Metric alarm states tell you if there is an alarm condition, not enough data to evaluate the condition, or that everything is OK.

Many potential metrics might make sense for alarms. The following are a few examples:

API Gateway

5XXError – This metric is the number of requests with server-side errors due to exceptions downstream or gateway service issues. This is a metric that you should react to quickly.

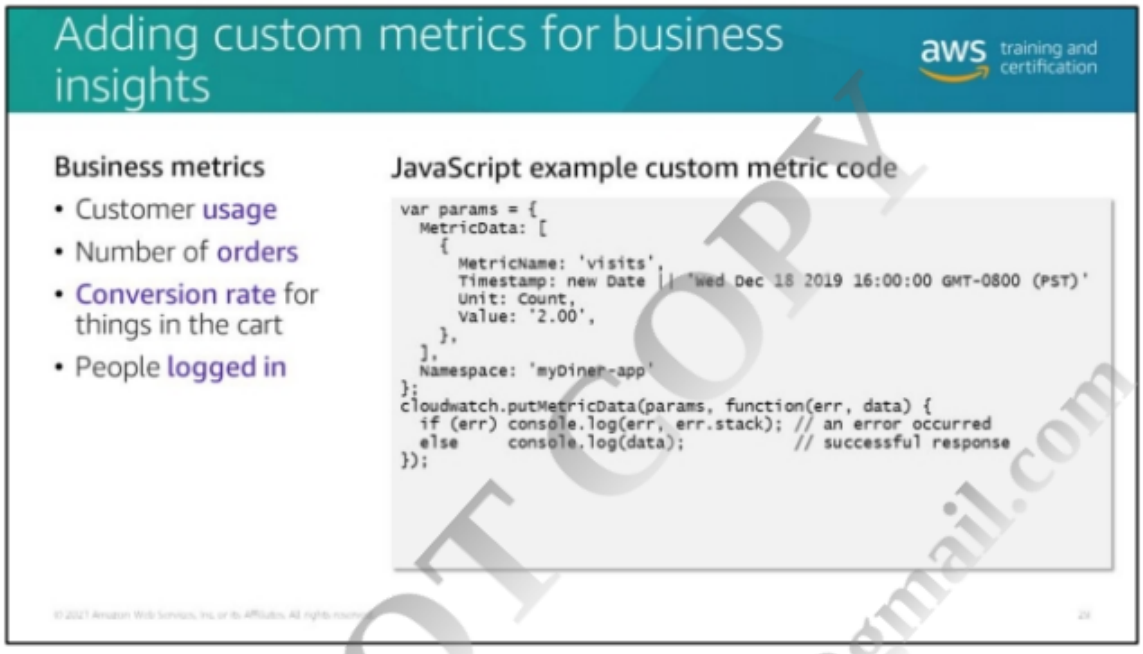
Lambda

Errors – This metric is the number of failed function invocations due to timeout, exceptions, or service issues. You might not want to alarm on all of your functions because there is a cost to alarms, but for critical functions, this alarm might be important.

IteratorAge – This metric is for stream-based invocations only. It measures the age of the last record for each batch of records processed, and an increasing value implies a stalled shard on the stream.

For metrics that you want to watch but may not need to alarm on, you can create dashboards. For example, concurrency invocations and unreserved concurrent invocations are good metrics to track or review, but you probably don't need an alarm on them.

Links are also available in the OCS.



You can add **custom metrics** via your application code to get the type of business metrics that you learned about earlier. For example, you can add metrics for how many people are using the application and in what access patterns, how many orders you are receiving, and what the conversion rate is for people putting things in their cart and actually completing the transaction.

For more information about custom metrics, visit https://awscli.amazonaws.com/v2/documentation/api/latest/reference/cloudwatch/put-metric-data.html.

Use the SDK for your runtime to write your code.

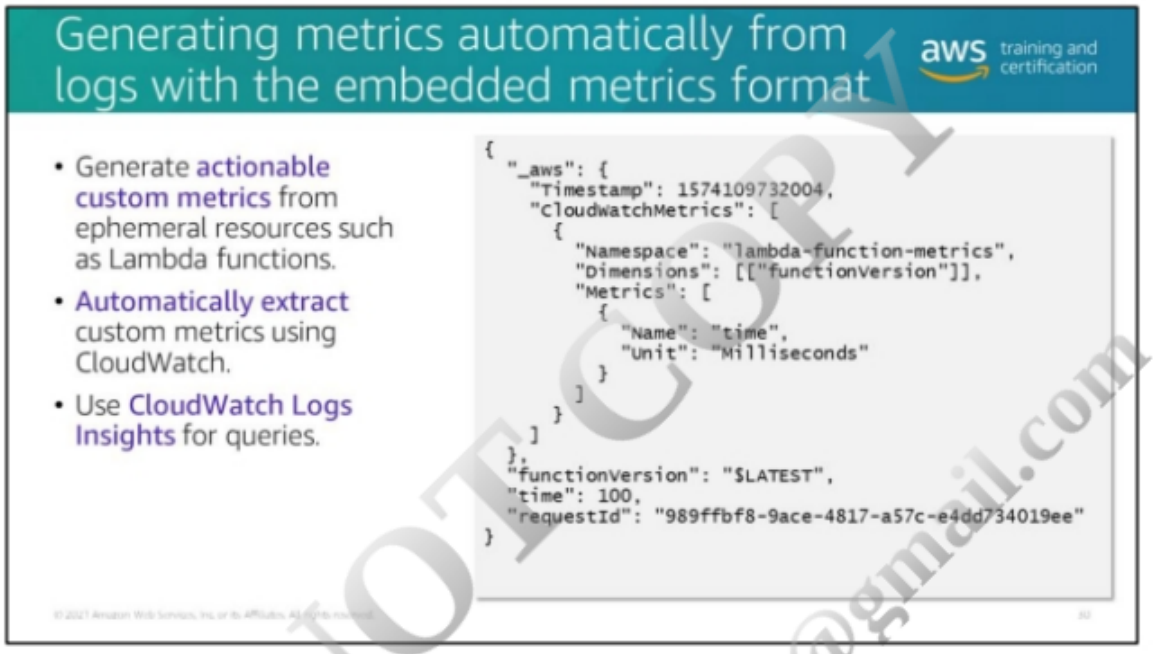
For more information about the **putMetricData**(params = {}, callback) metric, visit https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/CloudWatch.html#putMetricData-property (node).

For more information about the **put_data()** metric, visit https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/cloudwatch.html#CloudWatch.Metric.put_data.

You incur charges for each API call, so when doing this at scale, it can add a lot to your cost.

The call to the API is a synchronous call, so the function waits for it to complete. Even though this happens pretty quickly, you're adding to the time it takes to run your function. The time it takes to call the putMetricData API may be longer than the whole function would take otherwise, so writing the metrics costs more than running the function.

As an alternative, you can use the embedded metrics format to add the metrics information to your logs and let CloudWatch pull them out as metrics that you can act on.



For information about ingesting high-cardinality logs and generating metrics with the CloudWatch embedded metric format, visit https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/CloudWatch_Embedded_Metric_Format.html.

For information about using the client libraries to generate embedded metric format logs, visit https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/CloudWatch_Embedded_Metric_Format_Libraries.html.

For information about the embedded metric format specification, visit https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/CloudWatch_Embedded_Metric_Format_Specification.html.

When you use the embedded metric format, CloudWatch looks at the log stream and picks out the metrics. Then you can use them in alarms and dashboards. You then also have the data available in your logs for running queries from CloudWatch Logs Insights about those metric fields.

Use the TIO environment to navigate through the ServiceLens dashboard and review the combination of logs, metrics, and traces available as your instructor demonstrates them. Select the CloudWatch service from the AWS Management Console to get started.

Steps are listed in the Lab Guide. Slides marked with the map icon used on this slide are associated with steps in the exercise.

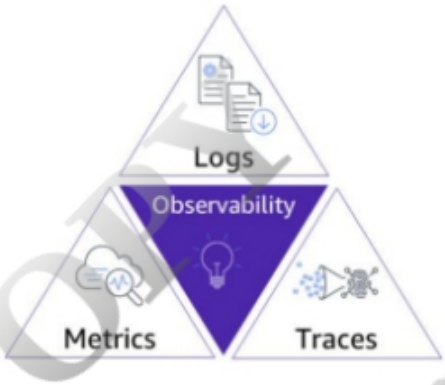
Bringing the three pillars together in CloudWatch ServiceLens

