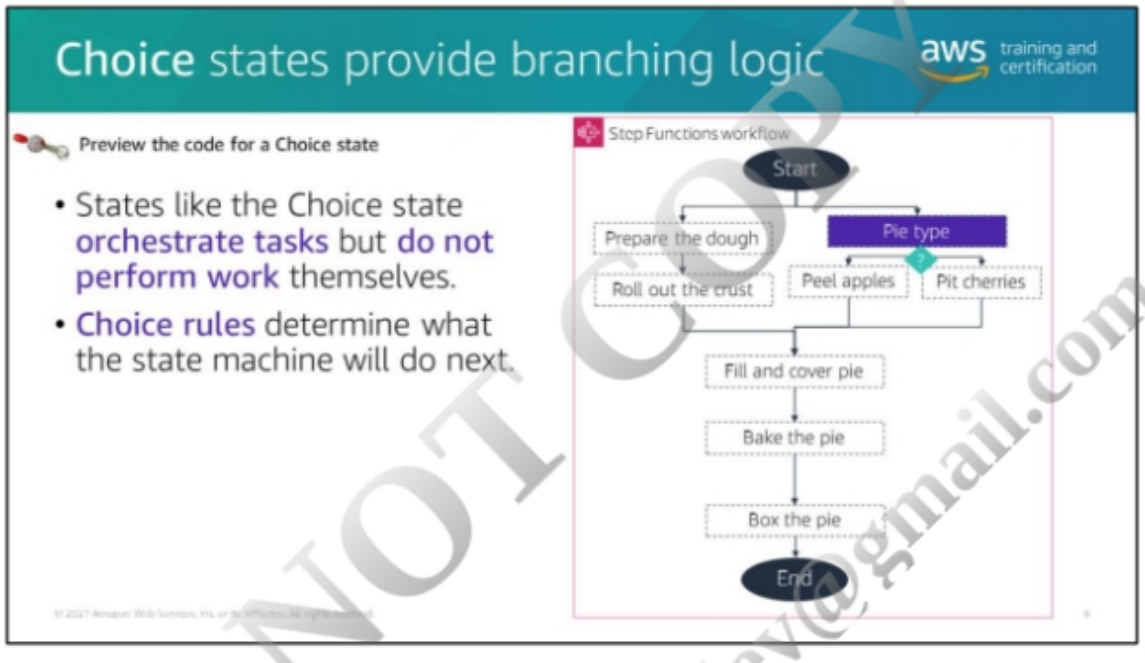
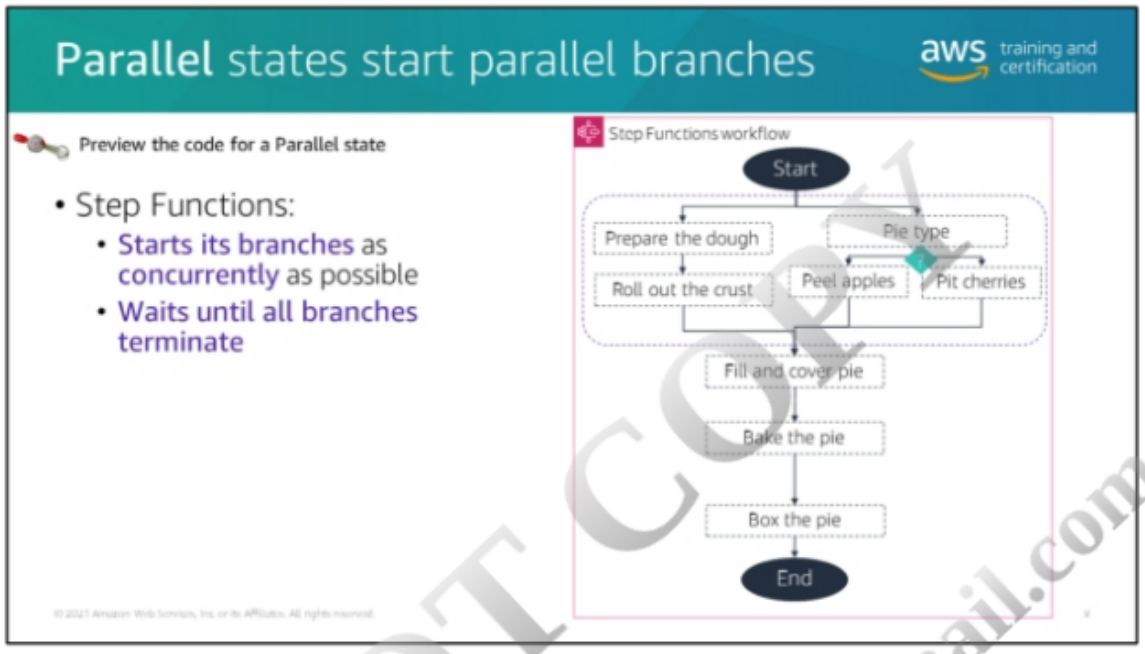
