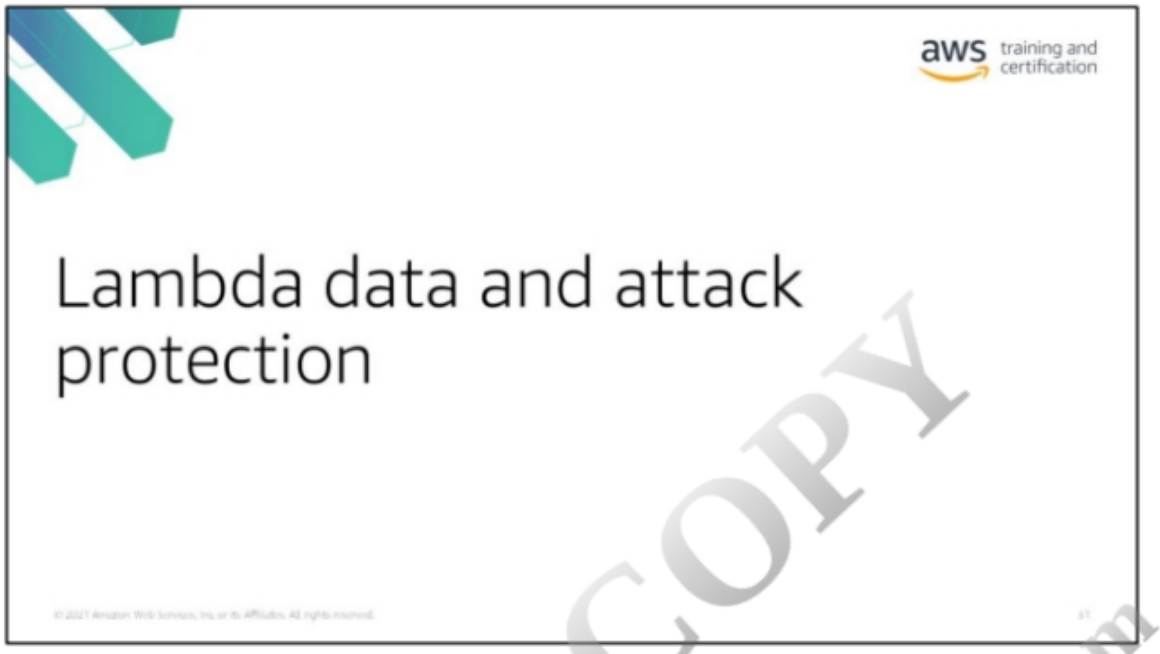
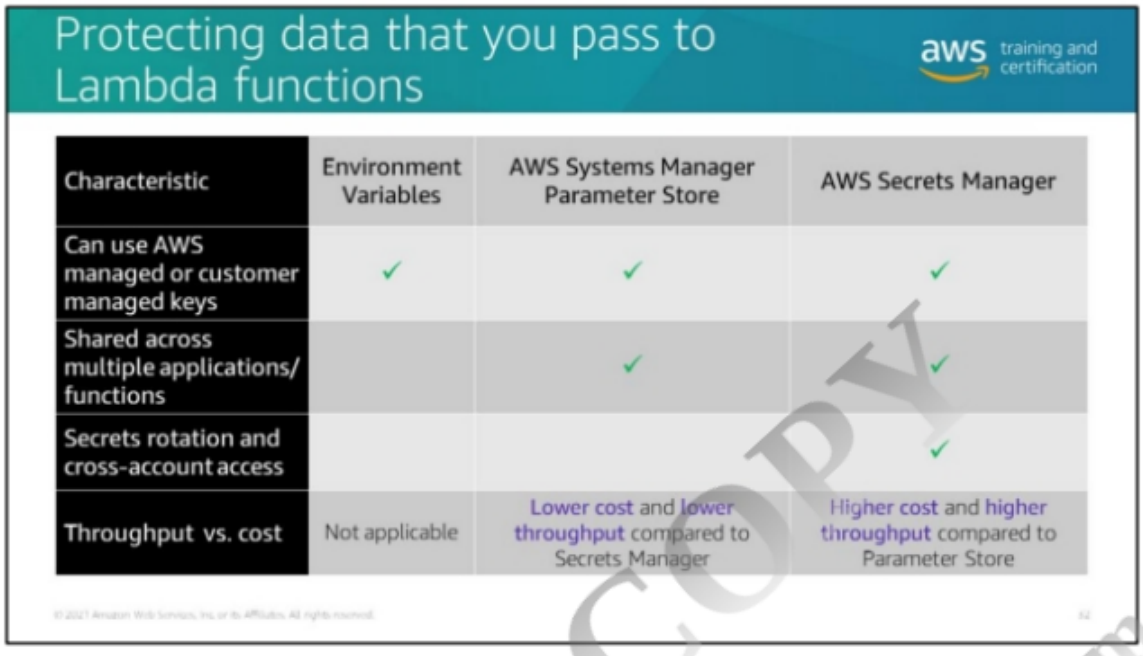




When it comes to passing data to Lambda functions, you have three options:

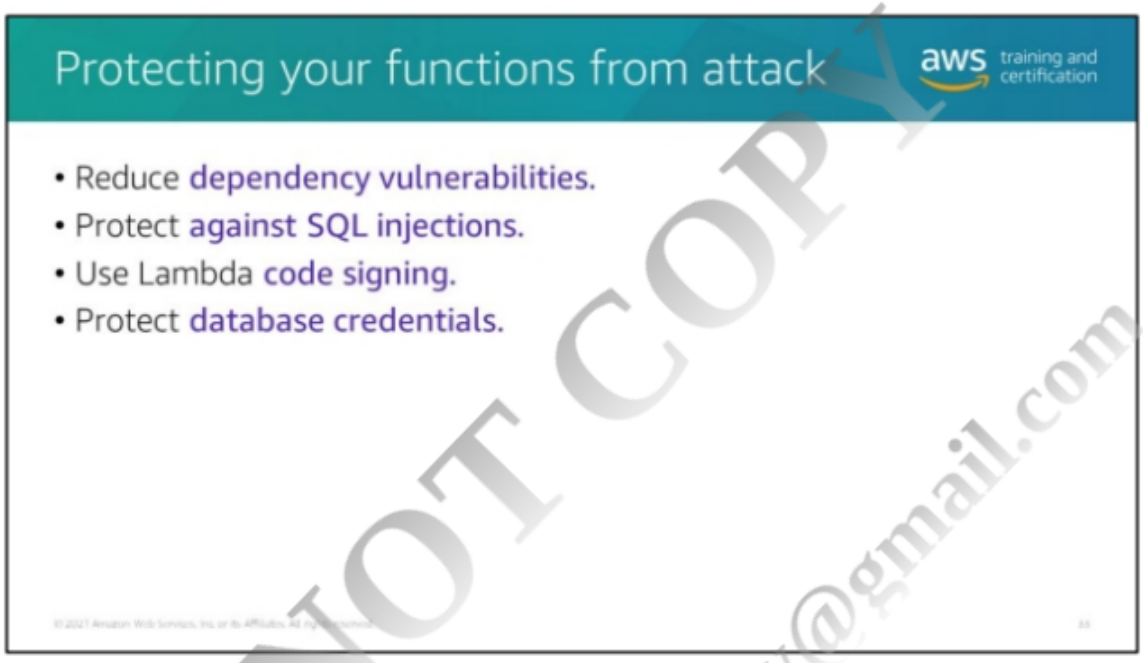
- Environment variables
- Values in AWS Systems Manager Parameter Store (https://docs.aws.amazon.com/systems-manager/latest/userguide/systems-manager-parameter-store.html)
- Values in AWS Secrets Manager (https://docs.aws.amazon.com/secretsmanager/latest/userguide/intro.html)

All three can use either AWS managed or customer managed keys to protect sensitive information.

Environment variables are scoped to a single function.

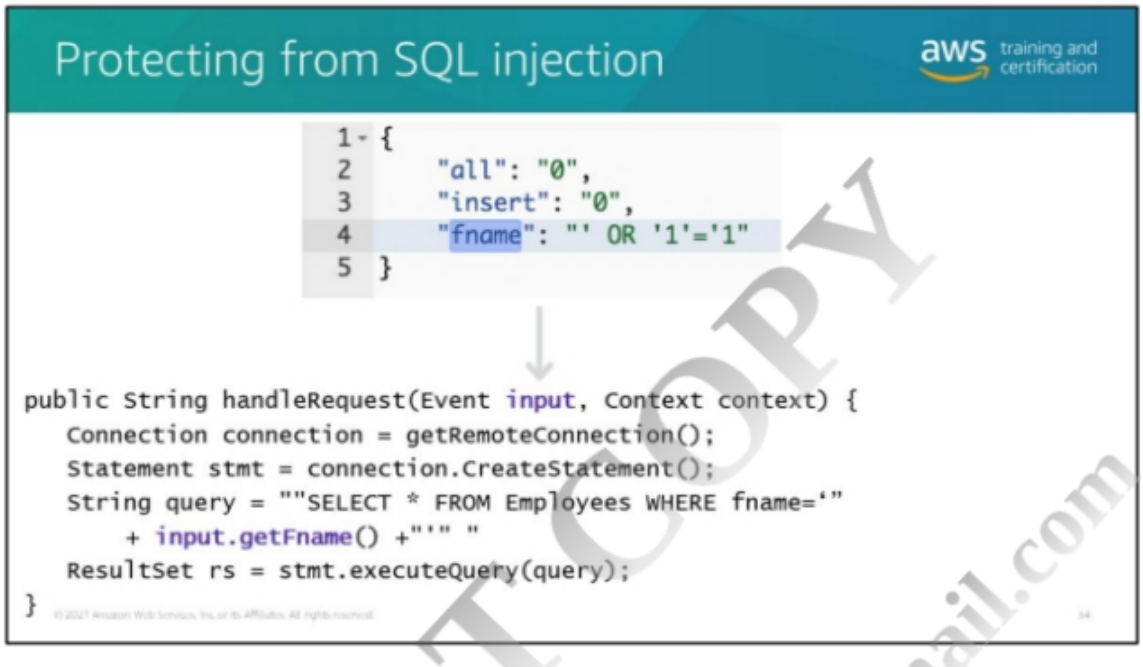
Values in Parameter Store or Secrets Manager can be shared across multiple applications. Secrets Manager has the added benefit of secrets rotation and cross-account access.

Parameter Store has a lower cost than Secrets Manager, but Secrets Manager provides better throughput than Parameter Store.



Let's look briefly at these considerations for protecting against attacks in your code.

The first is to reduce dependency vulnerabilities and protect against SQL injections. You could use third-party partners to scan for outdated dependencies, and you should do that as a best practice on all types of code.



Lambda functions are initiated by events. These events submit an event parameter to the Lambda function. In this example, a SQL injection on the "fname" string is passed into your Lambda function handler. If you trust user input on the Lambda function and run this Java snippet, your results are not protected.