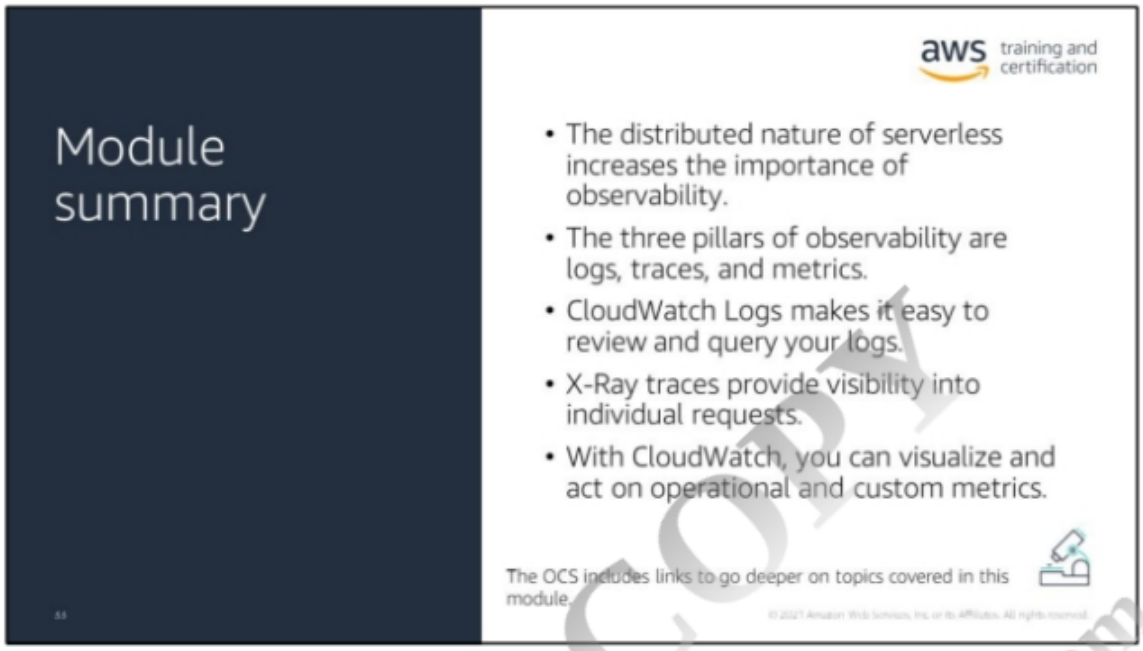
aws training and certification

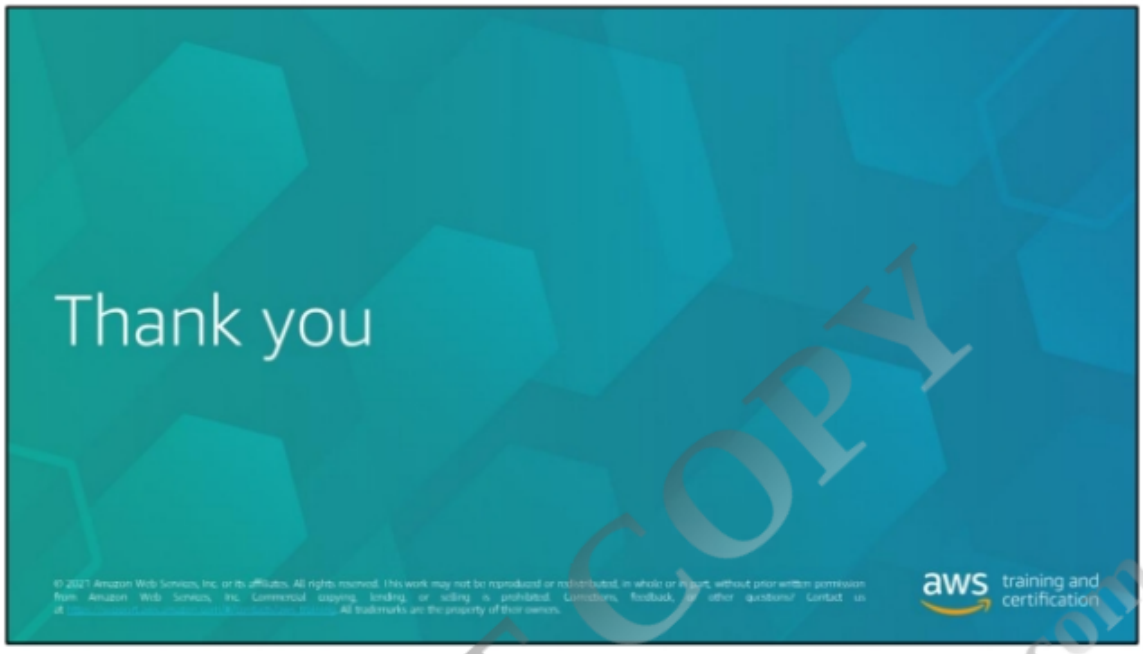
Review traces, logs, and metrics from an integrated view

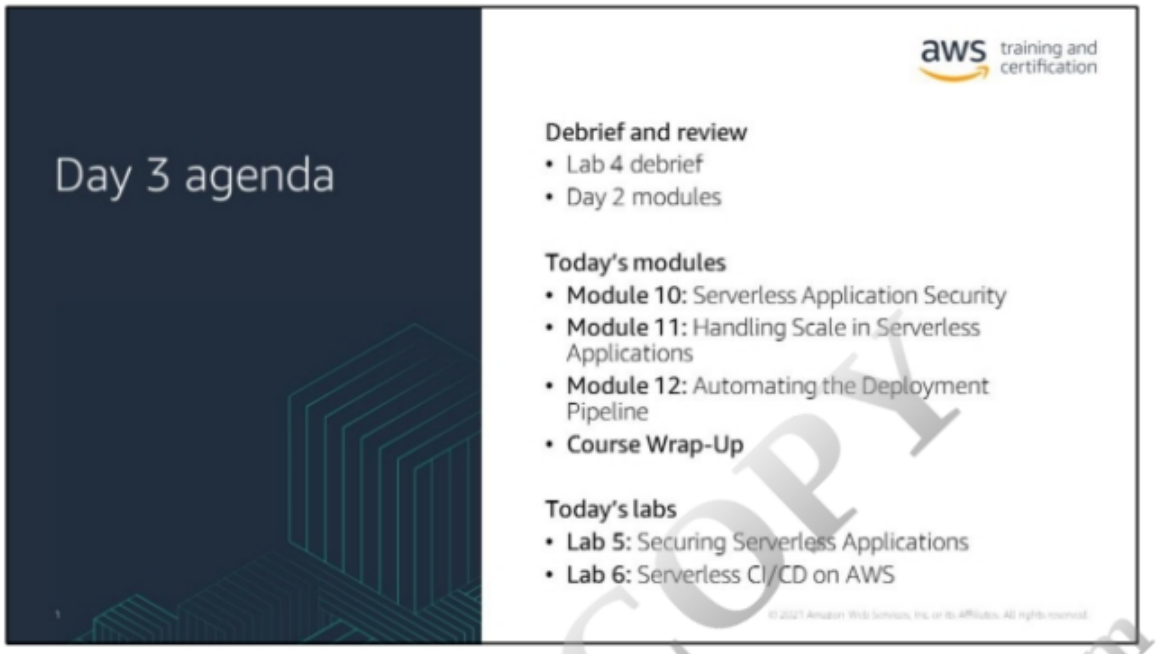
Developer features

- Integrates X-Ray and CloudWatch
- Includes a service map with integrated access to logs, metrics, and alarms
- Provides a one-stop shop for troubleshooting





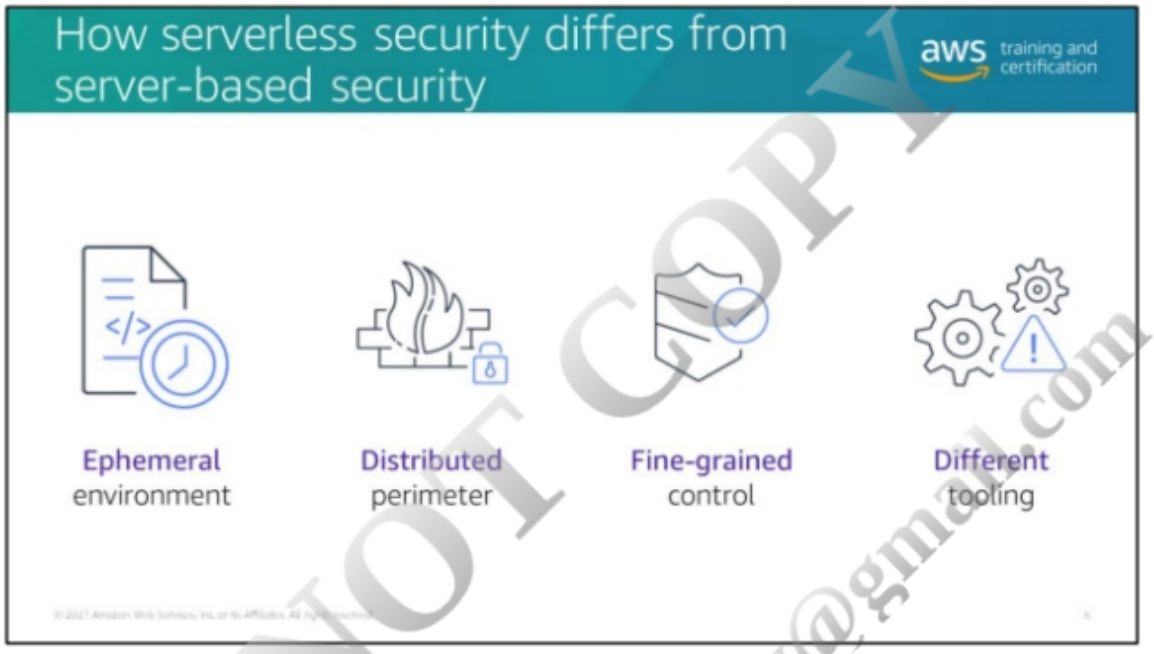




Security is a critical topic, and there is an entire 3-day course on security engineering with AWS. This module is designed to help you think about how to apply security best practices in a serverless application and point you to the types of resources and services you will want to investigate further.

Security best practices don't change with serverless. You still focus on:

- Following the principle of least privilege
- Securing data in transit and at rest
- Writing code that protects the data and credentials it uses
- Incorporating monitoring and auditing that alerts and potentially deflects security concerns before they occur



The OWASP Top 10 is a standard awareness document for developers and web application security. The document represents a broad consensus about the most critical security risks to web applications. For more information about the OWASP Top 10, visit https://owasp.org/www-project-top-ten/.

Your approach to application security should address mitigation of each of these risks. These risks are not unique to serverless, but you need to be aware of how differences in the serverless environment may change how you implement security in your application. You need to think a little differently with serverless in terms of how you apply those best practices.

First, the ephemeral environment in which AWS Lambda runs means that attackers who might gain a foothold within that container won't get the type of access that they might in a more traditional environment. Of course, this environment also means that the logging and metrics you design are more critical. Once the code runs, you can no longer go back to the server to find out what happened.

Second, the security perimeter you are defending has to consider the different services that might initiate a function, and your code needs to defend each of those potential paths.

Third, you can use Lambda's fine-grained controls to scope its reach with a much smaller set of permissions as opposed to traditional approaches where you may give broad permissions for your application on its servers. Scope your functions to limit permissions sharing between any unrelated components.

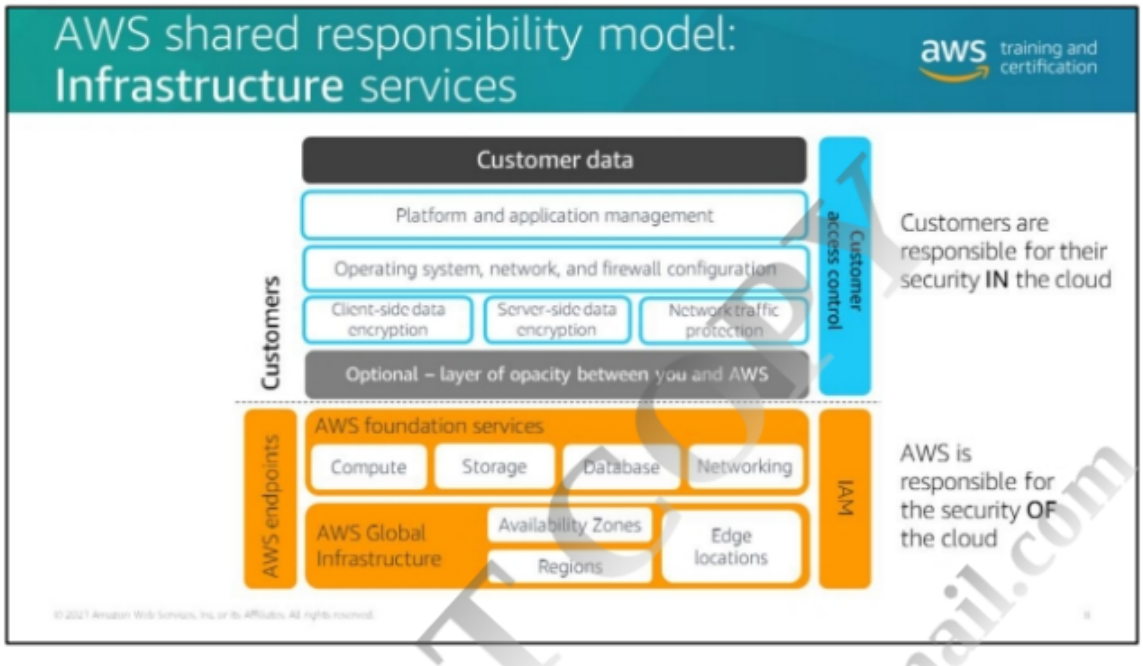
Finally, the tooling is different. For example, techniques such as installing agents on your host may not be relevant any more.

You can apply cloud security best practices through these security design principles. These principles are not unique to serverless, but in this module, you will look at examples of how they apply and how to implement them in your serverless applications.