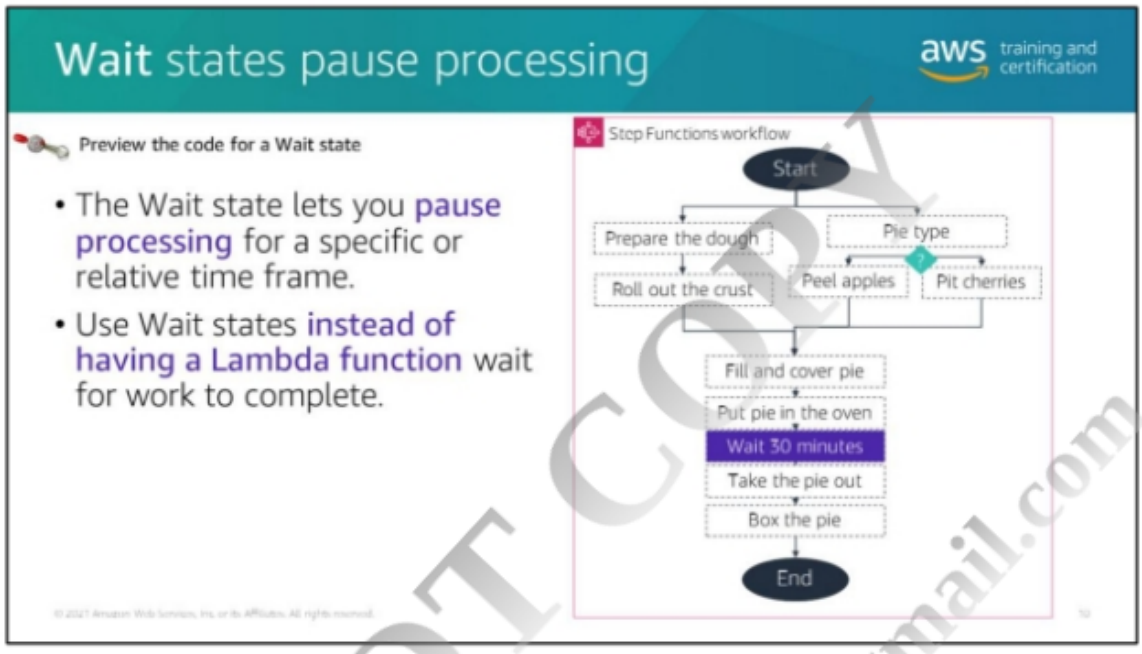