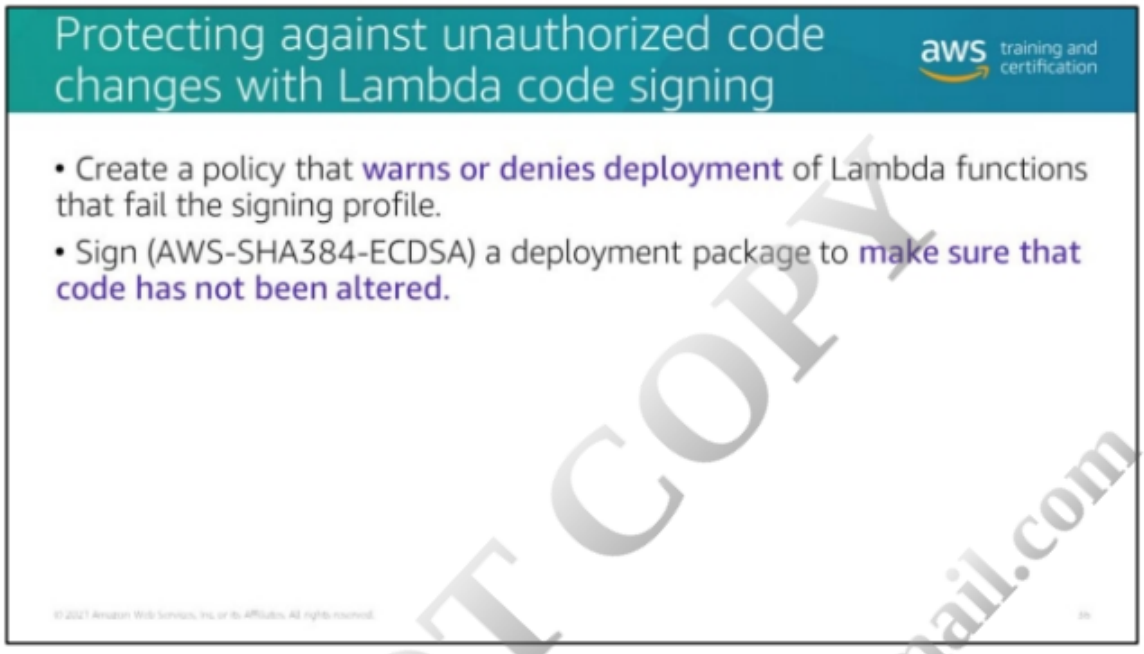
Use prepared/pre-compiled SQL statements to prevent SQL injection

aws training and certification

```
String selectString = "SELECT * FROM EMPLOYEES WHERE fname =
?";
PreparedStatement getEmployee =
connection.prepareStatement(selectString);
getEmployee.setString(1, "george");
getEmployee.executeQuery();
```

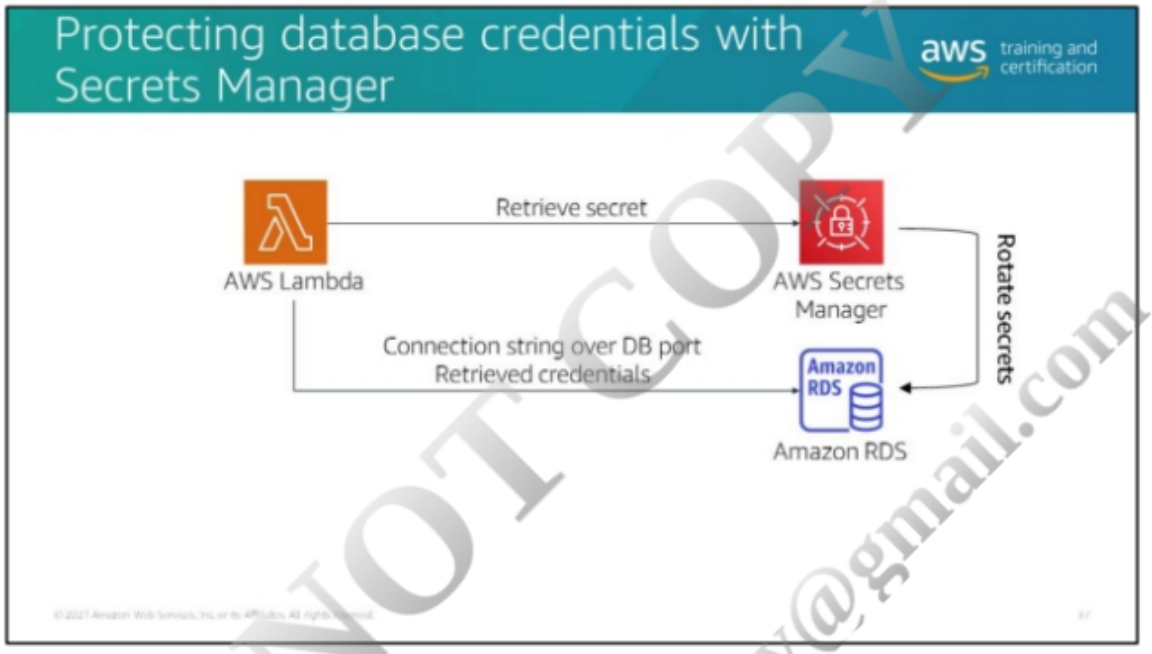
You can prevent this type of attack by parameterizing queries so that your Lambda function expects a single input.

This is an example, also in Java, that enforces single inputs.



Lambda code signing lets you set a profile that is used to determine how functions within that organization must be signed when uploaded to Lambda. Admins can create the resource code-signing configuration. Each code-signing configuration profile has a signature validation policy that either warns but allows an upload that isn't signed or denies uploads of code that isn't signed with the correct signature.

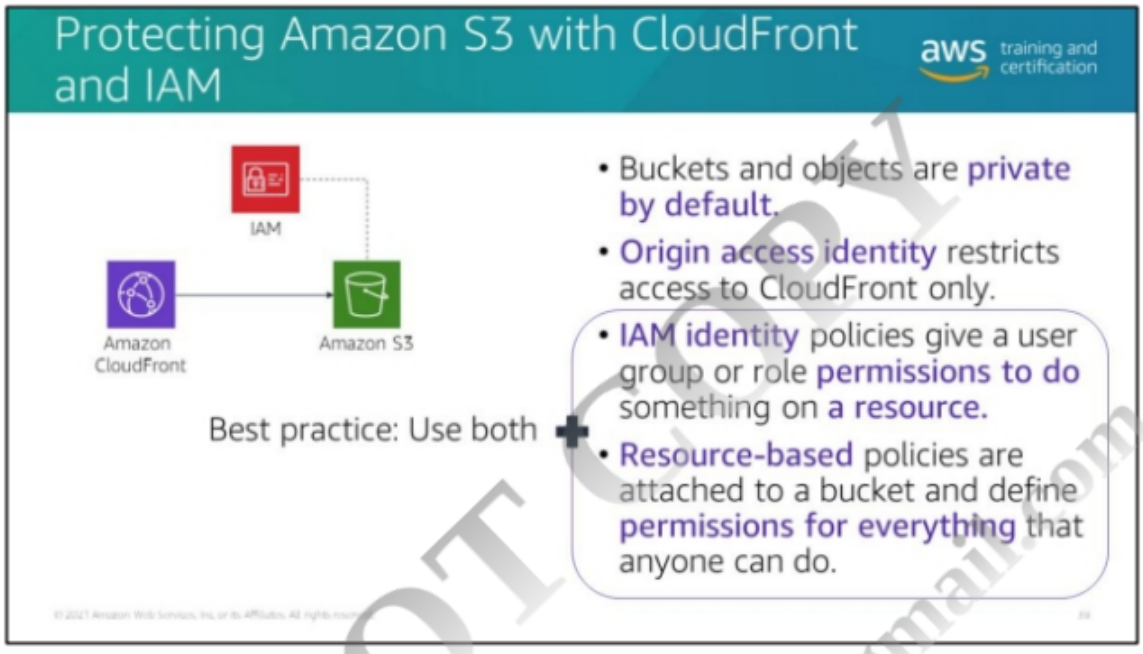
For more information about code signing with Lambda, visit https://aws.amazon.com/blogs/aws/new-code-signing-a-trust-and-integrity-control-for-aws-lambda/.



Use Secrets Manager to protect database credentials. Secrets Manager takes care of the following:

- Storing the credentials securely.
- Giving you SDK methods to retrieve them.
- Letting you rotate these credentials without disrupting applications for Amazon Relational Database Service (Amazon RDS), Amazon Redshift, and DocumentDB. This makes it much easier to use temporary credentials versus long-term ones that are less secure.
- Allowing you to secure access to the credentials with IAM.

Secrets Manager simplifies and enforces security best practices for handling secrets. For more information about Secrets Manager, visit https://docs.aws.amazon.com/secretsmanager/latest/userguide/intro.html.



By default, all Amazon S3 resources—buckets, objects, and related sub-resources—are private: only the resource owner (the AWS account that created it) can access the resource.

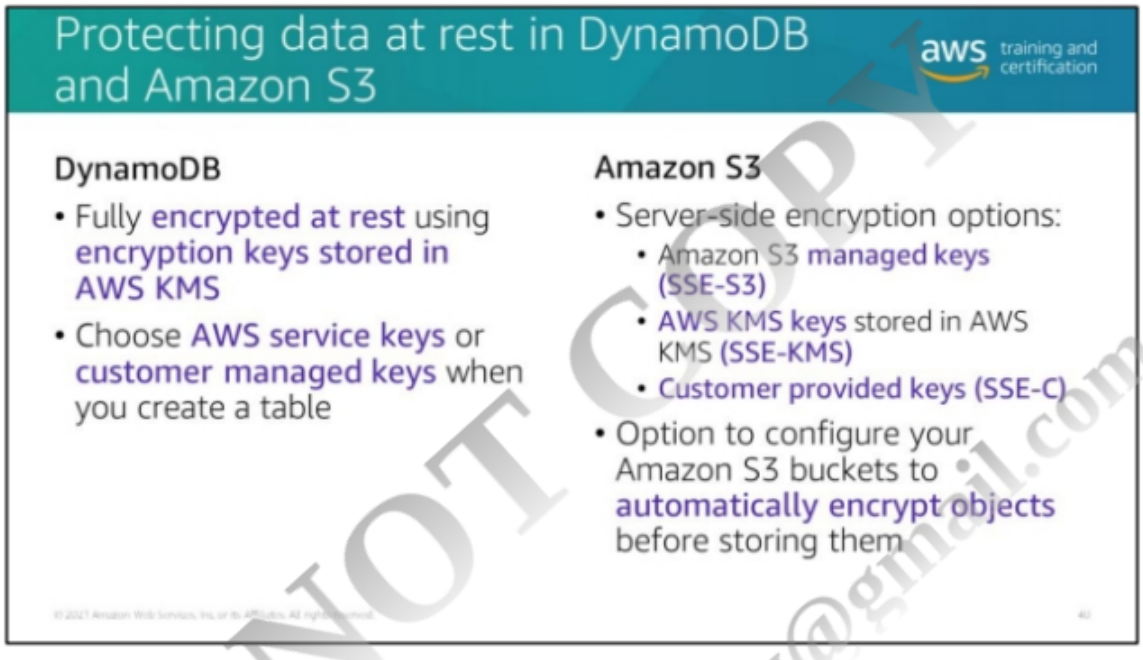
You might use Amazon S3 to host static data for a public-facing web application. Using origin access identity (OAI) with CloudFront locks down the bucket so that only CloudFront can read it.

Amazon S3 supports two types of access policies: identity-based (user) and resource-based (bucket) policies.

With identity-based access, permissions are associated with an identity, and those who have the identity can perform the actions in the policy. For example, a user policy might give two IAM users access to a specific bucket where they can add, update, and delete objects but limit their access to a single folder within that bucket. For examples, visit https://docs.aws.amazon.com/AmazonS3/latest/dev/example-policies-s3.html.

With resource-based access, the permissions are associated with the bucket or objects in the bucket. For example, you might deny permissions to perform operations on a specified bucket unless the request originates from within a range of IP addresses. An access control list (ACL) is another example of a resource-based policy. For examples, visit https://docs.aws.amazon.com/AmazonS3/latest/dev/example-bucket-policies.html.

As a best practice, grant least privilege access to users, groups, and roles and lock down individual resources.



In terms of protecting data at rest, AWS data stores provide encryption at rest via configuration options that you control. Encryption options take advantage of AWS Key Management Service (AWS KMS) and keys that you or AWS manage.

It's important to understand the default and configurable behaviors for encryption. For example:

- Amazon S3 supports server-side encryption using one of three options:
  - Amazon S3 managed keys
  - AWS KMS keys stored in AWS KMS
  - Customer provided keys
- You can configure your Amazon S3 buckets to automatically encrypt objects before storing them if the incoming storage requests do not have any encryption information.

All data stored in DynamoDB is fully encrypted at rest using encryption keys stored in AWS KMS. You choose whether to use AWS service keys or customer managed keys when you create a table.