Visit the Security Overview of Amazon API Gateway whitepaper as a reference for these principles: https://d1.awsstatic.com/whitepapers/api-gateway-security.pdf

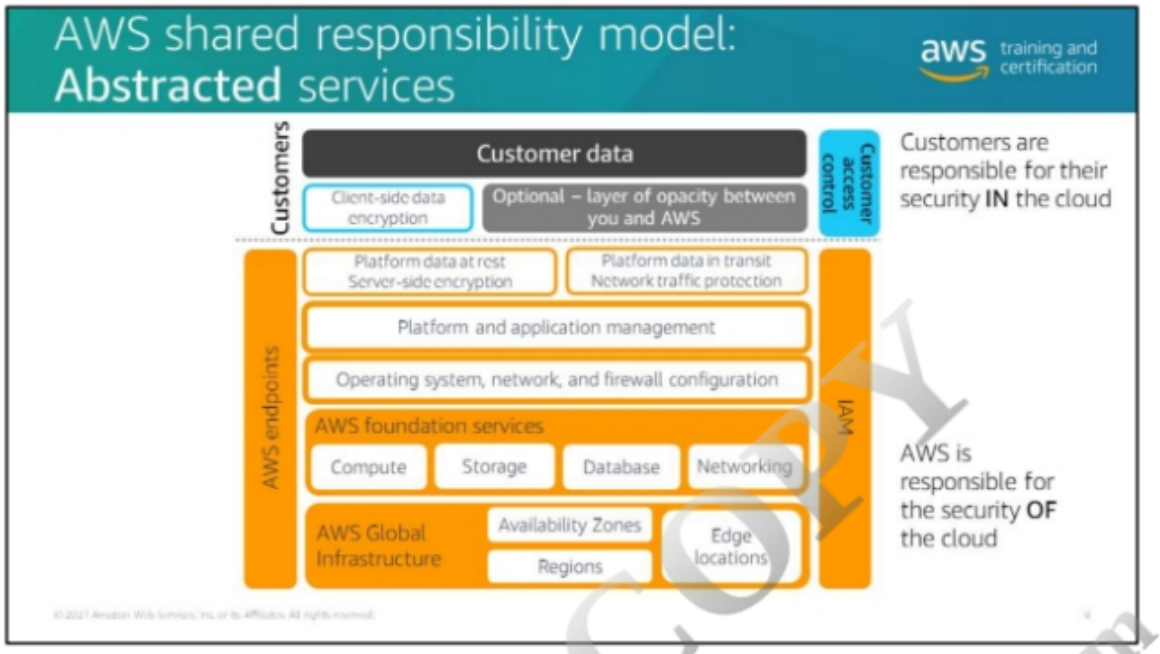
Applying security at all layers:

- Understand how the AWS shared responsibility model is applied in serverless applications
- Apply a defense in depth approach with multiple security controls
- Apply to all layers (for example, edge of network, VPC, load balancing, every instance and compute service, operating system, application, and code)



A benefit of serverless architectures is that when you leverage AWS managed services, you shift the burden of the shared responsibility model toward AWS.

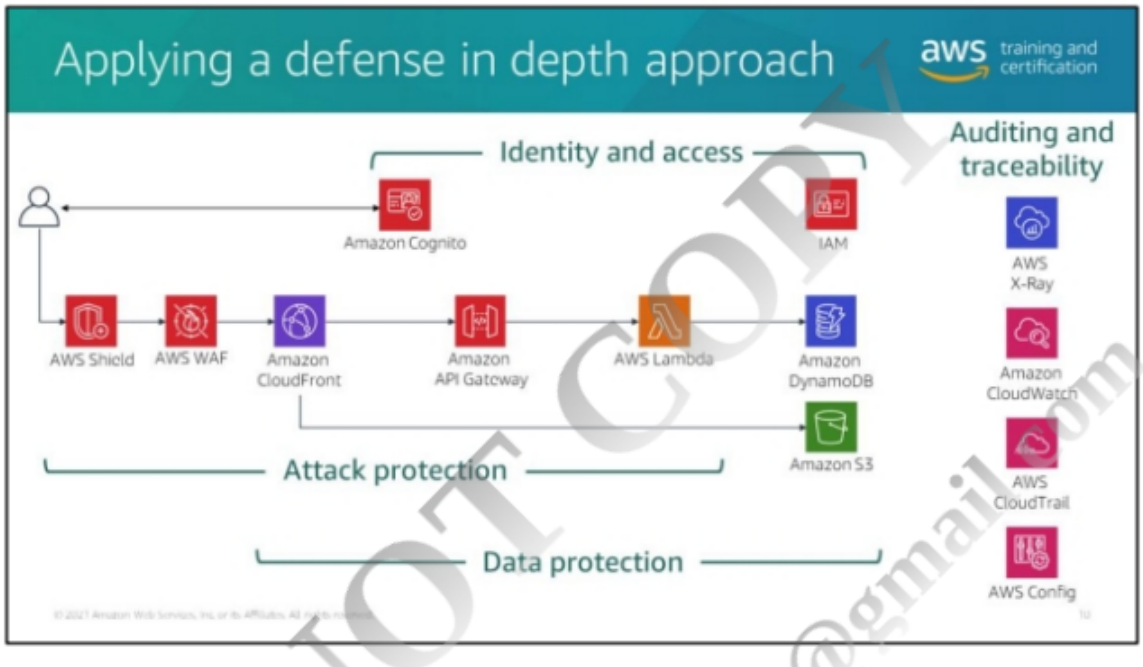
With infrastructure services like Amazon Elastic Compute Cloud (Amazon EC2), you are still responsible for your operating system, network, and firewall configuration, and the protection of network traffic and both client-side and server-side data encryption.



When you are using abstracted or managed services, AWS takes on the burden of some of those tasks.

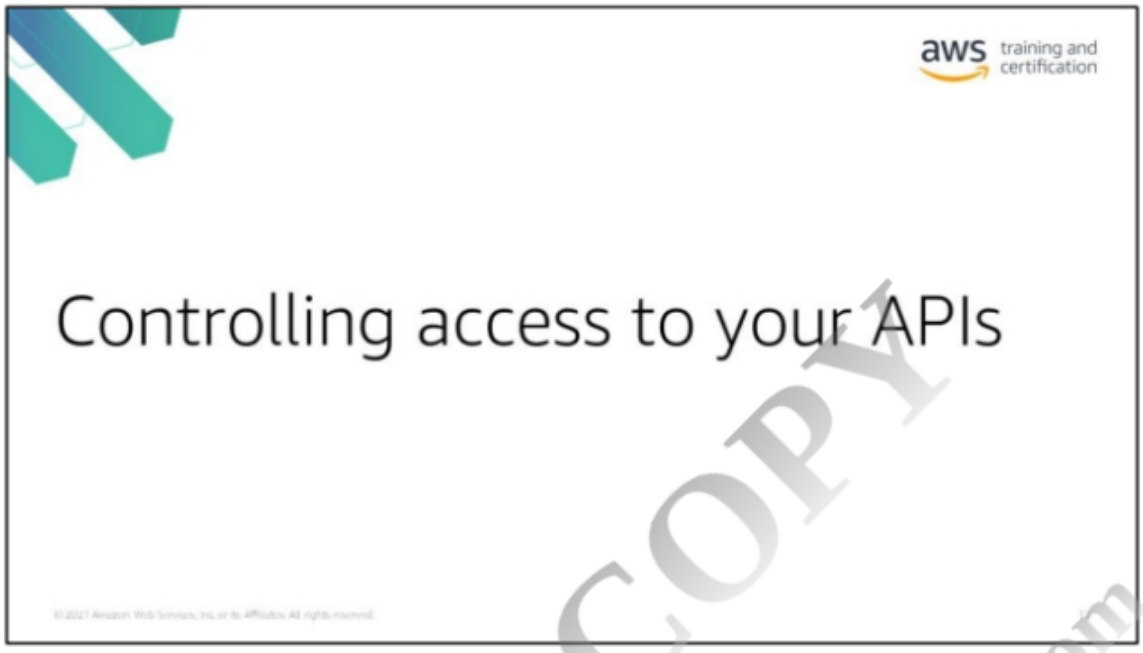
You aren't responsible for the operating system or network configuration where your functions run, and AWS ensures the security of the data within those managed services. You are responsible for protecting data entering your application and limiting access to your AWS resources. You still need to protect data that originates on the client side or that travels to or from endpoints outside of AWS.

You aren't responsible for the operating system or the network itself, but you do need to protect your network boundaries and mitigate external attacks.



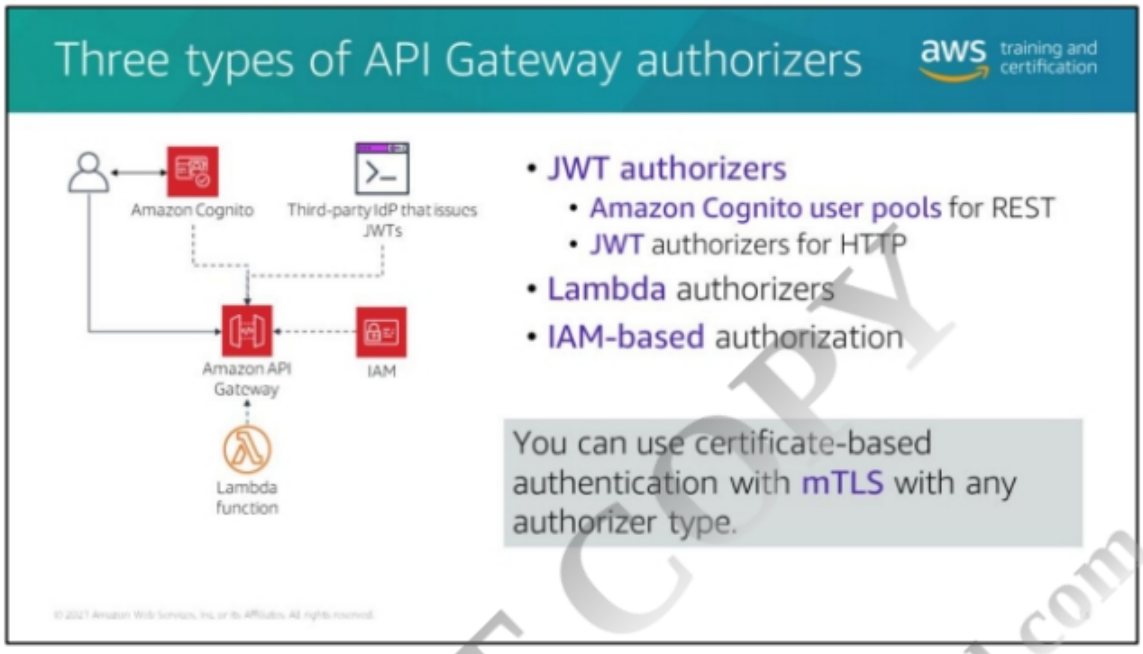
A key principle is that your security strategy should address security at all layers. Use a defense in depth approach that includes multiple security controls. Apply these security design principles at network boundaries, at your application's front door, between its components, and within your code.

During this course, you focus primarily on API security and best practices for your Lambda functions because those stand out as the biggest differentiators in applying security to a serverless application. Use the resources listed in the dive deeper section to go deeper into applying security best practices that apply to your use cases.



The first step to preventing unwanted access is implementing Amazon Cognito or another third-party provider for authentication in front of your APIs. Additionally, you need to secure your APIs to prevent unauthorized users from interacting with them.

AWS Identity and Access Management (IAM) plays an important role in controlling access and authorization among your components. IAM administrators control who can be authenticated (logged in) and authorized (have permissions) to use Amazon API Gateway resources. API Gateway integrates with IAM for a number of purposes.