In this example, the Choice state is called "Pie type," and the rules would direct the flow to either peeling apples or pitting cherries as the next step.

You can also specify a default option so that a path exists if the data doesn't match one of the rules.



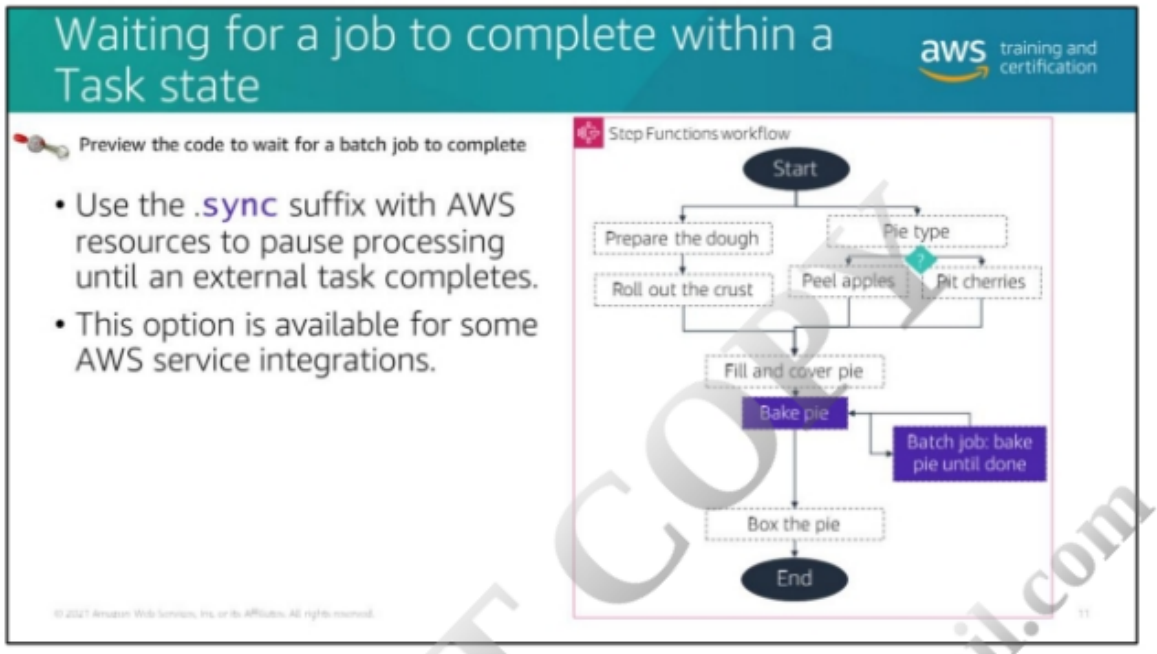


In this example, you replace the "Bake the pie" function with a "Put pie in the oven" function and the "Take the pie out" function with a Wait state set for 30 minutes.

This is more efficient and less costly for Lambda. If you build wait time in to a Lambda function, you're paying for that idle time. The cost of Step Functions is based on the number of state transitions, not how long it takes to move from one state to another.

Often, your wait may be dependent on external factors. You can use a couple of options with Task states to wait for unspecified periods of time:

- Wait for a job to complete
- Wait for a callback with a task token

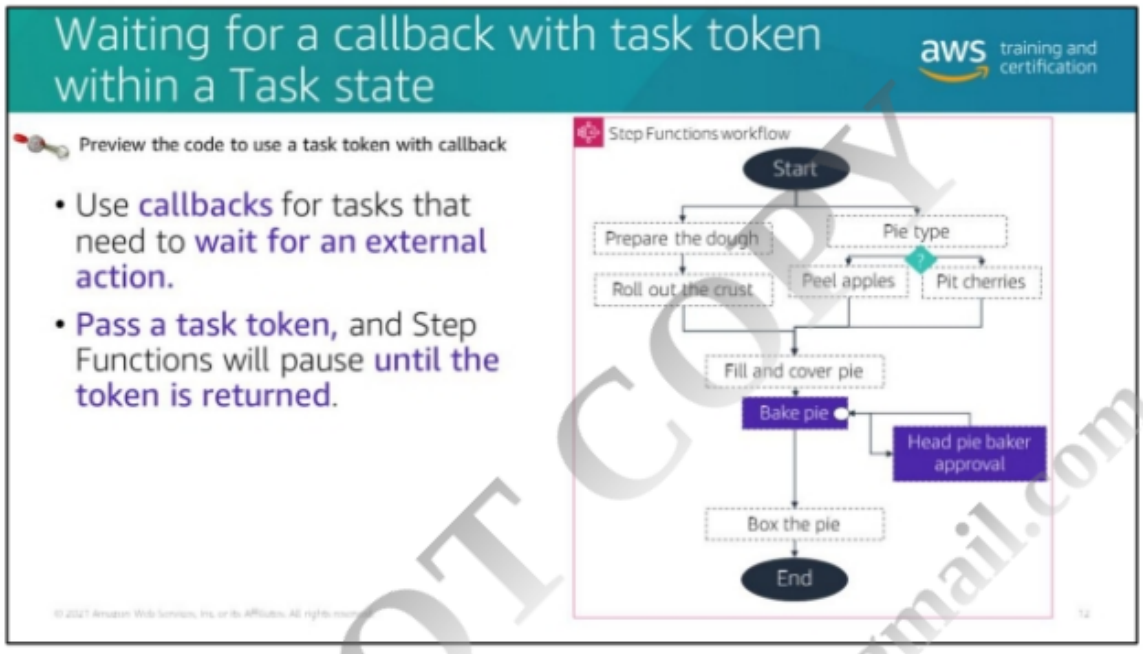


When you indicate an AWS resource (like a Lambda function, Amazon Simple Notification Service [Amazon SNS], Task state), Step Functions waits for an HTTP response and then progresses to the next state. Step Functions doesn't wait for a job to complete.

Some AWS service integrations, such as AWS Batch or Amazon Elastic Container Service (Amazon ECS), support an integration pattern that allows the Step Functions task to wait until the external task is complete before moving on using the **.sync** suffix.

To perform a similar "wait for something to finish" flow for a service that doesn't have a direct integration that supports the **.sync** suffix, follow the model found in this sample Step Functions project. For more information, visit https://docs.aws.amazon.com/step-functions/latest/dg/sample-project-job-poller.html.

For more information about which service integration patterns are supported for each service integration, visit https://docs.aws.amazon.com/step-functions/latest/dg/concepts-service-integrations.html.



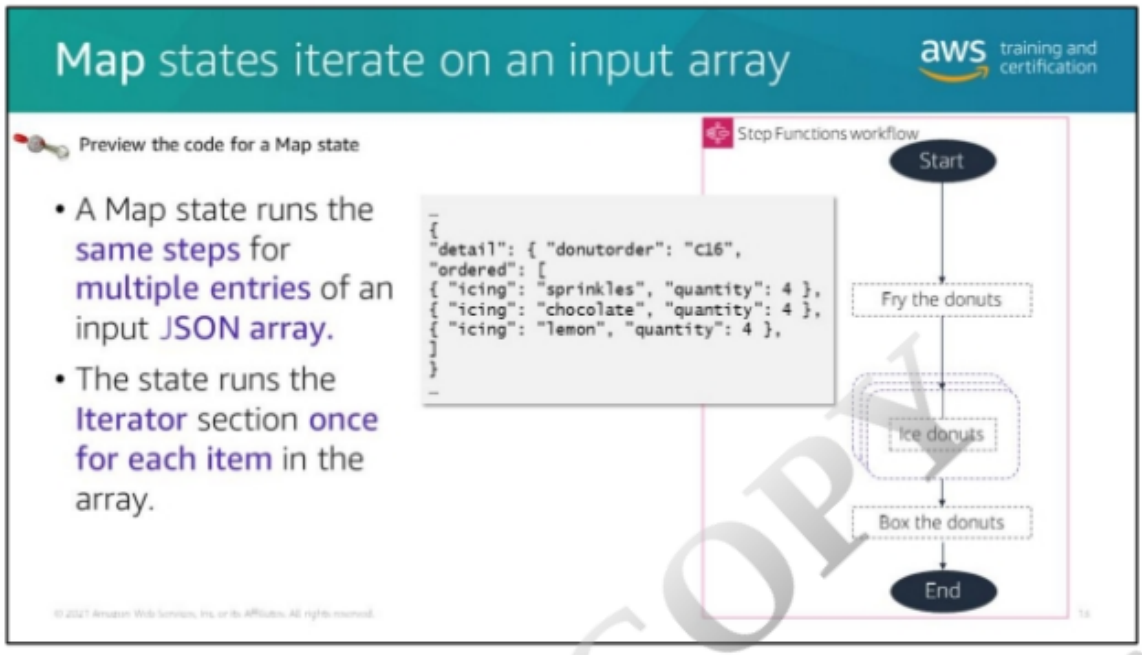
Alternatively, you might need the processing to pause until you get an external response (for example, a human approval or a third-party system that has to process your request and respond).