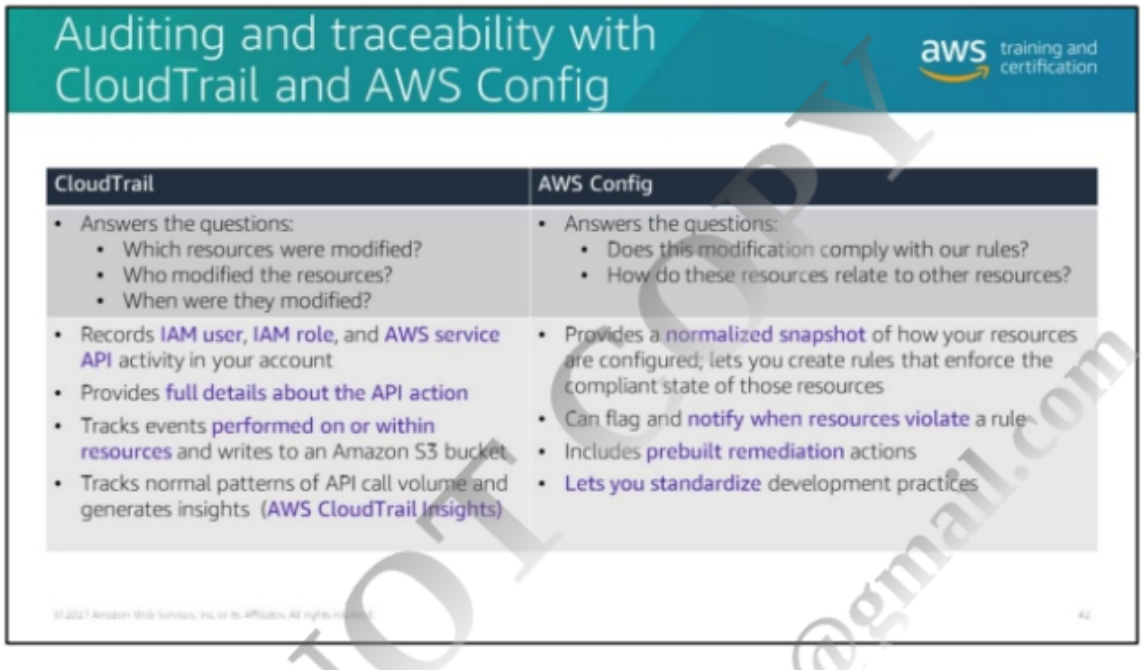
Amazon ElastiCache for Redis has an optional encryption feature using either managed or customer provided keys via AWS KMS.

Addressing this best practice:

- Monitor, alert, and audit actions and changes to your environment in real time.
- Integrate log and metric collection with systems to automatically investigate and take action.

You looked at options for monitoring your application in the previous module using AWS X-Ray and Amazon CloudWatch. You can certainly use metrics and alarms to alert to security threats.

This section looks a little closer at two options that help you audit and control changes to your application: AWS CloudTrail and AWS Config.



CloudTrail

CloudTrail is enabled when you create an account. When activity occurs in your AWS account, that activity is recorded in a *CloudTrail event*, and you can see recent events in the event history. The CloudTrail event history provides a viewable, searchable, and downloadable record of recent CloudTrail events. Details of API actions include the identity of the requestor, time of the API call, request parameters, and response elements returned by the service. Use this history to gain visibility into actions taken in your AWS account in the AWS Management Console, AWS SDKs, command line tools, and other AWS services.

A *trail* is a configuration that delivers CloudTrail events to an Amazon S3 bucket, CloudWatch Logs, and CloudWatch Events. Create your own trail to maintain a longer history of events. Trails track events performed on or within resources in your AWS account and write them to an Amazon S3 bucket. For example, a trail captures modifications to your API Gateway APIs. Optionally, add data events to track Amazon S3 object-level API activity (for example, a user uploads to the bucket) or Lambda invoke API operations on Lambda functions in the account.

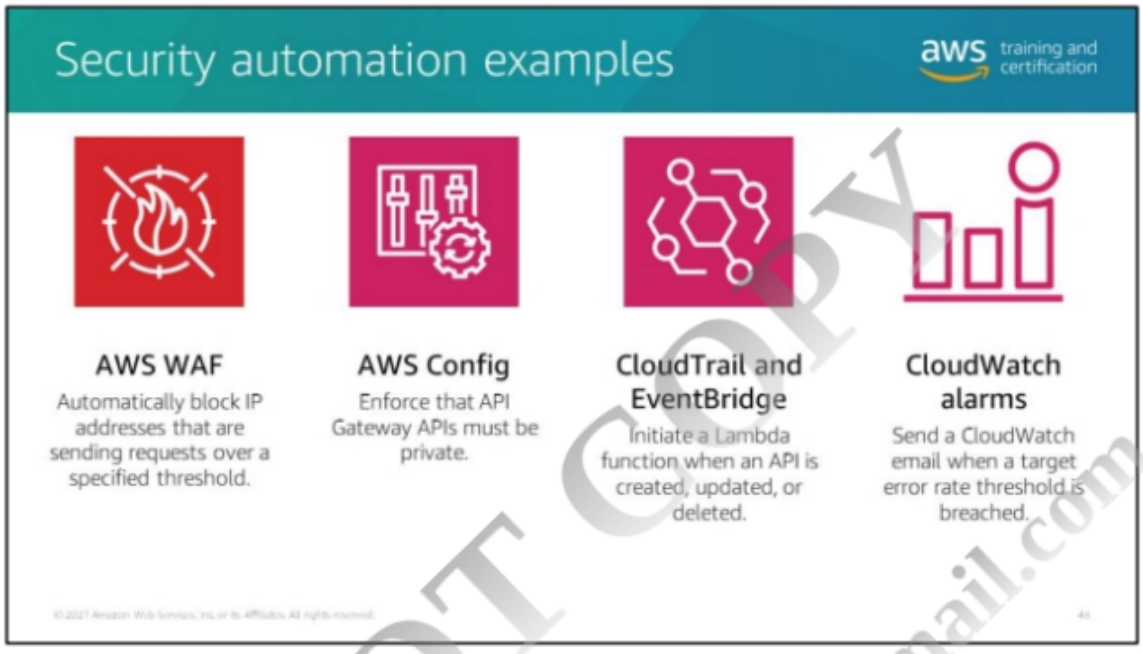
Configure *AWS CloudTrail Insights* to help you identify and respond to unusual activity associated with write API calls. CloudTrail Insights tracks normal patterns of API call volume and generates insights when the volume pattern is abnormal.

AWS Config

Alternatively, AWS Config provides a normalized snapshot of how your resources are configured and lets you create rules that enforce the compliant state of those resources. An AWS Config rule represents desired configuration settings for specific AWS resources or for an entire AWS account. If resources violate a rule, AWS Config flags this as noncompliant and notifies you through Amazon Simple Notification Service (Amazon SNS).

AWS Config provides customizable, predefined rules to help you get started. You might also want AWS Config rules that help standardize how your developers write Lambda functions. You might have rules based on runtime environment, handler name, code size, memory allocation, timeout settings, concurrency settings, or execution role. This gives you a holistic view of the Lambda function's lifecycle, and enables you to surface that data for potential audit and compliance requirements.

AWS Config includes prebuilt remediation actions and options to automatically remediate an issue. Connect AWS Config to CloudWatch Events and initiates actions off selected events.



Many of the services discussed in this module help to enforce the security design principle of automating security. The following are just a few examples of how these services can work together to provide automated security checks and remediation.

AWS WAF

AWS WAF provides a set of AWS managed rules to provide protection against common application vulnerabilities or other unwanted traffic, without having to write your own rules. In addition, you can automate updates to AWS WAF rules using Lambda, which can analyze web logs, identify malicious requests, and automatically update security rules. For example, AWS WAF rules can be updated to automatically block IP addresses that are sending requests over a specified threshold.

AWS Config

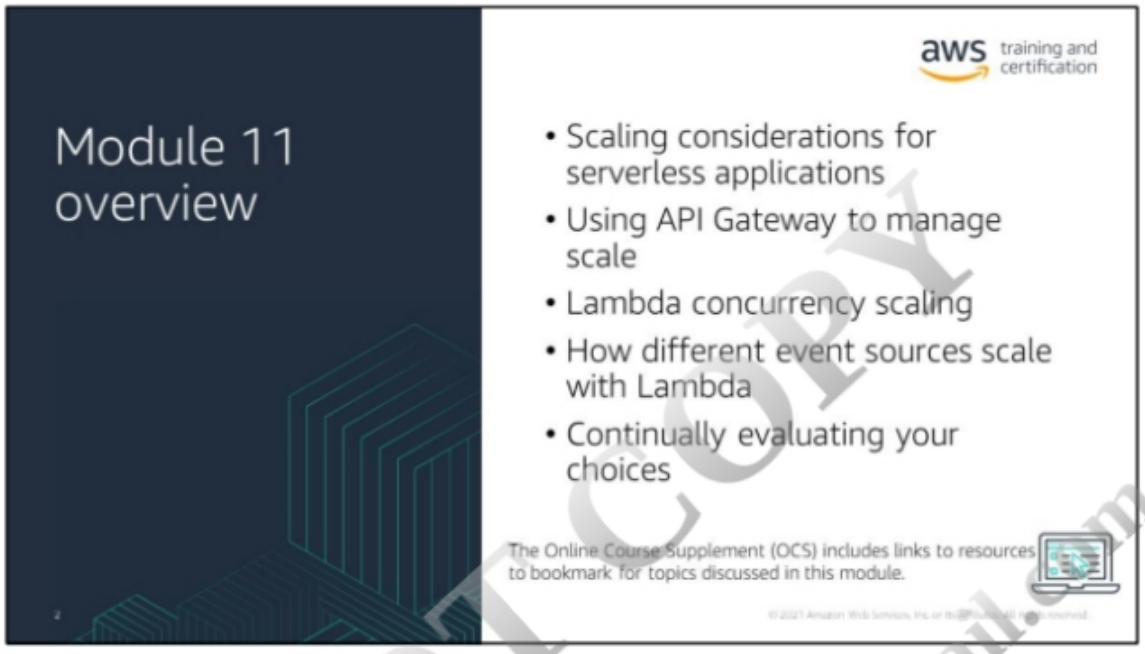
AWS Config provides a set of AWS managed rules to evaluate whether your AWS resources comply with common best practices. You can write custom rules. These rules can identify whether a resource is compliant or not. You can manually or automatically remediate noncompliant resources. For example, it would be possible to enforce that APIs defined in API Gateway must be private. Any attempt to change to a Regional or edge API could invoke a function to update the API back to a private endpoint.