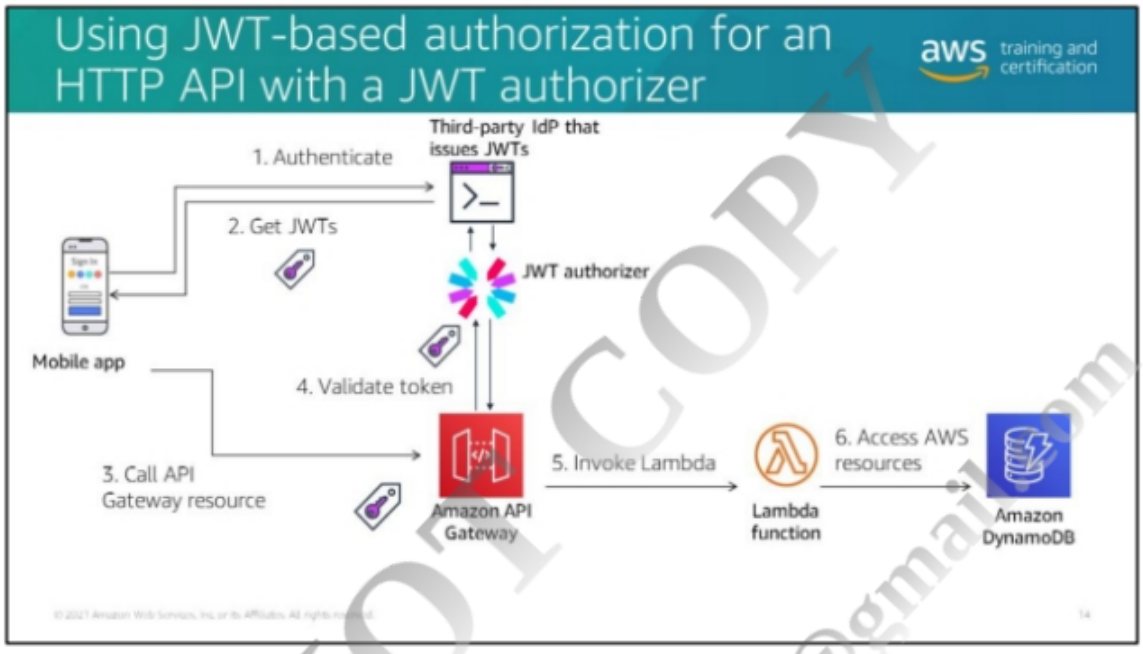
The first step to preventing unwanted access is implementing **authentication** in front of your APIs. Additionally, you need to secure your APIs to prevent **unauthorized** users from interacting with them.

As introduced earlier in the course, you can use three general types of authorizers with API Gateway:

- Amazon Cognito user pools and third-party JSON Web Token (JWT) authorizers
- Lambda authorizers, which allow you to write Lambda functions that perform custom authorization
- IAM based authorization where IAM policies provide permissions to AWS resources

API Gateway also supports certificate-based authentication via mutual Transport Layer Security (mTLS). API Gateway provides integrated mTLS authentication without the additional cost or operational overhead that was previously required to manage and scale a reverse proxy fleet for terminating mTLS connections. You can enable mTLS authentication on your custom domain(s) to authenticate Regional REST and HTTP APIs while still authorizing requests with bearer or JWTs or signing requests with IAM based authorization.

All that is required is uploading a trusted certificate authority (CA) public key certificate bundle to an Amazon Simple Storage Service (Amazon S3) bucket as an object containing public or private/self-signed CA certs to be used for validation of issuance of client certificates. All existing authorization options are available for use in conjunction with mTLS while offering additional request context on the calling user's certificate and identity for granular authorization decisions.



Earlier in the course, you learned that Amazon Cognito uses OpenID Connect and OAuth 2.0 authentication and that Amazon Cognito authorizers use three JWTs:

- ID
- Access
- Refresh

As a quick review, developers pass the ID and access tokens along to authorize access to application resources. Those tokens live for a relatively short period of time. The client must save the refresh token (which can live for up to 1 year) to silently refresh the ID and access tokens behind the scenes. These tokens simplify identity for developers and allow them to pass user context to downstream services in a lightweight manner. This makes it easy for developers to get this user context.

The process is effectively the same whether you are using a third-party identity provider (IdP) or Amazon Cognito to generate the JWT tokens. If you use a JWT authorizer with an HTTP API, the JWTs are issued outside of AWS.

When you use an Amazon Cognito user pool directory as your IdP, AWS hosts the entire experience. If you set up Amazon Cognito user pools to federate with a third-party IdP, Amazon Cognito user pools is trusting those identities and redirecting them as if the users were in its directory.

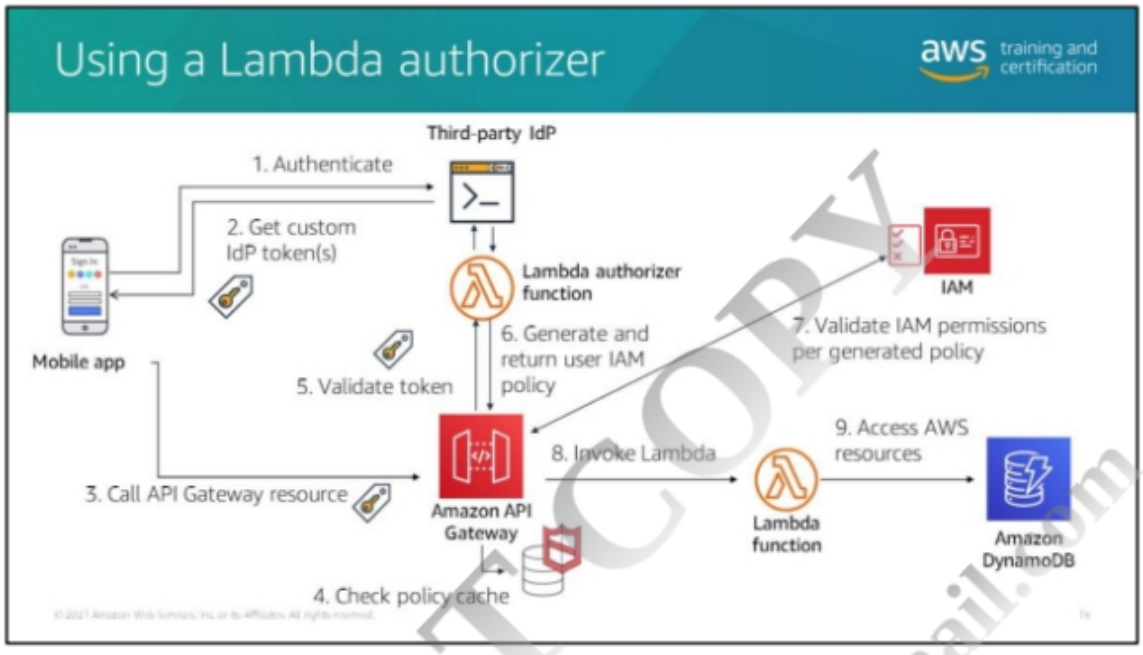
This diagram depicts an authorization flow for an HTTP API that uses JWT authorizer user pools:

1. The client first authenticates with the IdP.
2. The clients gets the JWTs in return.
3. The client then passes the ID and access token in the header as part of the call to API Gateway.
4. API Gateway validates the token.
5. API Gateway invokes the resource with which it integrates on the backend (in this example, a Lambda function).
6. Lambda accesses other resources within AWS.

Depending on how you have written your API and application, API Gateway may pass on the ID token or the access token.

If you don't need to further scope the access allowed, pass the ID token. If you need to scope access further, you can use the access token and configure predefined attributes within the API method request using OAuth scopes.

For more information about configuring these options, visit https://docs.aws.amazon.com/apigateway/latest/developerguide/apigateway-enable-cognito-user-pool.html.

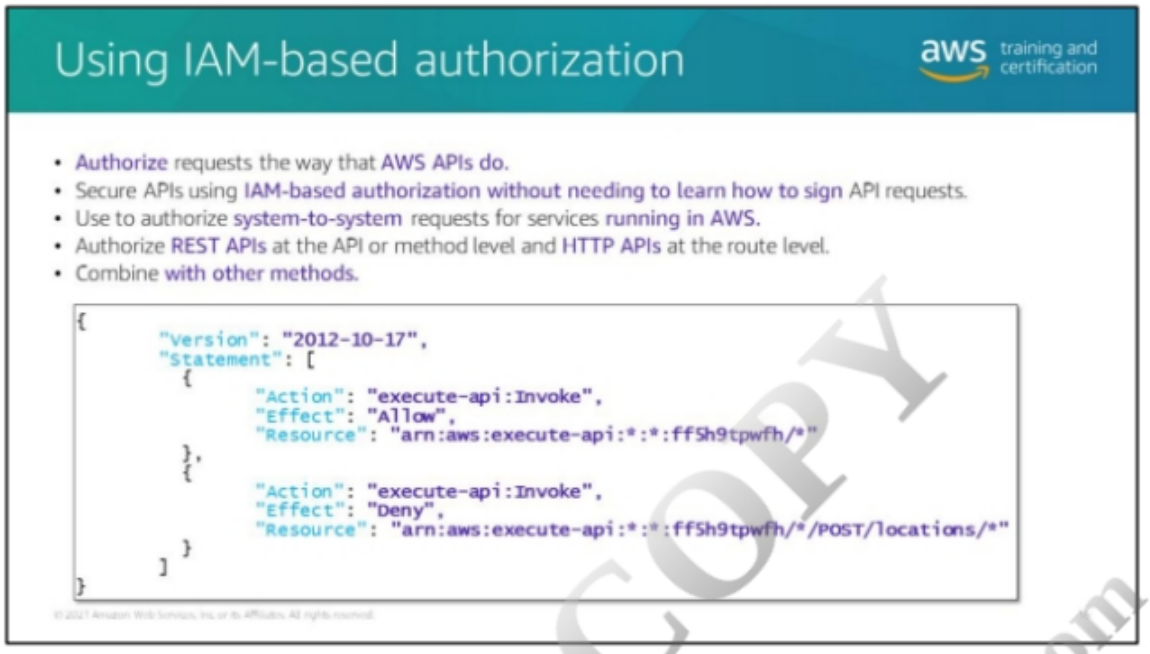


When you use a Lambda authorizer, your client authenticates with a third-party IdP and gets custom tokens back. Lambda authorizers are available for both REST and HTTP APIs.

HTTP APIs can use a simplified authorizer payload format as compared to the version that is required for REST APIs. For more information, visit https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-lambda-authorizer.html.

In this flow:

1. Your client authenticates with a third-party IdP.
2. The client gets custom tokens back.
3. As with the previous example, the token information is passed in the header to API Gateway.
4. API Gateway checks its policy cache to see if this requestor has already been recently authorized. If it has, API Gateway can skip the steps of validating the token and returning the IAM policy.
5. If the requestor's policy is not cached, API Gateway invokes the Lambda authorizer function.
6. The function includes the code to emit the IAM policy and return it to API Gateway.
7. API Gateway uses the IAM policy (whether cached or newly generated) to validate the permissions for the requestor.
8. API Gateway invokes the resource with which it integrates on the backend (in this example, a Lambda function).
9. Lambda accesses other resources within AWS.



With **IAM based authorization,** you can enable your service to authorize requests in the same manner that all AWS APIs do, which is to validate a unique canonical request signature that the API client generates and sends with each request. The signature is uniquely generated and incorporates the time of request, resource requested, and action so that even if the signature were compromised and re-used later on, the request signature would no longer be valid at a later time.