Some AWS services that are integrated with Step Functions can use a task token that you pass to them, and they return when the work is complete.

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

Configure your Task state to pause and wait for the task token using ".waitForTaskToken".

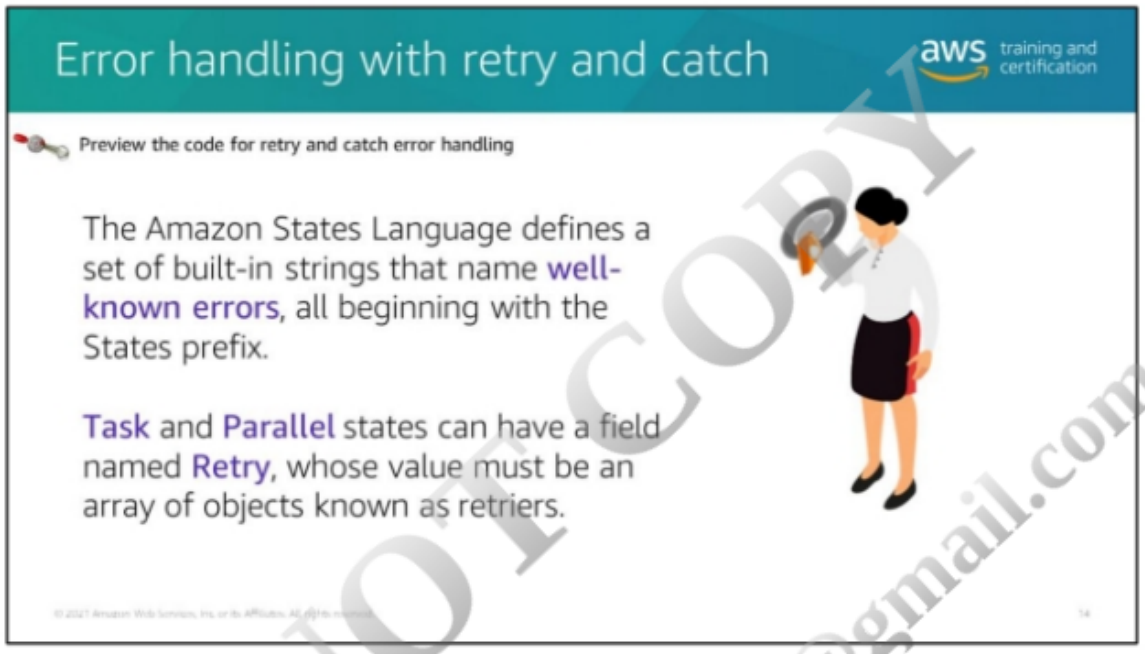
For more information : https://docs.aws.amazon.com/step-functions/latest/dg/connect-to-resource.html.



Often, the input into a step may be a list of items of a varying number. Use Map states to run the same steps for **multiple entries of an array** in the state input.

For example, if you switch to the donut ordering state machine, you might use a Map state to have the "Ice donuts" function run for each input in the array for the customer's donut order. The same function runs for each item in your array using values from the individual array item. The Map state isn't complete until the function has iterated through all of the items in the array.