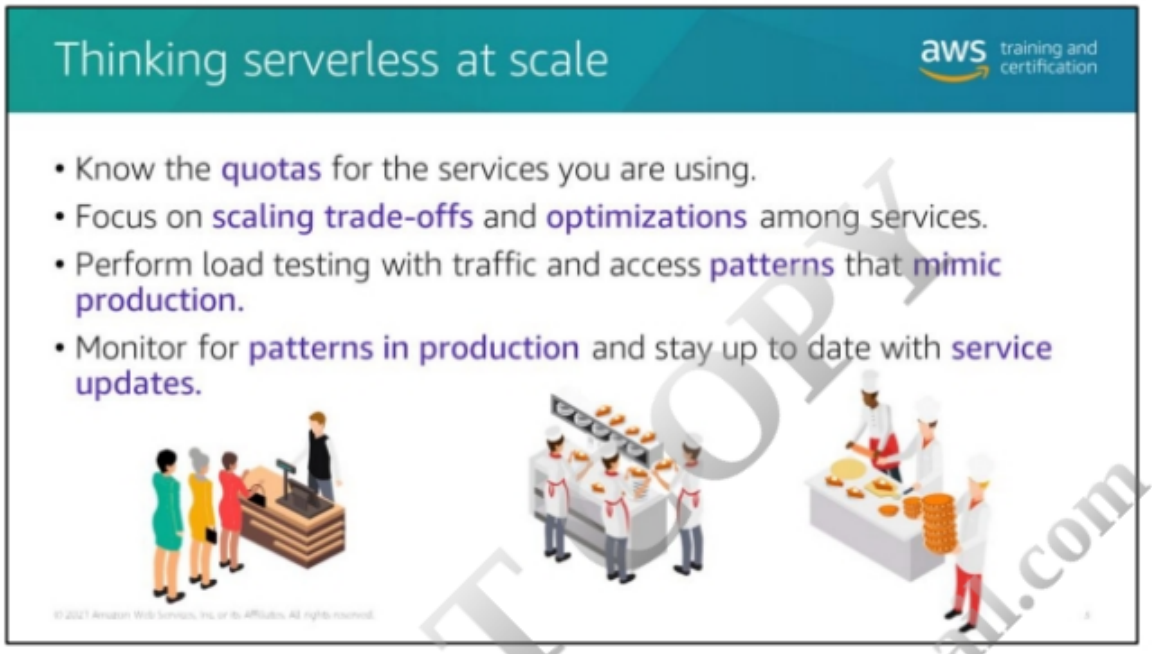
CloudTrail and EventBridge

CloudTrail can detect when management API calls are made against API Gateway and send notifications to EventBridge. EventBridge is a serverless event bus service that makes it easy to connect your applications with data from a variety of sources. EventBridge has the ability to direct messages to a number of available targets based on matched rules. For example, a rule could be created to invoke a Lambda function when an API is created, updated, or deleted.

CloudWatch alarms

CloudWatch alarms have the capability to send notifications to an Amazon SNS topic. When a target threshold for an error rate is breached, subscribers could receive an email notification, a Lambda function could be initiated, or a message could be published to an HTTP/S target.





As you migrate from on premises to cloud and to serverless, build for the scale you need and optimize for the scale you will grow into.

As noted in the module on thinking serverless, one of the benefits of a distributed serverless architecture is that you can independently update your components and iterate and learn from each iteration. This is also one of the challenges because you have to understand how increasing the scalability of service A impacts service B.

Each service has quotas that may impact your design choices about how your application scales and will likely be the cause of errors you find when you start to put all your pieces together under load.

As your solutions evolve and your usage patterns become clearer, you should continue to find ways to optimize performance and costs and make the trade-offs that best support the workload you need rather than trying to scale infinitely on all components.

Don't expect to get it perfect on the first deployment, but build in the kind of monitoring and observability that the course discussed earlier to help you understand what's happening, and tweak things that make sense for the access patterns that happen in production. Let's return to the food truck, which turned into a diner, to highlight the balancing act that you need to maintain.

Customers come in and want to place an order and have it fulfilled quickly, but they also want it to be at an acceptable level of quality. What's the right balance of letting customers wait to get the order "just right"?

Your chefs have tasks of varying length and difficulty, some of which require more precision than others. Your chefs pull different types of resources for their ingredients; what do they need at their fingertips, and what requires a trip to the back of the kitchen?

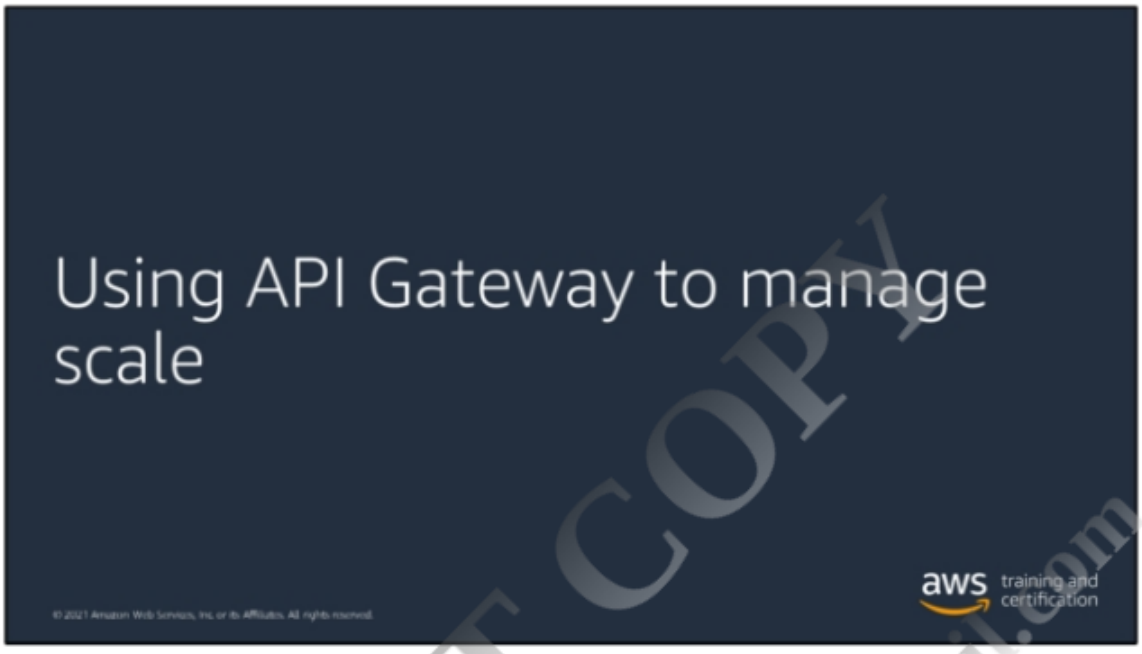
To deliver a complete order successfully, you must understand the slowest task in the chain and find ways to increase efficiency without overwhelming the system. And you want to spend resources only if that increase makes a difference to your customers.

How many orders should you try to prepare at one time? What is the maximum number of people that you can put at a table and manage the order without dropping things? What is your customers' tolerance for mistakes that can be resolved quickly?

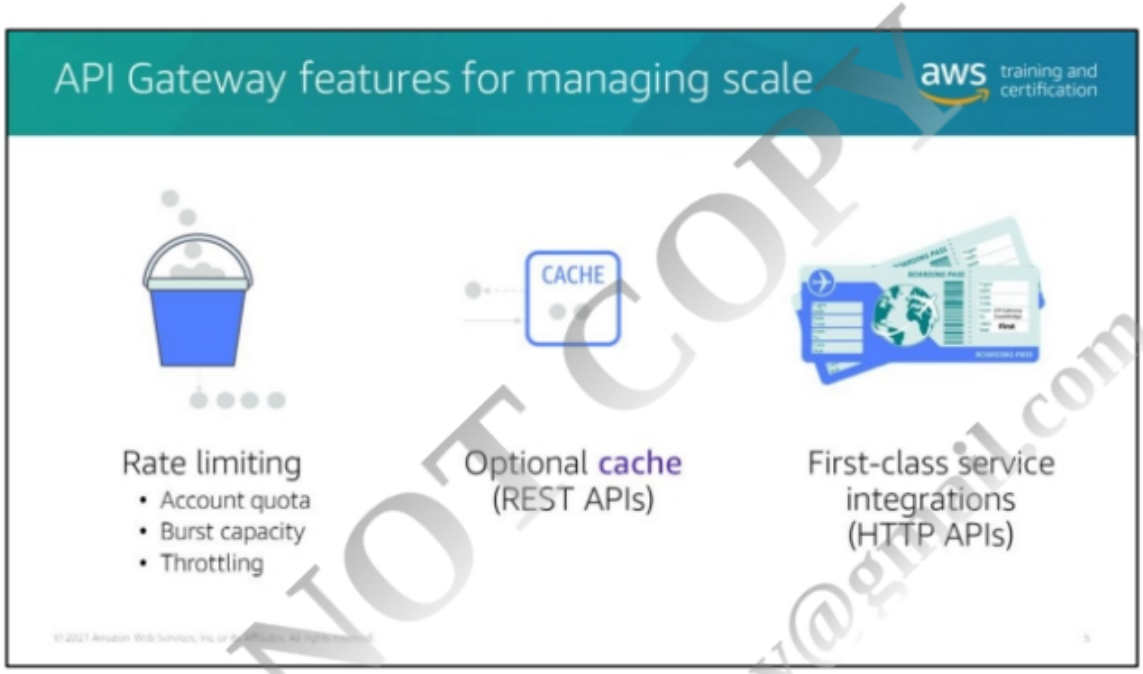
No matter how much you strategize about how things will work, what really matters is what happens with your customers. When do they arrive, and what do they order most often? Where do you get complaints, and what are they? What are customers doing that you hadn't predicted (for example, ordering late-night pancakes or burgers topped with eggs)? To sustain your success, you need to pay attention to how things are working in production and make adjustments that align to the patterns that are occurring.

As with any business, finding ways to simplify or increase the speed of your current process might improve your margins and your customer satisfaction. Maybe one of the donut bakers has found a method of preparing dough that decreases rising times, or maybe there's a new pan that lets you cook more eggs at once. You want to stay abreast of new products and practices.

In the next section of this course, you will look at considerations for managing scale in serverless applications and some of the trade-offs that you will make in designing your serverless application.



Earlier, you looked at how Amazon API Gateway features help prevent issues like distributed denial of service (DDOS) attacks on your serverless application. You can use those features in other ways to manage scaling in your application.



Rate limiting

As discussed in the application security module, API Gateway has rate-limiting features that reduce the impact of external attacks. You can also rate limit the traffic hitting your application, which might be helpful if you need to buffer request volume for downstream systems. The account quota is a soft limit, and you can request an increase if you need very high throughput.

However, the burst capacity uses the token bucket algorithm and is determined by the API Gateway service team. You cannot modify the burst capacity. At a high level, the token bucket algorithm means that, as requests come into the bucket, they are fulfilled at a steady rate. If the rate at which the bucket is being filled causes the bucket to fill up and exceed the burst value, a 429 Too Many Requests error is returned. For more information on API Gateway quotas, visit https://docs.aws.amazon.com/apigateway/latest/developerguide/limits.html. Because the burst capacity applies across all APIs in a given Region and account, it's important that if you are expecting very high throughput on your APIs, you include tests that exercise all of the APIs that might be used at the same time in the account and Region.

The next slide looks more closely at throttling options that you can control per API.

API Gateway cache

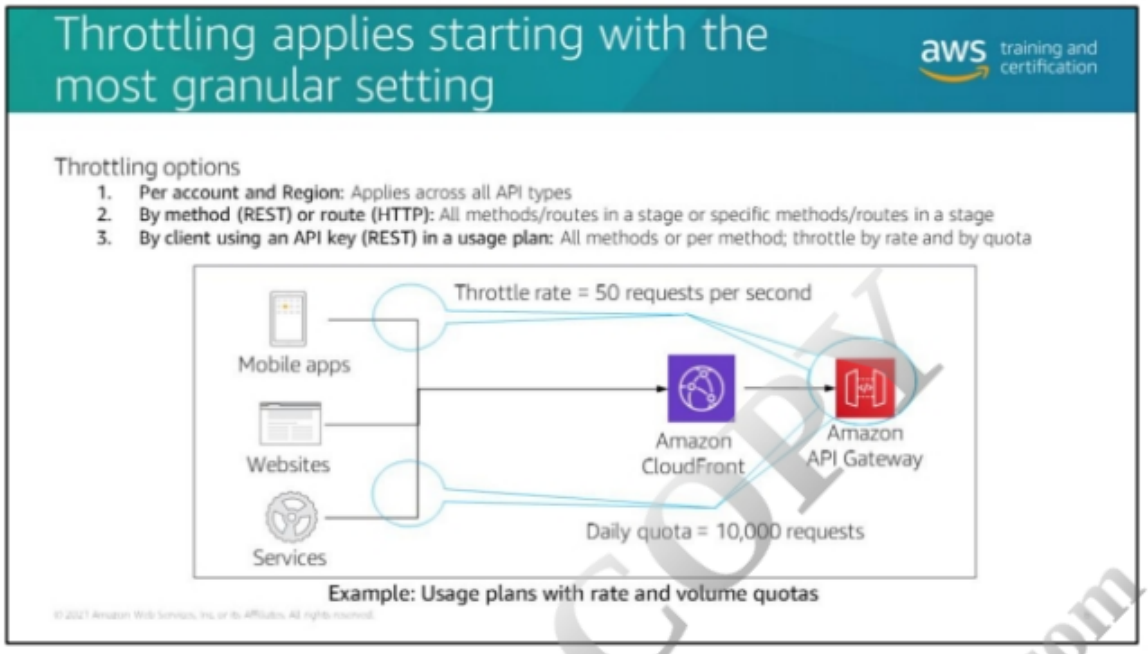
When you enable the API Gateway cache, API Gateway caches responses from your endpoint for a specified Time to Live (TTL) period. This reduces latency on the response and decreases the number of calls to the backend. When you enable caching for a stage, only GET methods have caching enabled by default.