This is the most secure authorizer option but requires that the API clients have intelligence on how to sign their requests. For API access between systems running in AWS, use IAM. IAM is great for system-to-system authorization for services running in AWS because it is the most secure. However, IAM is the most inconvenient if you need to write the code yourself. Using an SDK with request signing built in is advisable if choosing IAM based authorization.

https://docs.aws.amazon.com/general/latest/gr/signing_aws_api_requests.html

With IAM based authorization:

- The requestor must have IAM credentials.
- Key information is added to the authorization header.

- API Gateway determines whether or not the IAM user or role has permissions to invoke the API.

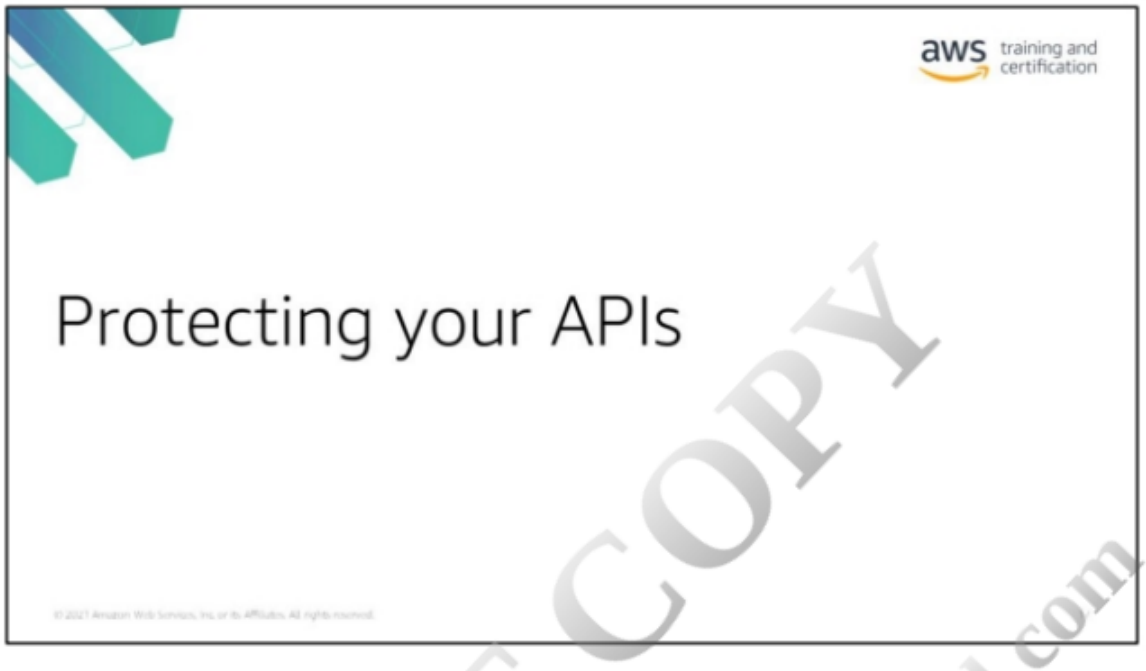
You can use IAM based authorization at the API level for REST APIs using a *resource policy* on the API. You can also use IAM based authorization at the method or route level on your REST and HTTP APIs.

IAM integrates with API Gateway to give you the ability to limit the scope of allowed actions for an API method or route to include whether or not it can invoke or manage the API. Integration with IAM allows you to put very granular permissions on API actions. You can use these IAM based authorizations in conjunction with the other types of authorizers that this course has just discussed. For more information about how resource policies work in conjunction with authorization methods, visit https://docs.aws.amazon.com/apigateway/latest/developerguide/apigateway-authorization-flow.html.

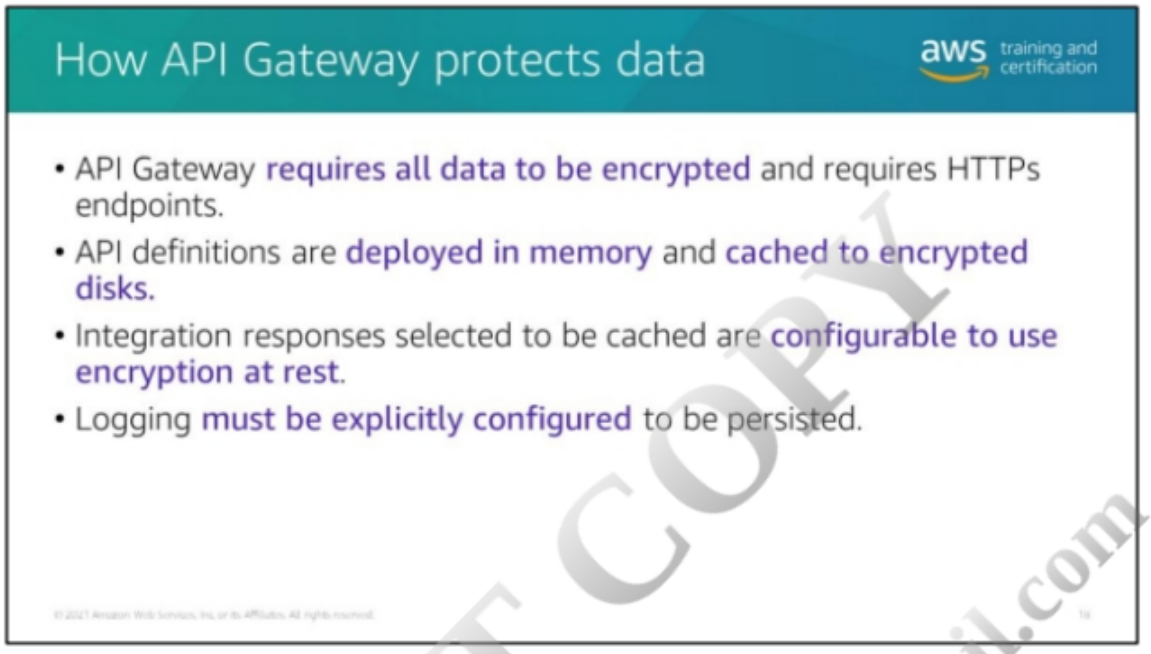
The example IAM resource policy allows the role with which it is associated to perform the invoke action on an API but denies access to particular methods of the API.

API Gateway:
- Supports using an abbreviated syntax to refer to an API resource rather than using the full Amazon Resource Name (ARN)
- Converts the abbreviated syntax to the full ARN when you save the policy. For example, you can specify the resource **execute-api:/stage-name/GET/pets**.
- Converts the resource to **arn:aws:execute-api:us-east-2:123456789012:aabbccddee/stage-name/GET/pets**

For more example resource policies, visit https://docs.aws.amazon.com/apigateway/latest/developerguide/apigateway-resource-policies-examples.html.



API Gateway provides options for creating private endpoints that are reachable from only within your Amazon Virtual Private Cloud (Amazon VPC). API Gateway can also provide backend integration to resources within your VPC via a VPC link. Let's look at both of these scenarios.



API Gateway communicates over HTTPS, but you should encrypt the payload on the client side because elements like the request path and query strings that are part of a URL might not be encrypted. For example, if you write logs using standard output, you could accidentally expose unencrypted sensitive data. API Gateway communicates with an integration endpoint regardless of whether it uses HTTP or HTTPS. To maintain end-to-end encryption of data in transit, you should encypt sensitive data before any processing or data manipulation.

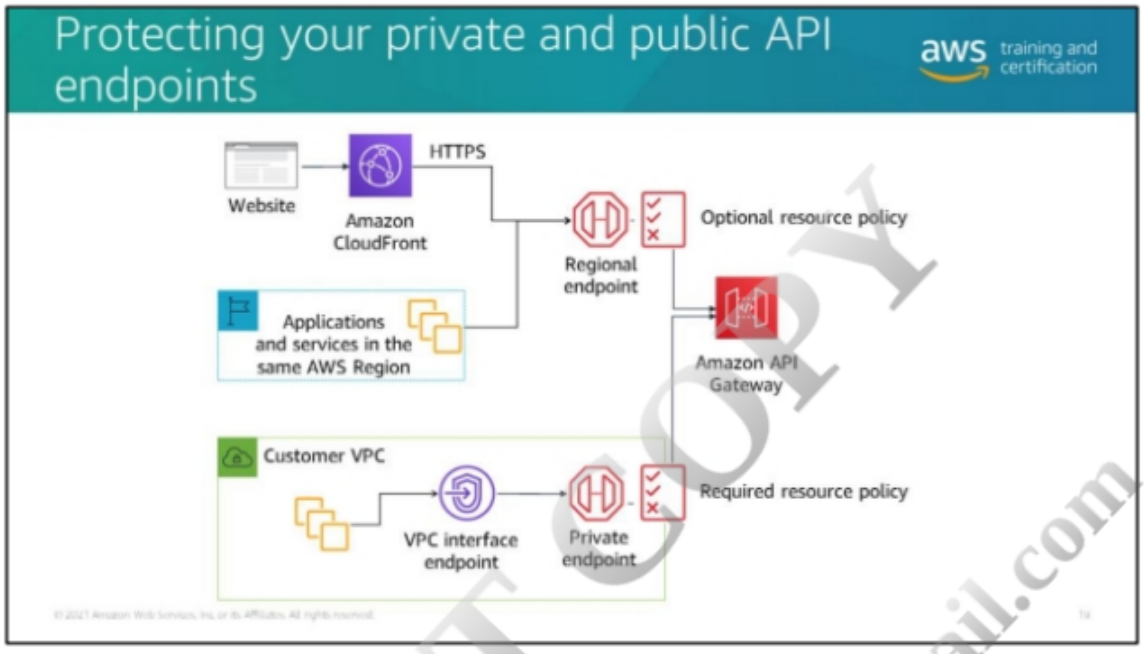
Do not send, log, or store unencrypted sensitive data, whether it's part of an HTTP request path/query string or standard output of a Lambda function. Although you aren't responsible for the Lambda infrastructure, you are responsible for the inputs and outputs. Encrypt what you write.

API Gateway requires that all data sent to both control plane and data plane operations be encrypted in transit using TLS and requires the use of HTTPS endpoints.

Unencrypted API Gateway endpoints are not supported. Additionally, API developers can choose to require a specific TLS version for their custom domain names if desired. You can configure mTLS using certificate-based authentication on a custom domain name.

All API definitions are deployed in memory and are cached only to encrypted disks. Customer log files are temporarily stored in encrypted form before being sent securely to Amazon CloudWatch Logs or Amazon Kinesis, which stores the logs encrypted at rest.

All integration responses selected to be cached with API Gateway are persisted on cache nodes and are configurable to use encryption at rest while stored. Logging is not configured or persisted by default unless explicitly configured.