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



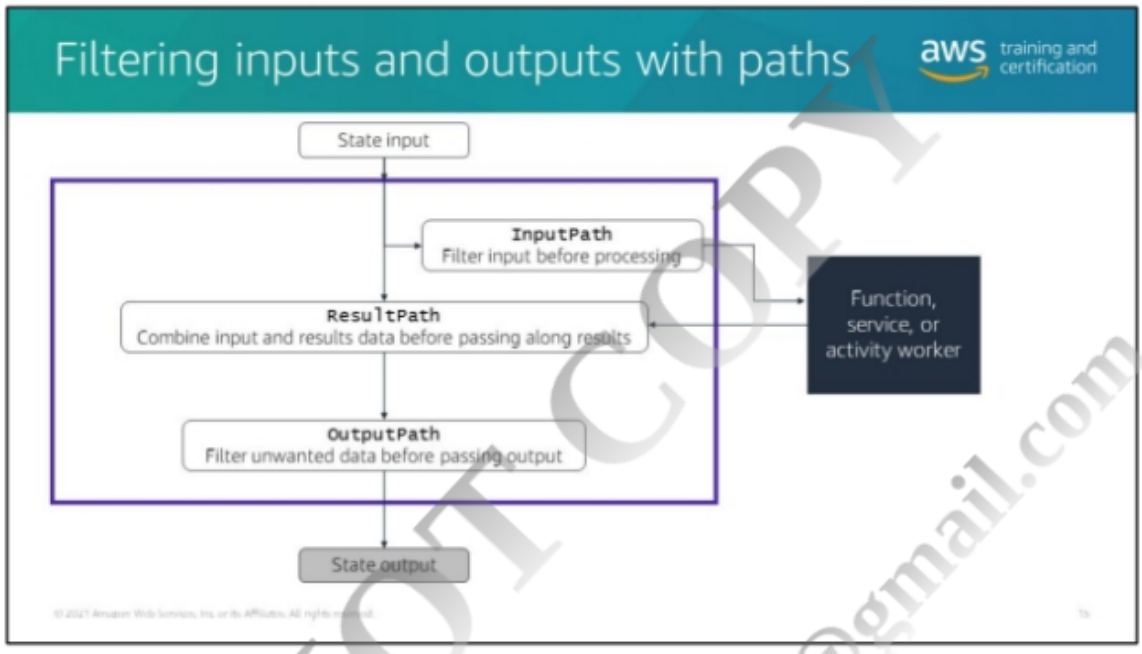
For information about error handling in Step Functions, visit https://docs.aws.amazon.com/step-functions/latest/dg/concepts-error-handling.html.

Any state can encounter runtime errors. Errors can happen for various reasons:

- State machine definition issues (for example, no matching rule in a Choice state)
- Task failures (for example, an exception in a Lambda function)
- Transient issues (for example, network partition events)

By default, when a state reports an error, Step Functions causes the execution to fail entirely. Step Functions identifies errors in the Amazon States Language using case-sensitive strings known as *error names*. The States language defines a set of built-in strings that name well-known errors, all beginning with the **States** prefix.

Task and Parallel states can have a field named Retry, whose value must be an array of objects known as *retriers*. An individual retrier represents a certain number of retries, usually at increasing time intervals.



For information about input and output processing in Step Functions, visit https://docs.aws.amazon.com/step-functions/latest/dg/concepts-input-output-filtering.html.