Note that caching is charged by the hour and is not eligible for the AWS Free Tier, so you want to analyze your traffic and access patterns to find where caching will optimize your application.

For more information about API Gateway caching, visit https://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-caching.html.

First-class integrations

With first-class integrations, you can call service APIs directly from your HTTP API. This decreases end-to-end latency. For more about which services you can use for first-class integrations and how to use them, visit https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-develop-integrations-aws-services.html.



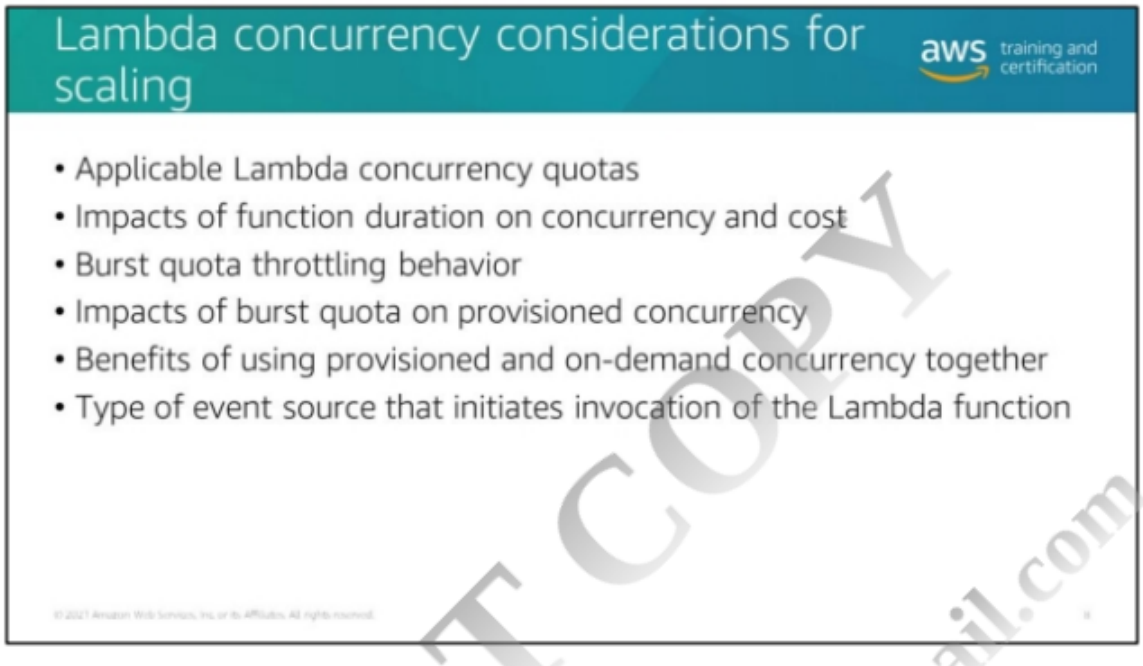
In addition to the account-level controls, API Gateway provides throttling at different levels so that you have granular control over where throttling applies. Throttling options include the following:

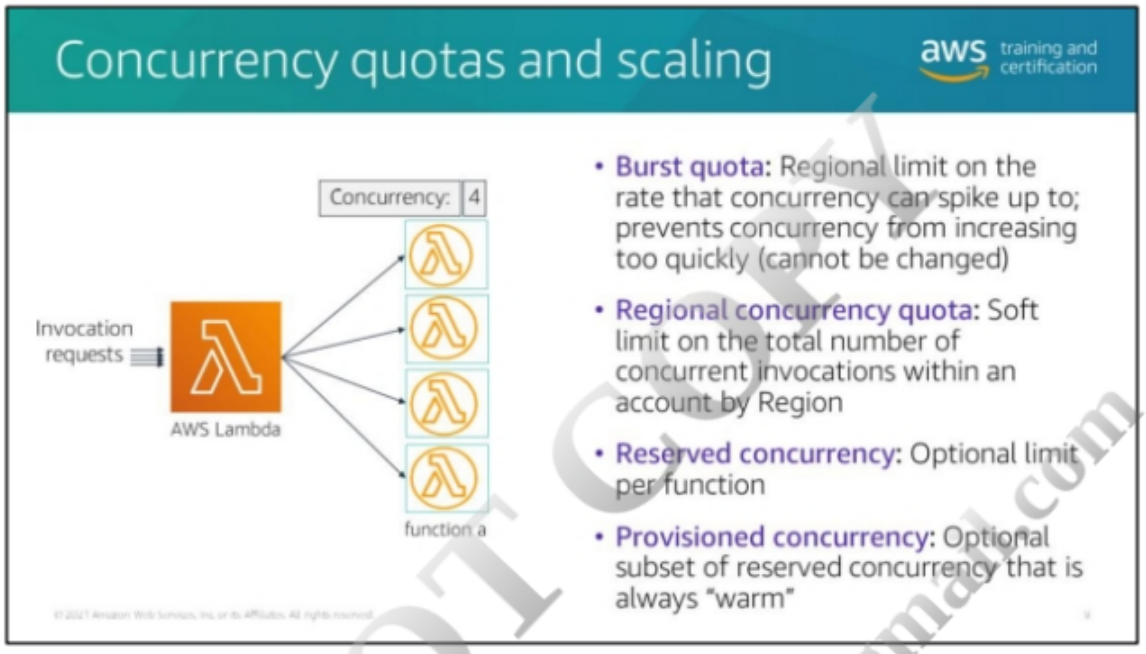
- Apply throttling to all methods or routes within an API (default throttling)
- Set throttling per method or route
- Assign throttling behaviors based on client via usage plans and API keys (REST APIs only)

For more information about throttling requests to your HTTP API, visit https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-throttling.html. For more information about throttling API requests for better throughput, visit https://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-request-throttling.html.

Settings are applied to the most granular control first. For example, if an incoming request is from a client tied to a usage plan on a REST API, throttling related to individual methods for that client is applied first and followed by any client-level limits. After that, any throttling settings indicated for a particular method and stage are applied first, followed by any limits placed across the entire stage. Finally, if no throttling is applied at more granular levels, the account quota is applied.

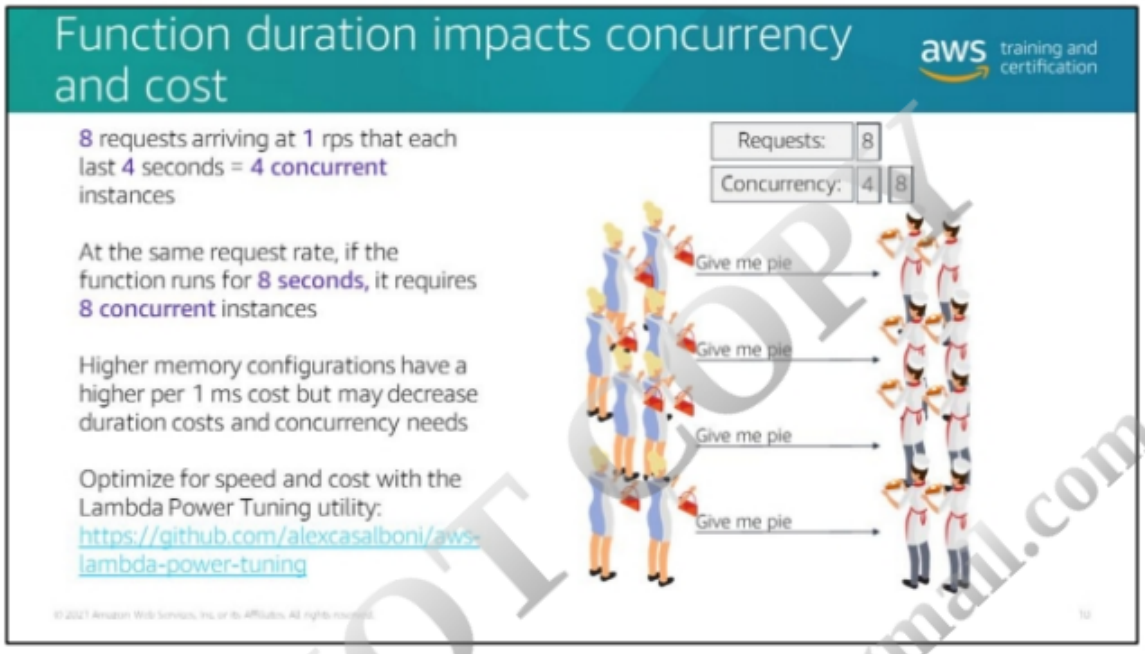
A usage plan specifies who can access deployed stages and methods, and controls the rate and number of requests that a client makes. Within usage plans, you can set throttling limits to control the request rate, and you can set a quota to control how many requests that API key can use within a specific time frame. API keys (REST) are unique string values that you give out to grant access to APIs. For information about creating and using usage plans with API keys, visit https://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-api-usage-plans.html.





As discussed earlier, **concurrency** is the number of AWS Lambda function invocations running at one time. Lambda provides environments on demand as it receives requests. Factors that influence concurrency include the following:

- **Burst quota:** Each Region has a burst limit that prevents concurrency from increasing too quickly in the event of a large spike of requests in a very short time. You cannot modify this limit. This burst quota does not limit the total number of invocations in a Region, but it does limit how quickly you can spin up a large number of new invocation environments in that Region.
- **Regional concurrency 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.
- **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.
- **Provisioned concurrency:** This keeps a number of instances always warm.



A key driver of how many concurrent instances your function will use is how long each individual invocation runs.

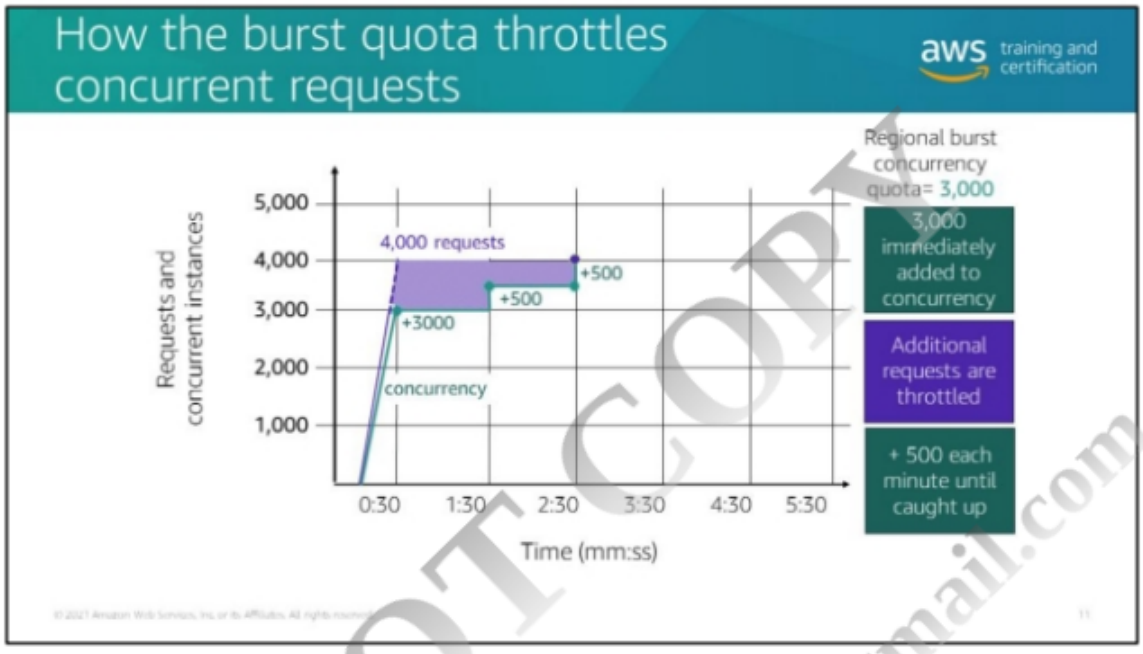
In this simple synchronous invocation example, the rate of requests is 1 request per second (1 rps), and on average the function runs for 4 seconds. In this case, Lambda needs four concurrent instances to keep up with requests. Requests 1-4 spin up a new instance, but request 1 ends before request 5 is made, so request 5 (and requests 6-8) get warm starts. No additional concurrency is required.