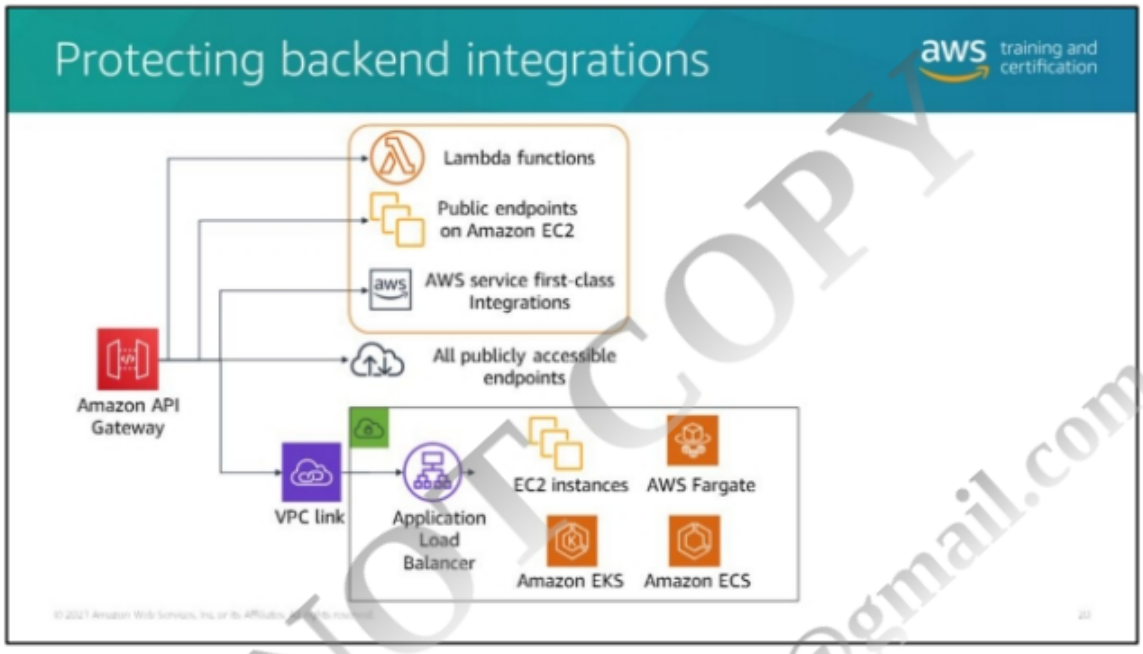
Let's go back to the API Gateway flow that the course discussed earlier. This time, let's focus on the private endpoint option where traffic is originating within your VPC. A private endpoint can be reached from only a VPC interface endpoint, and traffic to and from the private endpoint stays within your VPC.

API Gateway resource policies are required on the API prior to deploying it. Resource policies on endpoints for private APIs allow you to control whether instances and services in VPCs and VPC endpoints can invoke your API in addition to all the same controls that are offered for public endpoints. This also insulates the endpoints from public DDOS attacks because they are not exposed to the internet.

Sample use cases that you can implement via resource policies include the following:

- Restricting calls to production API Gateway deployments to only services in production VPCs
- Restricting calls to preproduction API Gateway deployments to services with an assumed role

If you must use public endpoints, use optional resource policies to help improve your security posture and reduce the possibility of an impact to your service via configuration. For example, restrict access to users from a specified AWS account or deny traffic from a specified source IP address or CIDR block.



You have looked at how private endpoints and resource policies on public endpoints protect access to your APIs.

API Gateway also offers security for API integrations to backend resources. API integrations allow you to invoke applications, functions, or services to respond to API requests. These security mechanisms allow API Gateway to securely integrate and access AWS services and other HTTP endpoints to respond to requests to your API. The IAM permissions policies assigned to the backend service determine which resources the backend service can or cannot access.

Lambda integrations
Lambda integrations allow the mapping of a single resource or method on your API to a Lambda function. This integration works directly with the Lambda service endpoints. A Lambda function resource policy is used to allow only API Gateway to invoke the specified Lambda function to respond to an API request.

HTTP integrations with public endpoints
HTTP integrations allow you to integrate your API with HTTP/S services with public endpoints. You can create API Gateway generated client certificates to secure requests made to HTTP endpoints, which can be used to verify the requester's authenticity.

AWS service first-class integrations

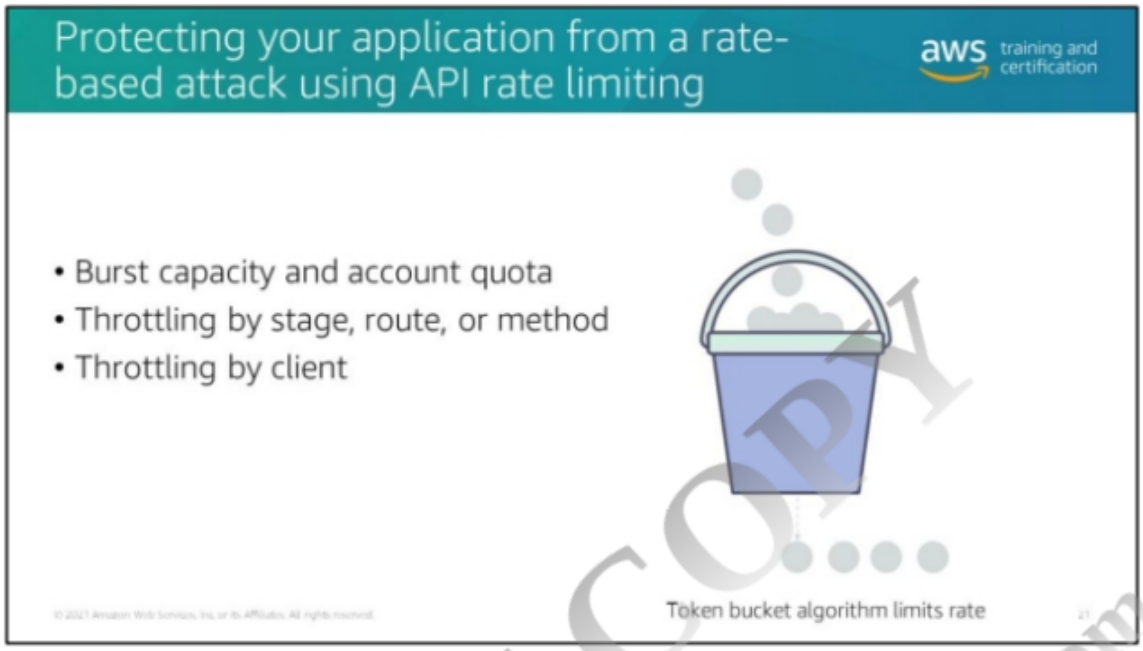
AWS service first-class integrations allow you to directly integrate your API with AWS services, such as Amazon Kinesis Data Streams or Amazon Simple Queue Service (Amazon SQS). This integration requires the creation of an IAM execution role with a trust policy, where API Gateway is the principal. You also need to create a permissions policy to allow the action required for the AWS service you are integrating with.

HTTP integrations with VPC resources

On the backend, if your API integration needs to access resources within a VPC, you can use a VPC link to connect to an Application Load Balancer or Network Load Balancer within the VPC for REST and HTTP APIs. HTTP APIs can also connect to AWS Cloud Map.

In this example, a VPC link is used to connect to an Application Load Balancer in a VPC, which is connected to the Amazon Elastic Kubernetes Service (Amazon EKS).

Note that you can use a VPC link on backend integrations regardless of whether the request is coming via a public or private API Gateway endpoint.

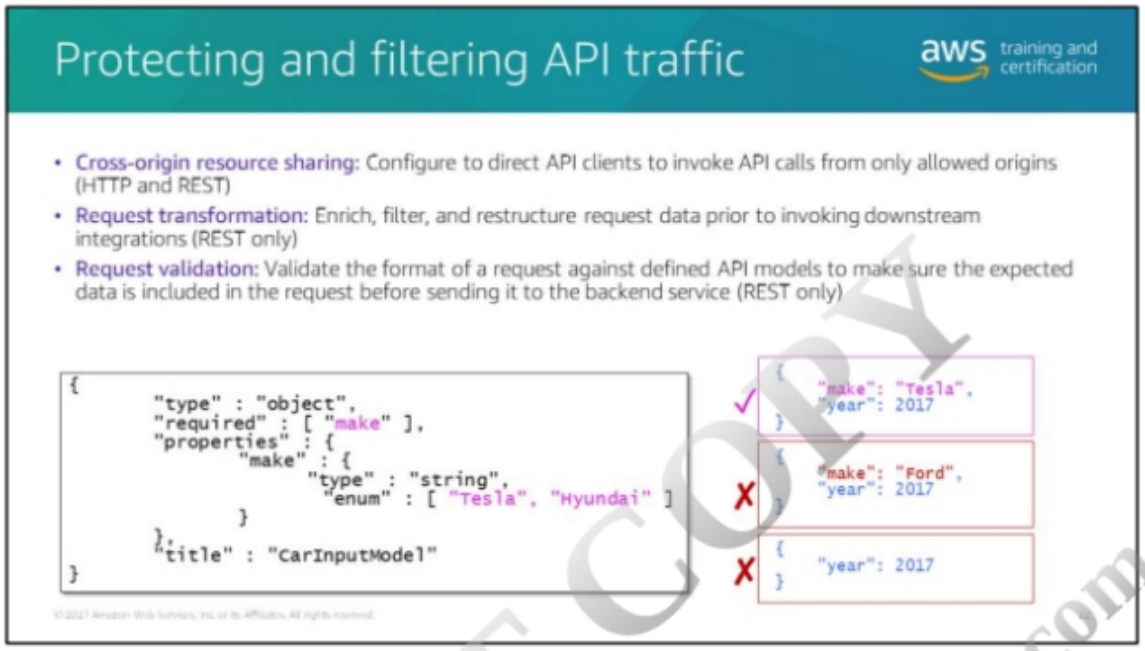


For information about API Gateway request throttling, visit https://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-request-throttling.html.

The scaling module talks a bit more about API Gateway throttling, but it also applies to securing your APIs. In particular, the API Gateway burst capacity prevents your API from being overwhelmed by too many requests. API Gateway throttles requests to your API using the token bucket algorithm. For more information about the token bucket algorithm, visit https://en.wikipedia.org/wiki/Token_bucket.

A token in this case counts as a request, and the burst is the maximum bucket size. At a high level, requests come in to the bucket and 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.

API Gateway sets a quota on a steady-state rate and a burst of request submissions against all APIs in your account. You cannot control the burst capacity, but you can request a modification to the account quota. You can also configure throttling options for your APIs to restrict the volume of requests.



Inspecting and filtering your traffic at each layer allows you to validate the requests and identify and stop invalid requests before they reach your backend services.

Three features within API Gateway can help you do this:
- Enabling cross-origin resource sharing (CORS)
- Validating requests
- Transforming requests

Cross-origin resource sharing (CORS): CORS is a browser security feature that restricts cross-origin HTTP requests that are initiated from scripts running in the browser. The browser enforces CORS, which is only relevant to you if your clients are browsers. If your APIs will receive cross-origin requests, you should enable CORS support. This allows you to accept the requests you want while preventing cross-origin requests to domains that do not explicitly allow the originating domain/origin.

You can configure CORS headers on API Gateway to direct API clients to invoke API calls from only allowed origins. You can enable and configure CORS on both REST and HTTP APIs.

CORS requires a preflight request using the options method to find out which methods are allowed. If the type of request specified in the preflight request is approved, then the actual request gets sent. That call has a Time to live (TTL) setting, which you configure. When you enable CORS for your API, you specify a list of allowed headers, methods, and origins that may access the resource. These are used in response to the preflight request. Requests that do not meet the criteria for the allowed headers, methods, and origins will generate an error.