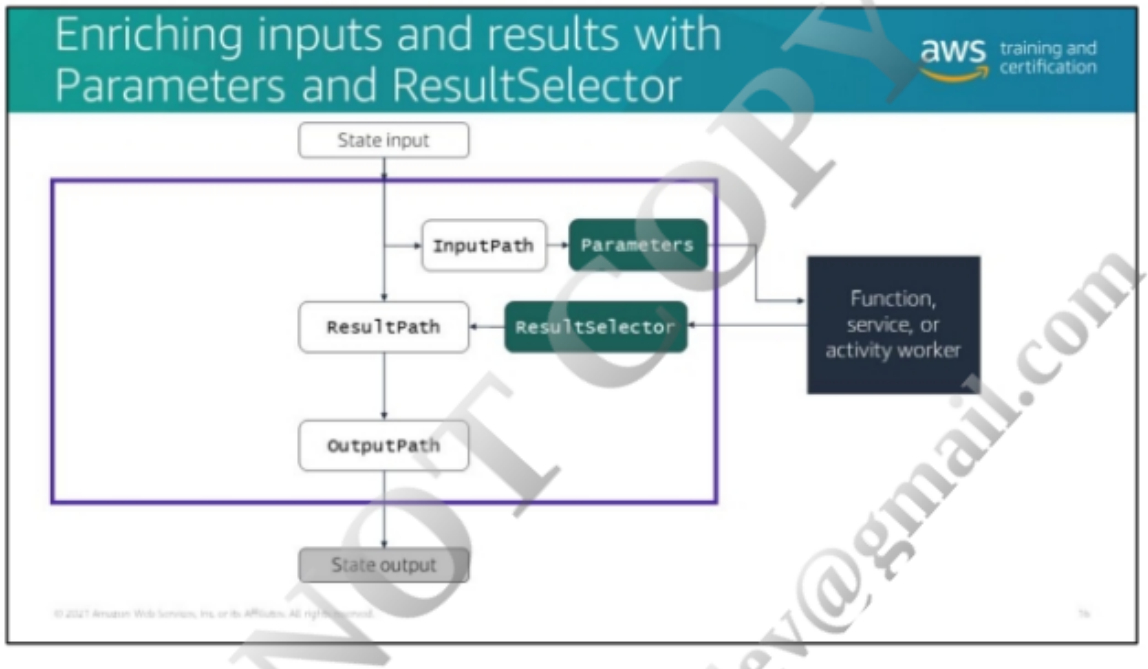
Within each state, you can use additional fields to filter the inputs and outputs.

Paths let you select portions of the JSON. With all states except the Fail state, you can use **InputPath**, **ResultPath**, and **OutputPath** to filter the data before and after the step processes it.

InputPath lets you limit the input that is passed to the function, service, or worker in a Task state.

ResultPath lets you choose how results are passed on. Results can be a copy of the input, the results produced, or a combination of both.

OutputPath lets you select a portion of the output to pass to the next state.



For information about these features, visit https://docs.aws.amazon.com/step-functions/latest/dg/concepts-input-output-filtering.html.

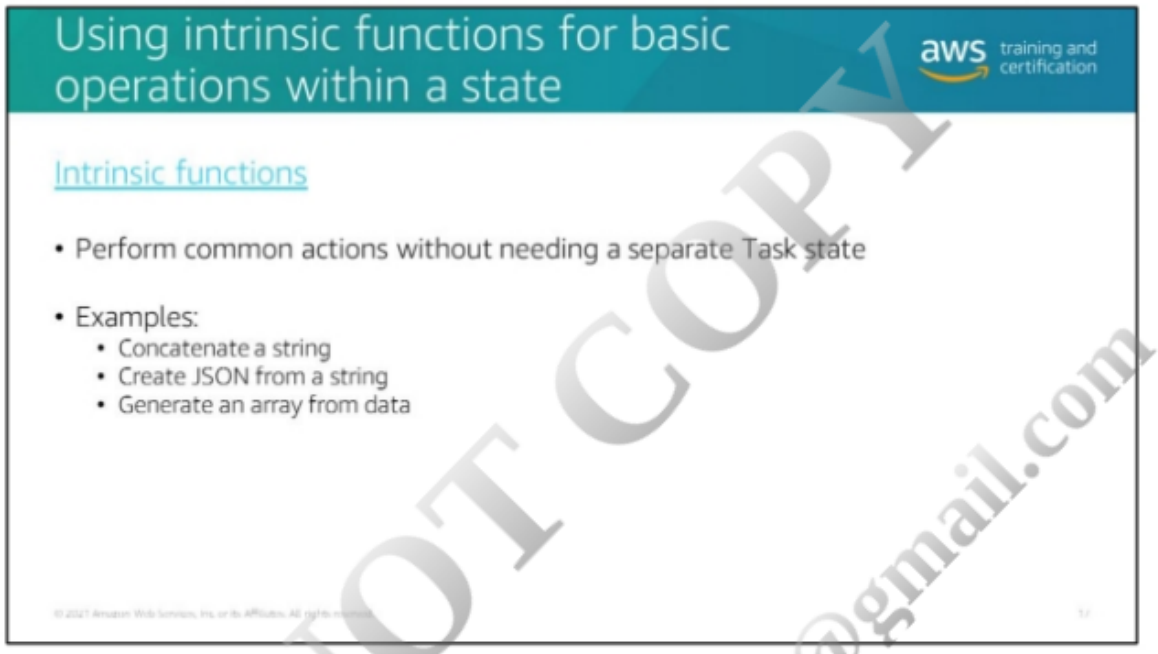
The **Parameters** and **ResultSelector** fields let you enrich the input and output data within the state.