This is where some of the best practices discussed earlier come into play; for example, having functions that download quickly and initializing global variables outside the handler will help keep your function duration shorter on average. If the available concurrency is not enough to keep up with the concurrency needed by your application, requests are throttled.

It's also important to analyze your memory configuration choice because this choice can have a significant impact on function duration. As this course discussed earlier, the function memory configuration determines function CPU and I/O capacity. The impact of additional memory or CPU on your function duration is based on the type of work your function is doing.

Trying different memory configurations and testing the impact on average duration is an important part of tuning your function to scale for your needs. The increased memory may shorten function duration and thus require fewer concurrent invocations. Even though higher memory settings have a higher per gigabyte per second cost, the overall impact may be less expensive when multiplied by a much shorter duration.

The Lambda Power Tuning utility lets you test your function at different configurations and optimize for speed, cost, or a balance of both. The Online Course Supplement has a demonstration video of this utility. For more information about this utility, see https://github.com/alexcasalboni/aws-lambda-power-tuning.



Separately, there is a Regional burst quota that controls how quickly concurrency will increase in response to a burst of traffic. This quota differs depending on the Region and is not configurable. In this example, the function is running in a Region where the burst quota is 3,000.

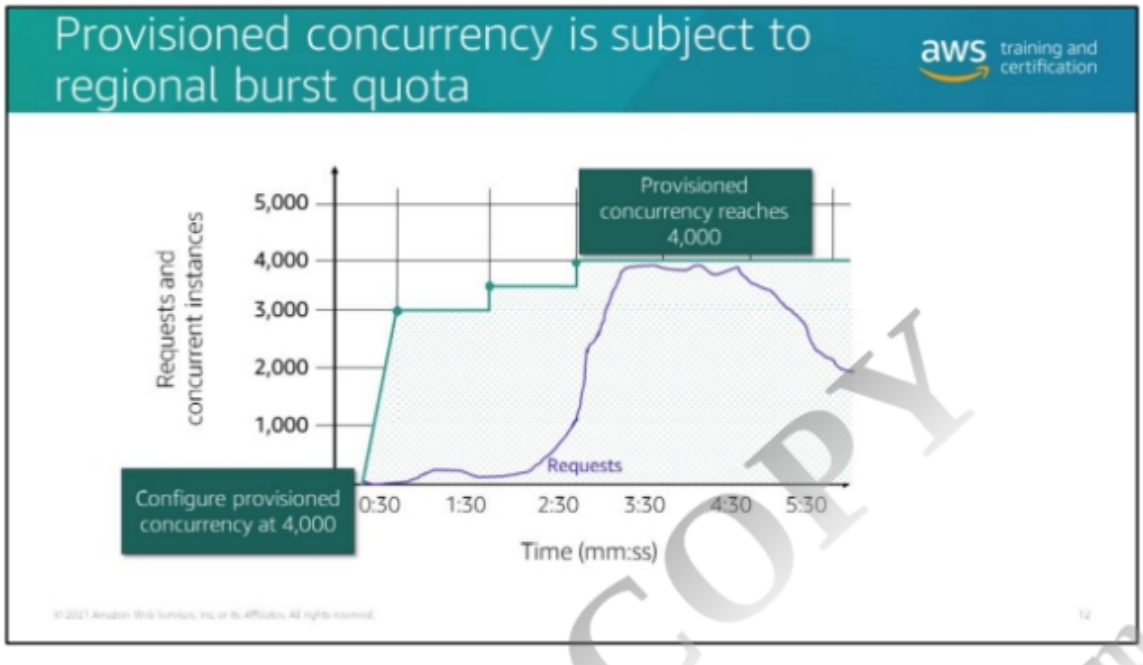
The example starts when the number of requests and concurrent invocations is 0 and requests suddenly burst to 4,000. Lambda immediately increases the number of concurrent invocations by the Regional burst concurrency quota (in this example, 3,000). This means that concurrent invocations go from 0 to 3,000 immediately.

After this immediate increase, Lambda adds up to 500 concurrent invocations per minute until it has enough to run all of the requests concurrently or until it reaches the function or account limit. In this example, Lambda adds concurrency for each of the next 2 minutes and at that point has enough concurrency to run the 4,000 requests concurrently. This means that during those 2 minutes, your requests are being throttled, and your application has to manage the backup. Keep in mind that all of those new invocations will be cold starts, so you need to account for that potential latency as well.

What might you do within the application to minimize the impact on your application?

1) Build your application with error-handling mechanisms to minimize the impact and accept this type of throttling. You will look at how each event source type scales with Lambda a little later.
2) Allocate provisioned concurrency in advance of a spike so that you have enough available concurrent instances to avoid any throttling.

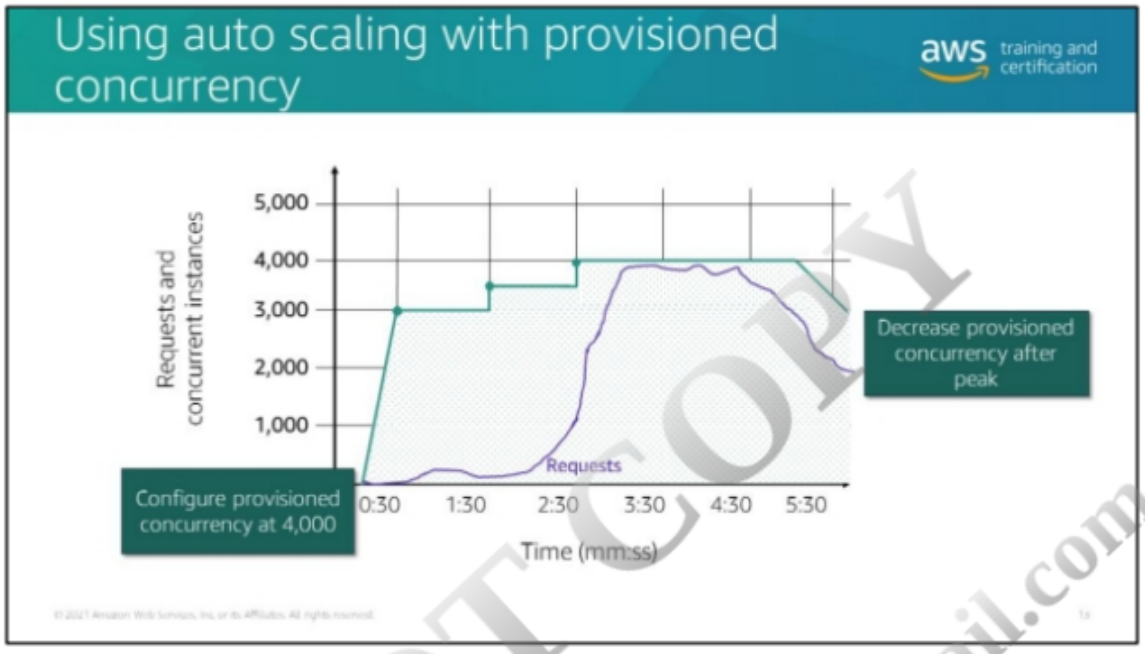
Let's look a bit closer at using provisioned concurrency to prevent this kind of throttling.



If you increase provisioned concurrency **in advance of** a spike, you can have warm instances ready to process the burst of requests. You can provision concurrency up to the reserved concurrency. Note that the time it takes to initialize your provisioned concurrency is subject to the same burst quota as on-demand instances, and you don't have access to any of that provisioned concurrency until the requested number of instances are ready.

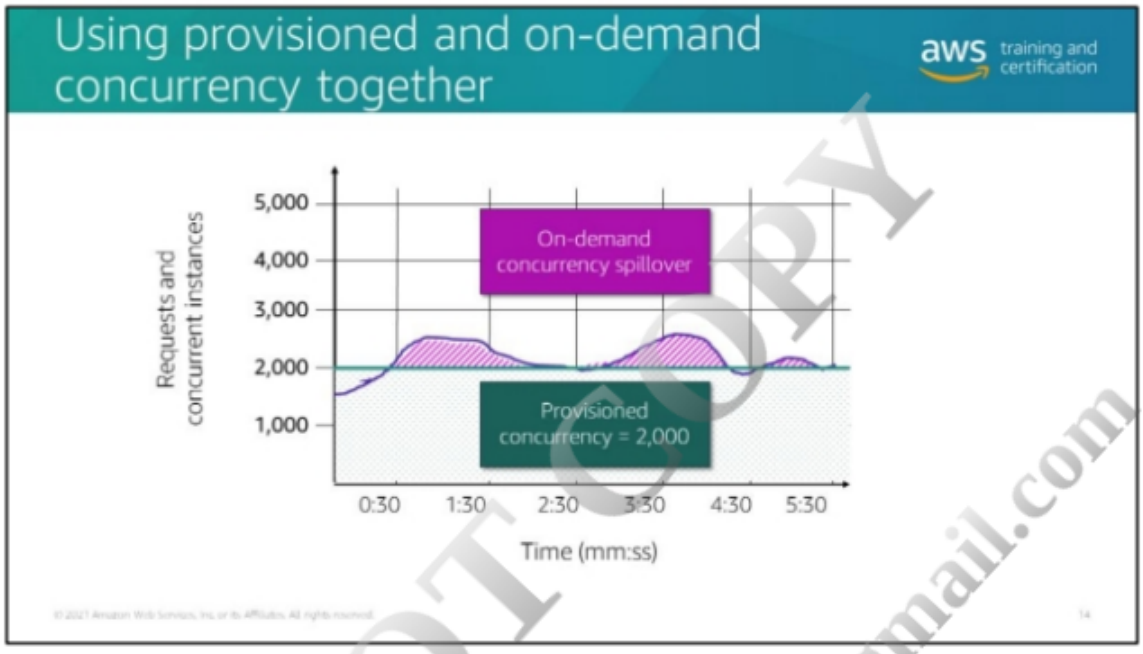
For example, if you are expecting a peak approaching 4,000 concurrent requests and you want to have 4,000 instances ready to go before that spike occurs, you need to allow enough time for Lambda to burst to the regional burst quota (for example, 3,000) and then increase by 500 instances per minute to reach 4,000. In this example, you need about 2 minutes in advance of the spike to have enough warm environments for all incoming requests. You don't want to continuously pay to keep 4,000 concurrent instances provisioned if you can predict when traffic will spike.

In the diner analogy, if your weekly special is Fried Egg Friday, you want to warm up extra burners before the morning rush on Friday, but you probably don't want to keep them on continuously once the rush has passed.



You can use AWS Application Auto Scaling to schedule the provisioned concurrency increase in advance of your traffic peak and then reduce it at a scheduled time when you expect the peak has passed.

What if you're not sure when the peak will come, but you expect that there will be some points when you know you will need to quickly handle bursting and waning? You can also use Application Auto Scaling to create a target tracking scaling policy that adjusts provisioned concurrency levels automatically based on the utilization metric that Lambda emits.



Based on expected traffic and an analysis of metrics and requests patterns, a combination of provisioned concurrency and on-demand concurrency may be the best approach for your application scaling, and it may also be more cost-effective. In this example, the request pattern is relatively steady and not prone to large spikes.