When you use a REST API and configure CORS, API Gateway sets up a mock endpoint to handle the options method. The setup is simplified when using HTTP APIs. For more information, visit these pages:

- https://docs.aws.amazon.com/apigateway/latest/developerguide/how-to-cors.html
- https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-cors.html
- https://docs.aws.amazon.com/apigateway/latest/developerguide/apigateway-test-cors.html

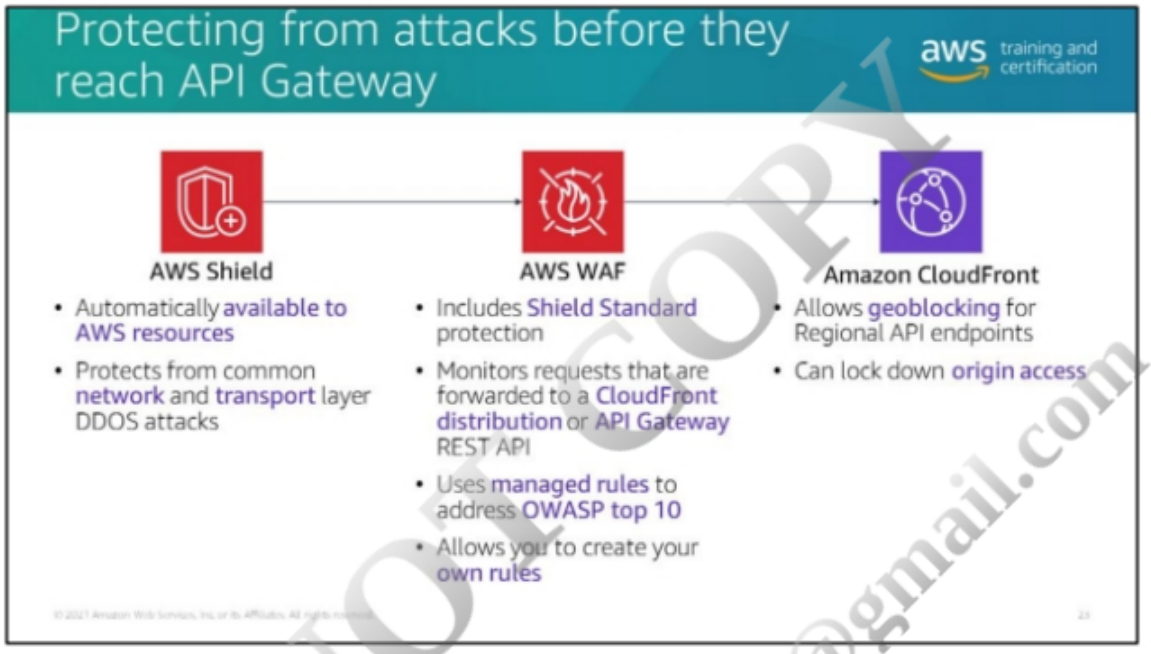
Request transformation: Requests can be transformed by enriching, filtering, and restructuring request data prior to invoking downstream integrations. API Gateway can enrich a request with data returned from Lambda custom authorizers, providing more data to backend services regarding the requestor. This enables the backend service to take appropriate actions, such as allowing or denying the transaction. Additionally, for sensitive request data such as authorization or security details, headers and other request data can be filtered from downstream integrations after successful authorization of the client with API Gateway.

For more information about setting up data transformations for REST APIs, visit https://docs.aws.amazon.com/apigateway/latest/developerguide/rest-api-data-transformations.html.

Request validation: You can validate the format of a request against your defined API models to make sure the expected data is included in the request before sending it to the backend service. If the request properties do not match the API model's schema, API Gateway responds with a 400 Bad Request and does not invoke the backend service.

In this example, the request must have a make value, and only values of Tesla or Hyundai would make it to the backend:

- The first request passes.
- The second fails because its make is not Tesla or Hyundai.
- The third fails because it does not include a "make" value.



The AWS Shield Standard service is automatically available to AWS resources and protects them from common network (3), and transport (4) layer DDOS attacks. Note that AWS Shield Advanced provides higher levels of protection against attacks that target applications running on Amazon EC2, Elastic Load Balancing, Amazon CloudFront, AWS Global Accelerator, and Amazon Route 53 and includes proactive event response and specialized support. For more information about AWS Shield and Shield Advanced, visit https://aws.amazon.com/shield/features/.

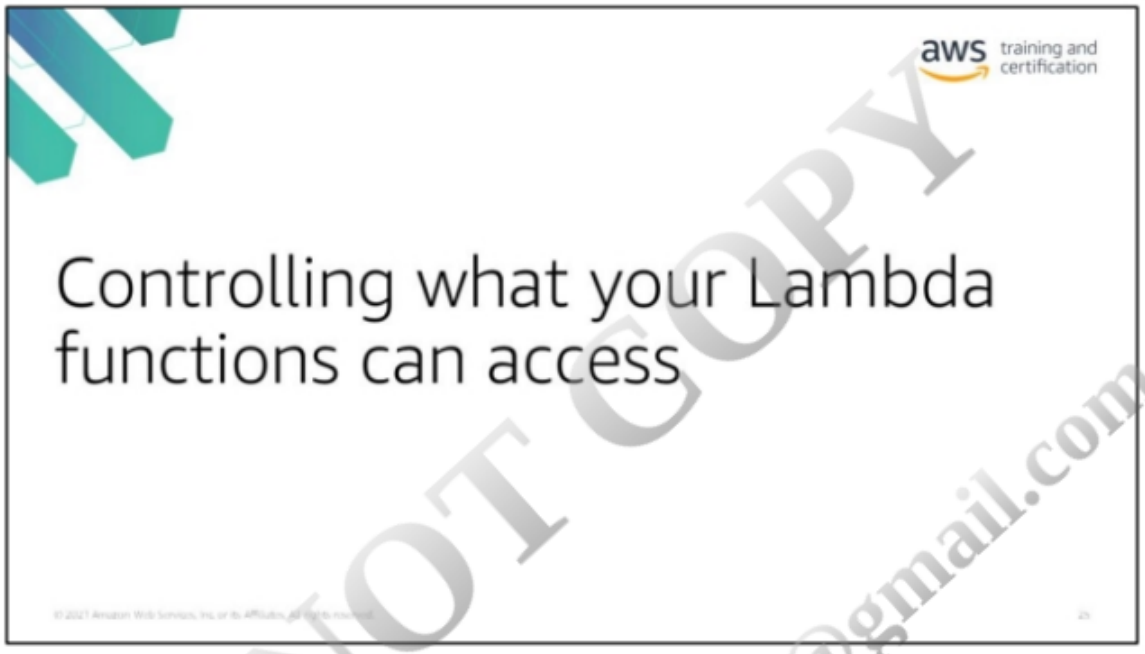
AWS WAF includes Shield Standard protection by default. AWS WAF is a web application firewall that helps protect your web applications or APIs against common web exploits. You can deploy AWS WAF on CloudFront as part of your content delivery network (CDN) solution or on API Gateway for your APIs. You can use managed rules, which are preconfigured sets of rules. The managed rules for AWS WAF address issues like the OWASP Top 10 security risks. These rules are regularly updated. With AWS WAF, you can create security rules that block common attack patterns, such as SQL injection or cross-site scripting, and rules that filter out specific traffic patterns that you define.

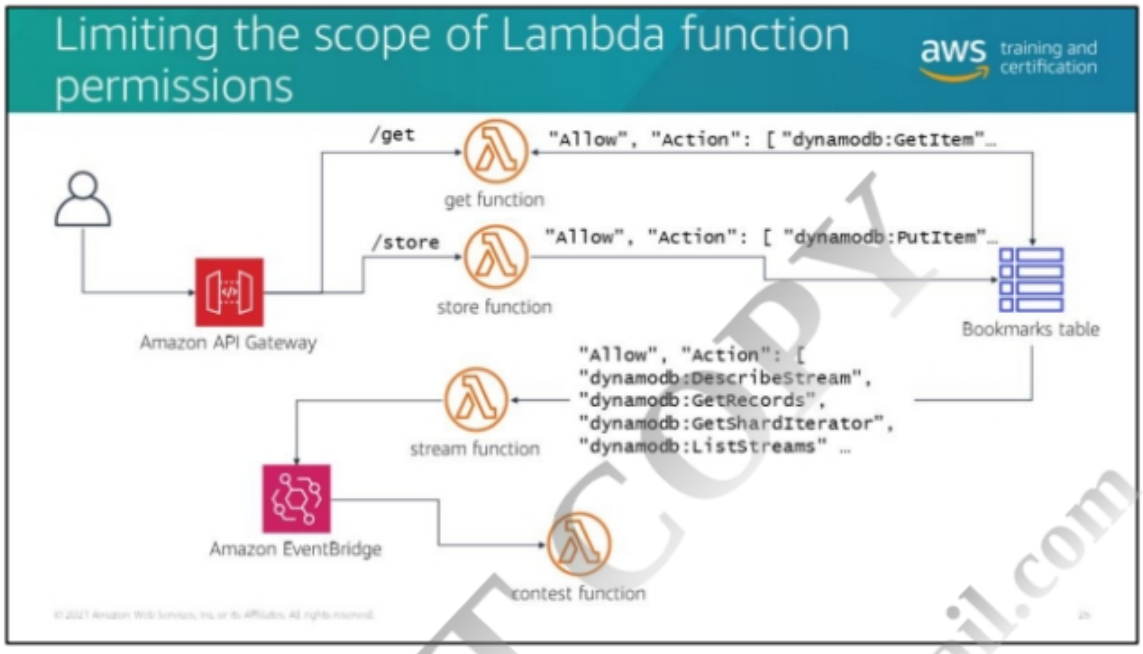
CloudFront and Route 53 provide comprehensive availability protection against all known layer 3 and 4 attacks.

CloudFront distributes traffic across multiple edge locations and filters requests to make sure that only valid requests are forwarded to API Gateway.

When integrating CloudFront with Regional API endpoints, CloudFront also supports geoblocking, which you can use to prevent requests from being served from particular geographic locations.

You can configure API Gateway to accept requests only from CloudFront and use origin access identity (OAI) with Amazon S3 to allow bucket access only through CloudFront.



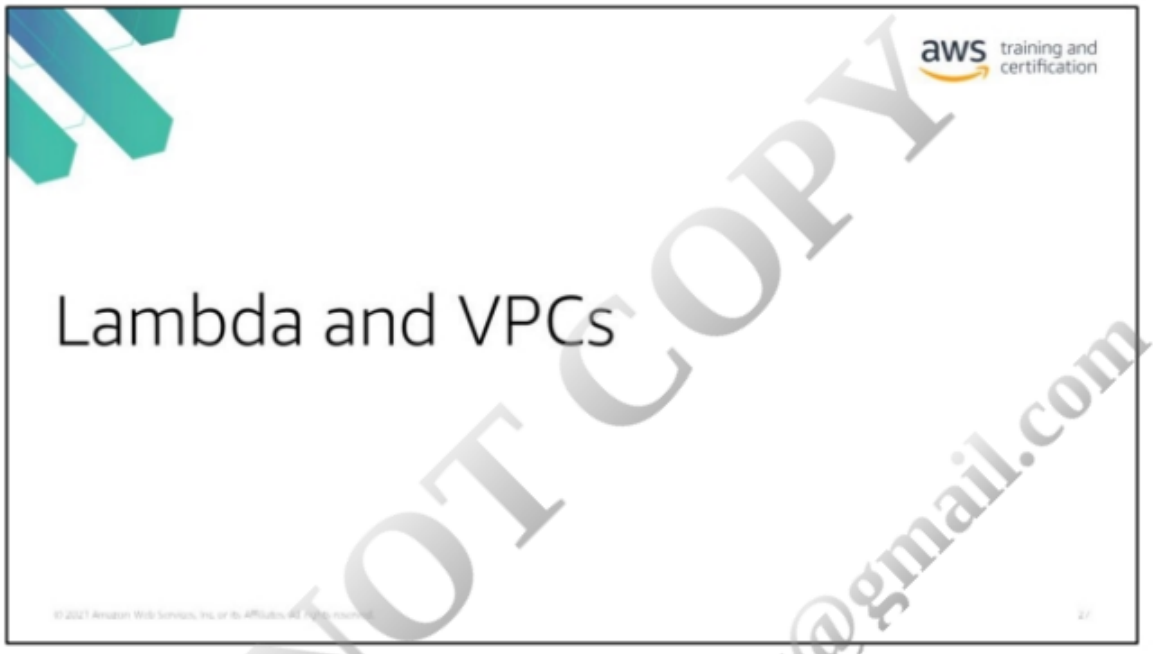


As discussed earlier in the course, use IAM based permissions to control who can invoke a function and what the function is permitted to do. For more information about using resource-based policies for Lambda, visit https://docs.aws.amazon.com/lambda/latest/dg/access-control-resource-based.html. For more information about the Lambda execution role, visit https://docs.aws.amazon.com/lambda/latest/dg/lambda-intro-execution-role.html.