Here it is set at 2,000, and on-demand concurrency handles the spillover. This provides the majority of invocations with a warm start but prevents you from paying to keep instances warm that aren't used as often. Finding the balance between provisioned and on-demand concurrency depends on the criticality of a warm start, function duration and concurrency expectations, and the cost of keeping instances warm.

In this example, even if you aren't concerned about warm starts or extreme bursts, it makes sense to use provisioned concurrency for a portion of traffic because the cost for the **duration of a function** that runs on provisioned concurrency is actually **less** than the cost of running it in an on-demand instance for the same duration.

Depending on your requests patterns, it can cost less to use provisioned concurrency. You can dig into the details of pricing examples on the provisioned concurrency pricing page.

At a high level, if you are using your configured provisioned concurrency more than 60 percent of the time that it's configured, it will be more cost-effective to have it configured as provisioned concurrency rather than on demand, even if warm starts aren't a concern.

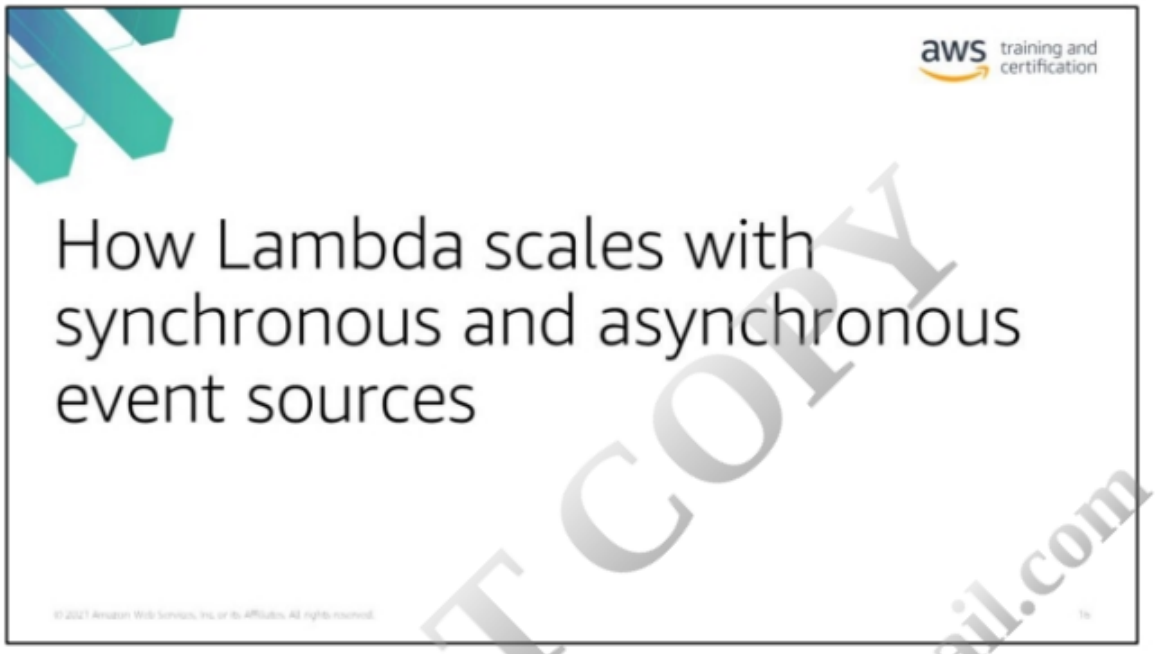
If your provisioned concurrency is used less than 60 percent during a given time period, then it will probably be less expensive to use on-demand concurrency if you don't need the provisioned concurrency to allow for unplanned bursts or provide warm starts.

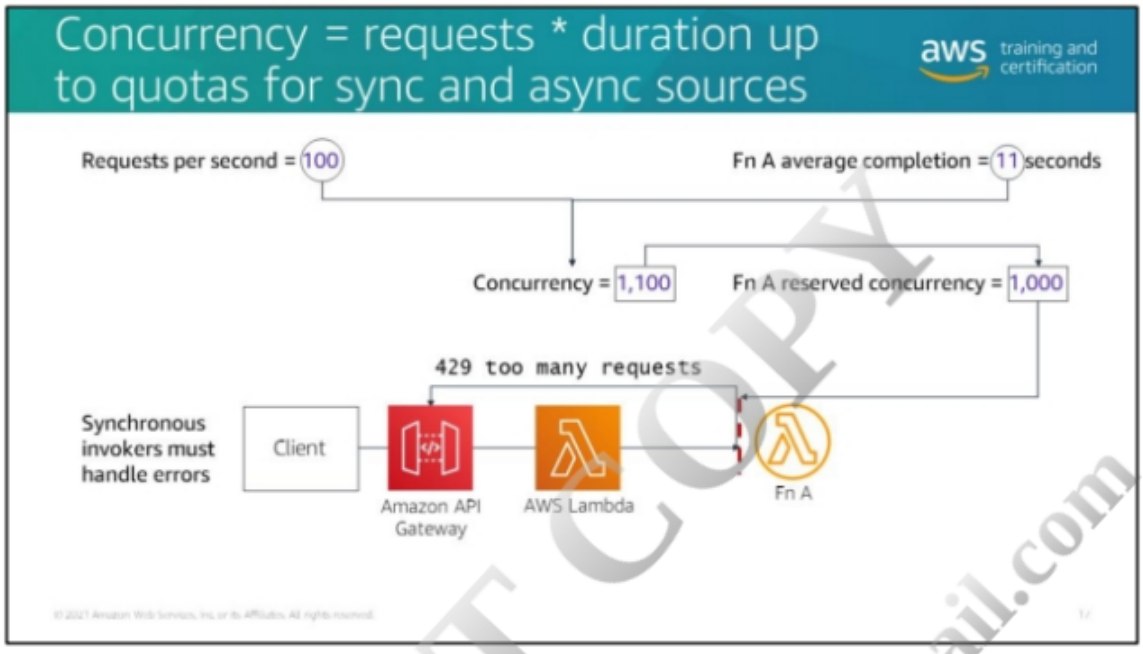
As noted earlier, discovering the right balance as your application scales should be an active exercise of reviewing metrics on invocations and spillover rate, analyzing cold starts, and learning more about production traffic patterns. For a calculator that estimates your costs based on expected traffic and concurrency settings, visit https://aws.amazon.com/lambda/pricing/. Go to the **Provisioned Concurrency Pricing** section. This link is also available in the OCS.

As you saw earlier in the course, Lambda handles aspects of interacting with the services that you use to invoke it, and there are differences based on the type of event source you choose.

This section looks closer at how Lambda manages scale across three categories of event sources:

- Synchronous and asynchronous
- Amazon Simple Queue Service (Amazon SQS) queues
- Amazon Kinesis Data Streams





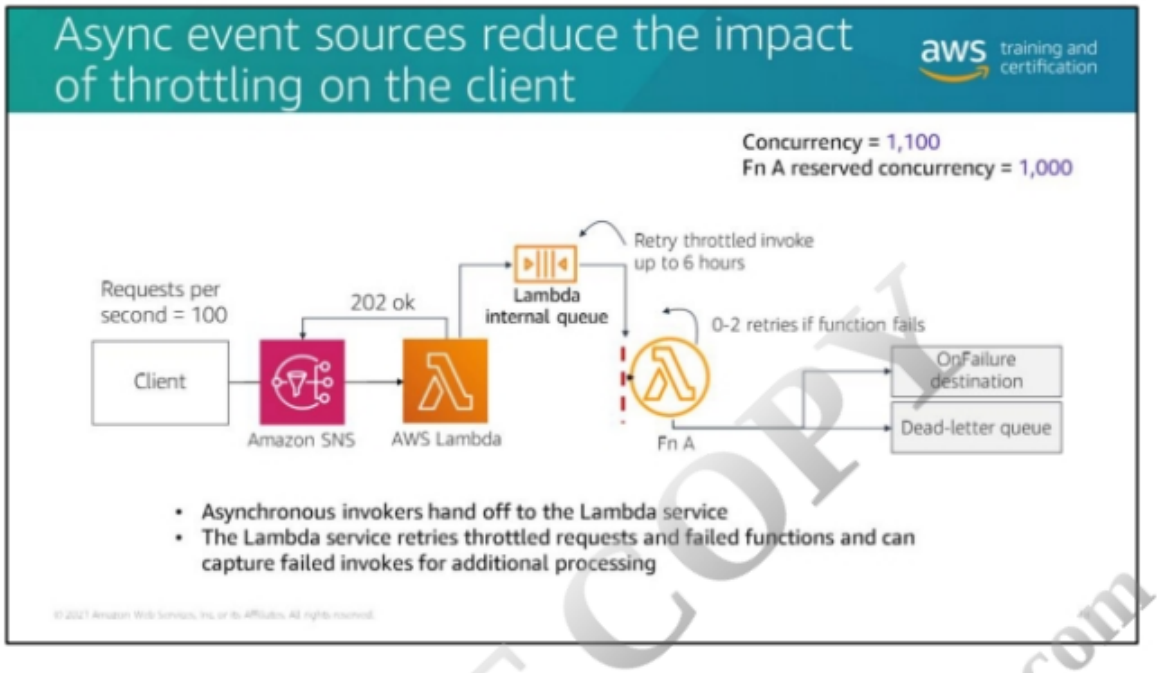
With both synchronous and asynchronous event sources, Lambda increases concurrency to keep up with demand, up to an account quota or the reserved concurrency set on a function. As highlighted in the example earlier in the module, you can determine the concurrency that you need for synchronous and asynchronous sources by multiplying the rate of incoming requests by the average duration of the function. For example, if it takes on average 11 seconds to complete a request and you are getting 100 requests per second, it will take 1,100 concurrent Lambda instances to keep up. As earlier invocations complete, new requests use the warm environment that is now available, and at that rate, no additional instances are needed.

Note that if you were able to reduce the average duration by just 1 second, you would need 100 fewer concurrent invocations to keep up, and you would stay within the set reserve. If your function hits a function reserve or gets throttled in a burst scenario, you get a 429 error indicating that the function is being throttled.

Pay attention to the metrics around invocation frequency, cold start time, and average duration as well as the spillover metric, which shows how often you are going over any provisioned concurrency. Use AWS X-Ray traces to visualize problem areas, and examine traces to analyze each segment in the event to understand the throttling behaviors.

When you think about how this plays out at a larger scale and where to set quotas, the following are some of the trade-offs to analyze:

- **Cost:** What is the cost trade-off between letting concurrency go as high as requests go versus putting a limit to control costs? Cost is also intertwined with other decisions about optimizing for your workload versus building something that can scale to any level.
- **Provisioned versus on-demand concurrency:** Based on your metrics, will configuring provisioned concurrency at a different level prevent errors?
- **Coding best practices:** The practices discussed earlier for building functions that load quickly might significantly impact the concurrency that your application uses.
- **Downstream impacts:** You might need to set a concurrency reserve to prevent Lambda from overwhelming a downstream system that can't handle the scale.
- **Synchronous versus asynchronous:** With synchronous event sources, when the incoming requests are throttled or can't complete successfully, the error goes back to the client, so regardless of the cause of the error in a synchronous event, you have to include code to handle the error. Would the introduction of an asynchronous connection with its built-in error handling be more suitable?



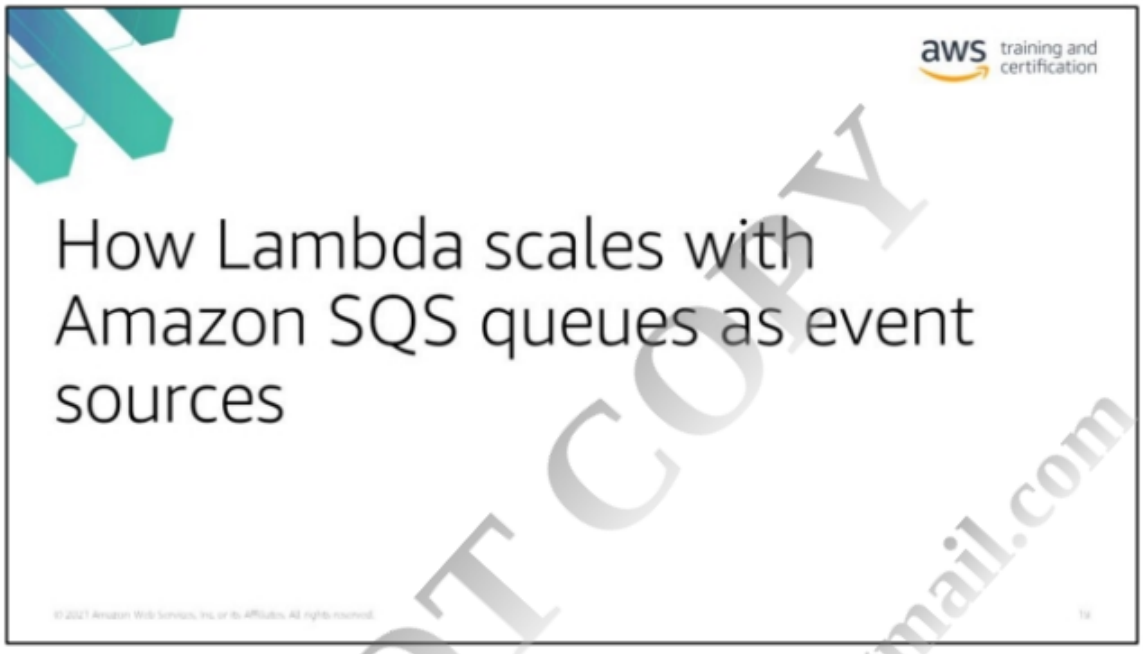
Asynchronous events also scale concurrency up to Lambda limits to keep up with the pace of requests.

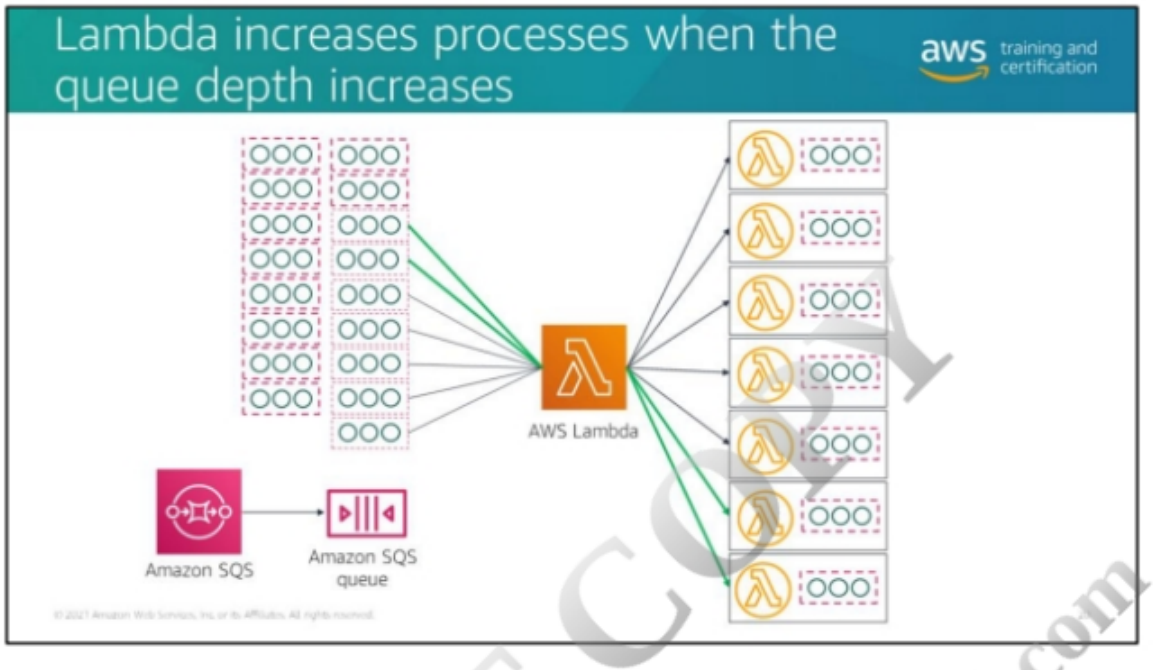
This scenario reflects the error handling that the course covered earlier. Two main points to consider:

- Remember that when Lambda accepts an event from an asynchronous source, Lambda returns a success message to the caller and then puts the request in its own internal queue for processing.
- If an asynchronous function gets throttled because no concurrency is available, Lambda retries the event for up to 6 hours using its own backoff pattern. You can configure the retry duration to be less than 6 hours.

If Lambda successfully invokes the function but you have scaling issues downstream that throw an error or prevent an invocation from successfully completing within the timeout set for the function, you can have Lambda retry invoking that function up to two more times and then have failures route to a dead-letter queue or on-failure destination for later processing.

In addition to the metrics noted earlier, pay attention to the dwell time in your X-Ray traces. This time represents how long a request sits in the Lambda internal queue before being invoked.

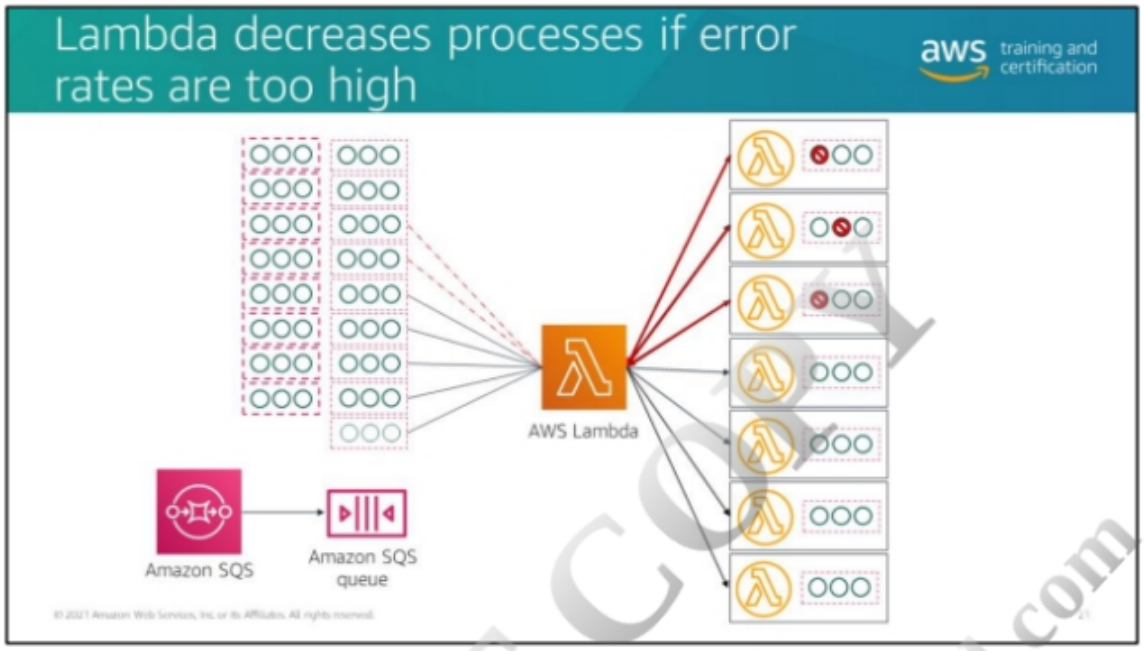




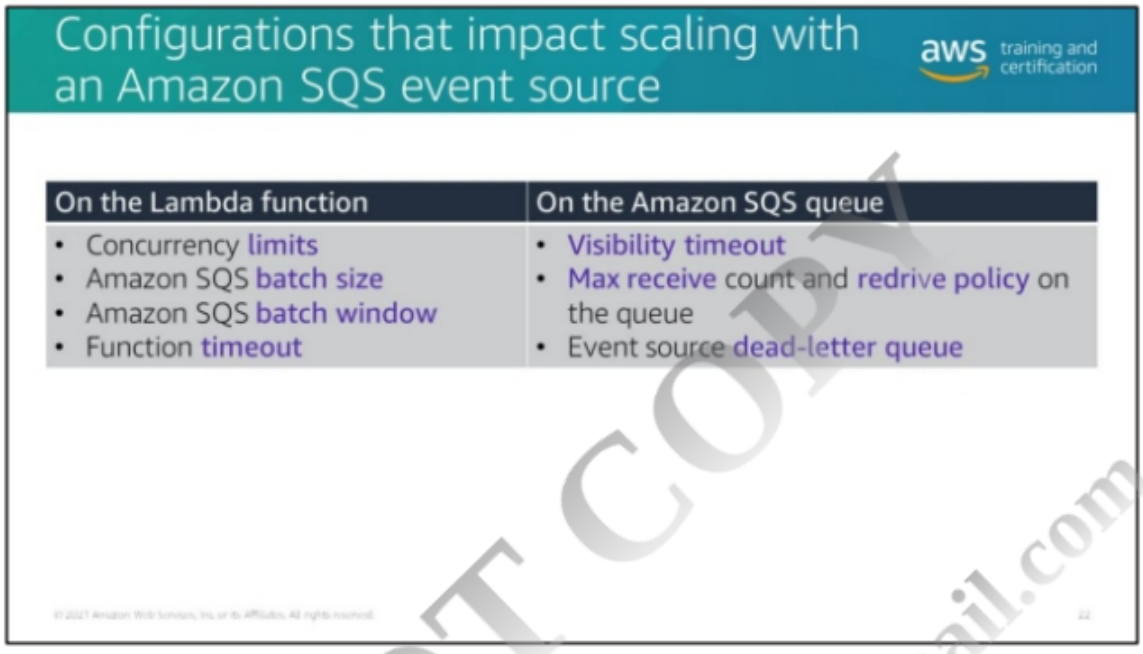
As discussed in the module on event-driven development using polling event sources, when an Amazon SQS queue is used as the event source, Lambda polls the queue in batches and synchronously initiates one invocation per batch of records.

Lambda starts with five polling processes, which invoke five concurrent instances of your function. To avoid your Lambda function getting throttled from the start, make sure that the reserved concurrency on the function is at least five. If the Lambda service detects an increase in queue size, it automatically increases how many batches it gets from the queue each time. That means Lambda will increase the number of concurrent Lambda functions that it invokes. Lambda continues to add additional processes each minute until the queue has slowed down or it reaches maximum concurrency. Maximum concurrency is 1,000 unless the account or function quota is lower.

When Lambda pulls batches of records, those records become invisible to other consumers of the queue (other Lambda invocations) for the duration of the visibility timeout. This is a setting that you control.



However, if the Lambda service is getting an increase in errors being returned from the function, Lambda assumes that this indicates downstream back pressure and decreases the polling rate in response.



As the previous slides indicated, the Lambda service takes on much of the scaling work when Amazon SQS is used as an event source, but you control options to tune for your application. In addition to the concurrency settings that you just looked at, you should tune the batch size and timeout on the function to match the scale that you need.

You might also want to adjust the batch window to avoid invoking Lambda with a smaller number of records. By default, Lambda invokes your function as soon as records are available in the SQS queue. Use the batch window to tell the event source to buffer records for up to 5 minutes. With the batch window configured, Lambda continues to poll messages from the queue until the batch window expires, payload limit is reached, or full batch size is reached.

These settings work in conjunction with the visibility timeout, max receive count, and redrive policy that you set on the queue.

On the queue, you need to configure the visibility timeout to allow enough time for your Lambda function to complete a message batch and delete its messages off of the queue before they become visible again and another function invocation picks them up.

If you stick with the example of 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, but you don't want to use a visibility timeout that close to your average processing time. You need to leave some buffer in the visibility timeout to account for Lambda invocation retries when the function is getting throttled. The best practice is to set your visibility timeout to at least six times the timeout that you configure for your function.