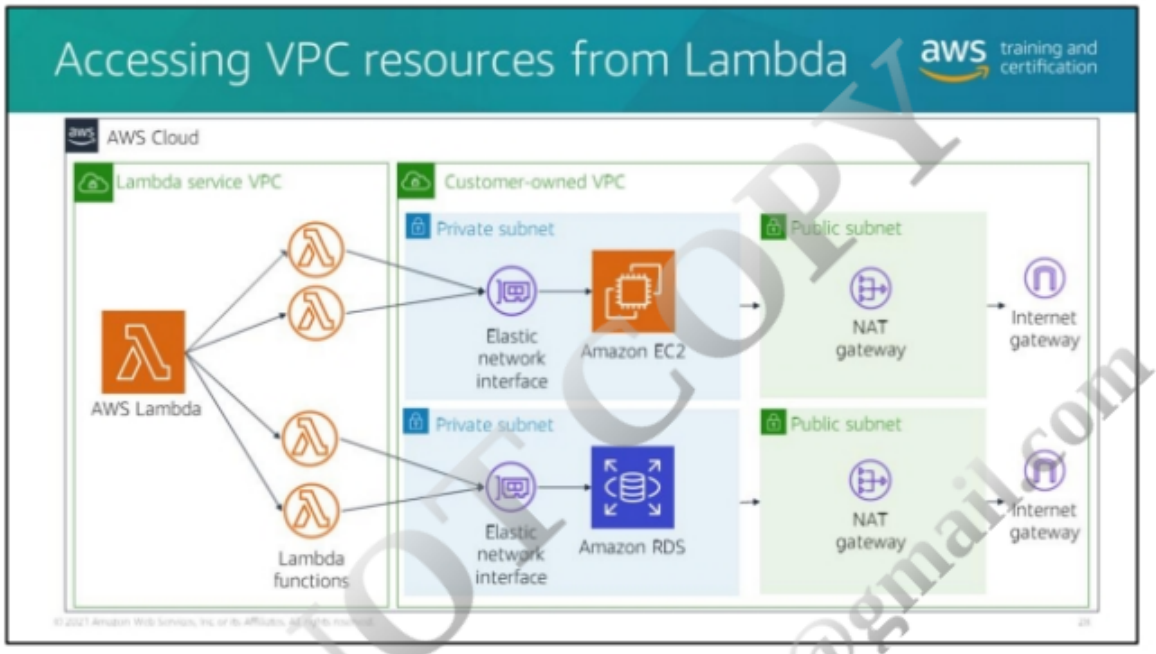
Lambda function permissions should be as granular as possible. For example, in the application you are building, the invoke permissions for the get and store Lambda functions are scoped by the way the API is defined and integrated with Lambda. Each method has permissions to invoke just the function associated with that method, and the Lambda contest function can be invoked only by the Amazon EventBridge rule associated with it. The resource-based policy associated with each Lambda function controls these permissions.

Similarly, limit the scope of what the function can do via its execution role. The get function should have only read permissions for the Bookmarks table that houses the bookmark data, whereas the store function has permissions to write to it. Each function should be given just enough permissions to do its task. If you find yourself needing to give broad permissions to a function, consider splitting it up.

The stream function has specific permissions for reading records off of the Amazon DynamoDB stream for the Bookmarks table. For more information about using an IAM policy to allow a Lambda function to access DynamoDB stream records, visit https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/iam-policy-example-lamda-process-dynamodb-streams.html.



By default, resources within a VPC are not accessible from within a Lambda function. Let's look at how you can allow your Lambda function to connect to resources in your VPC and how to allow your VPC-enabled function to interact with public resources.

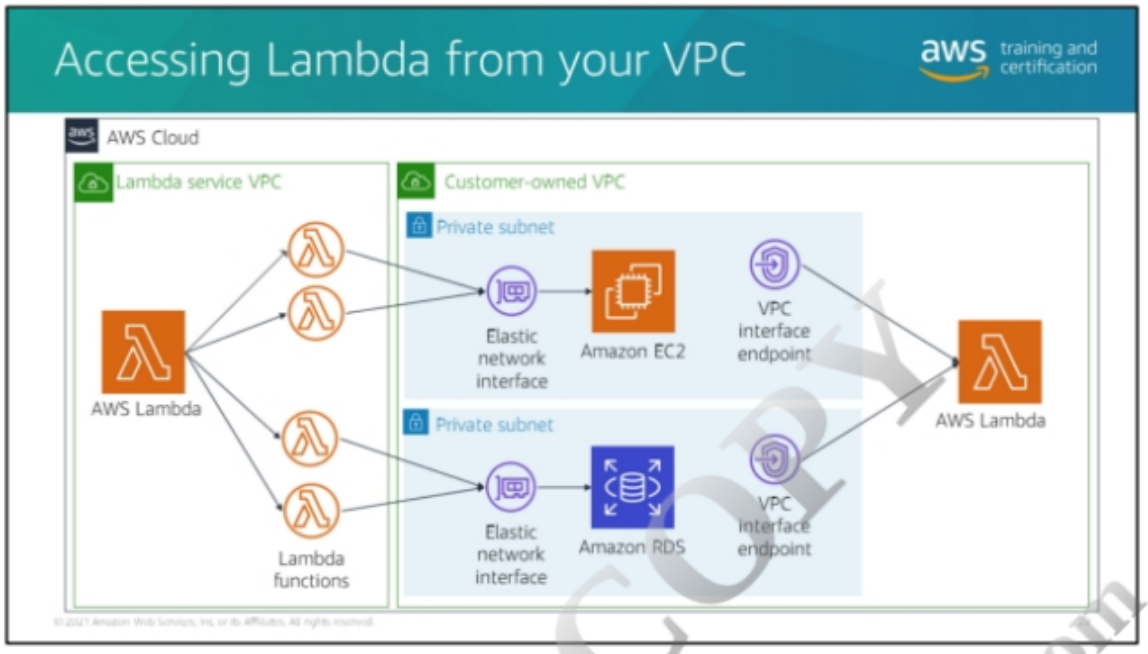


When your function is invoked, Lambda runs your function code securely within an AWS owned VPC.

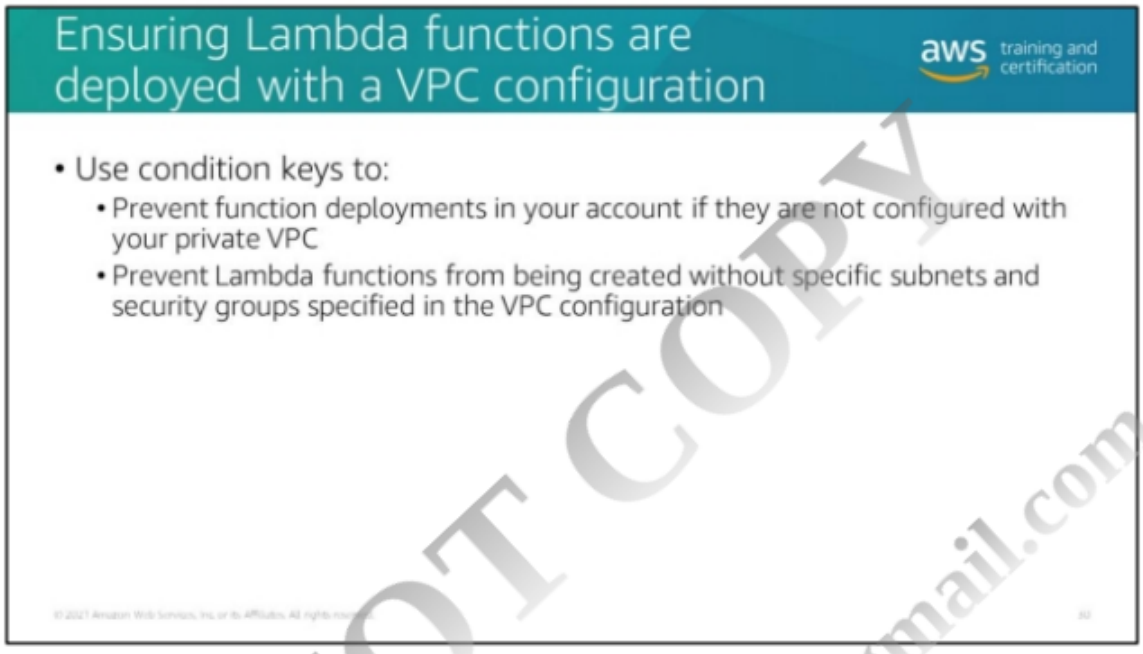
To enable your Lambda function to access **resources inside your private VPC**, you must provide additional VPC-specific configuration information that includes VPC subnet IDs and security group IDs. Lambda uses this information to set up elastic network interfaces, which enable your function to connect securely to other resources within your private VPC.

For this reason, the Lambda function execution role must have permissions to create, describe, and delete elastic network interfaces. Lambda provides a permissions policy, named **AWSLambdaVPCAccessExecutionRole**, that has the necessary permissions. AWS reuses the elastic network interfaces that it creates to route multiple instances of your function to the VPC resources.

Once you VPC-enable your function, all traffic is subject to the routing rules of your VPC, which means that you can target only private subnets and have no direct outbound routes. If your function needs public internet access, you need a route to the public internet via a NAT gateway.



If needing access to the public APIs for Lambda is the only reason you need to access the public internet, you can use a VPC interface endpoint for Lambda and keep your traffic entirely within your VPC. To invoke or manage Lambda through the APIs, you can use that endpoint privately. For details about using VPC interface endpoints with Lambda, visit https://docs.aws.amazon.com/lambda/latest/dg/configuration-vpc-endpoints.html.



With condition keys, you can:

- Prevent function deployments in your account if they are not configured with your private VPC
- Prevent Lambda functions from being created without specific subnets and security groups specified in the VPC configuration

This course doesn't spend time on this feature, but the following links provide more information:

- Documentation: https://docs.aws.amazon.com/lambda/latest/dg/configuration-vpc.html#vpc-conditions
- Blog post: https://aws.amazon.com/blogs/compute/using-aws-lambda-iam-condition-keys-for-vpc-settings/

These links are also available in the OCS.