The Parameters field lets you create a collection of key-value pairs that are passed as input. These can be static values, but they can also be selected from the input or the **context object** with a path. For more information about the context object, visit https://docs.aws.amazon.com/step-functions/latest/dg/input-output-contextobject.html.

The context object contains information about your state machine so that you can access information specific to the individual execution. During an execution, the context object is populated with data for the Parameters field where you can access it. This is useful for things like task tokens or Amazon Simple Queue Service (Amazon SQS) message IDs.

The ResultSelector field lets you manipulate the results before the ResultPath is applied and is available for Task, Map, and Parallel states.

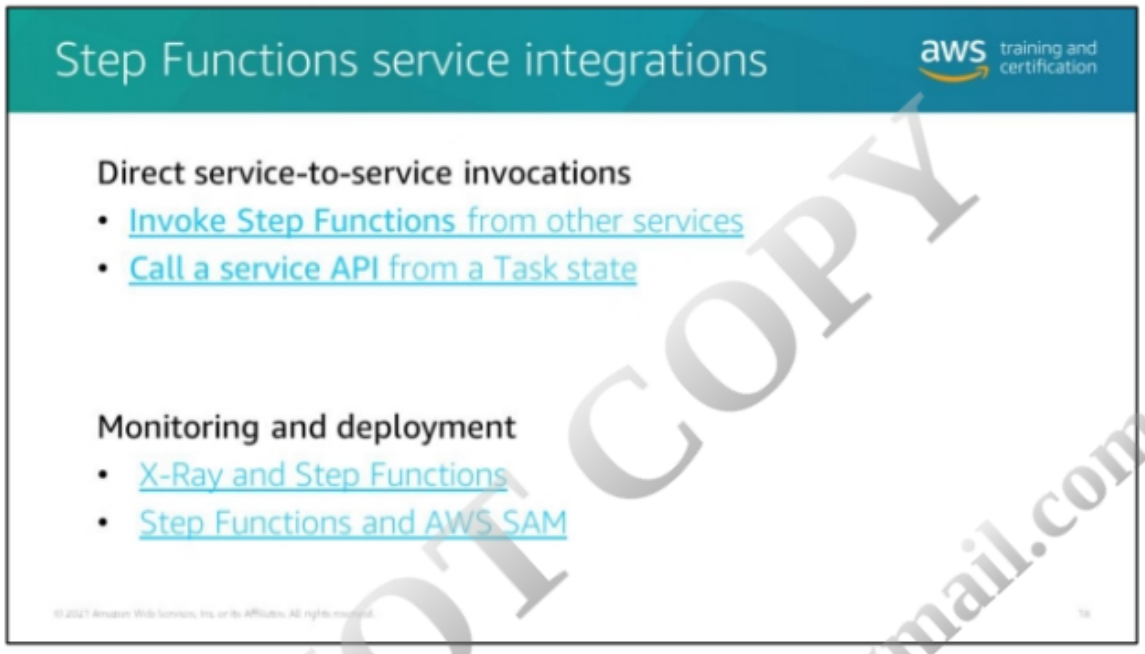
For example, you could select portions of the metadata returned with service integrations and merge them with the state input with ResultPath.



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

The Parameters and ResultSelector fields also support intrinsic functions for some state types. Intrinsic functions let you do basic operations, such as concatenating a string, creating JSON from a string, or generating an array from data. Using intrinsic functions lets you perform these types of common actions without needing to create a separate Task state that runs a function to return the modified data.

This link is also available in the OCS.



Step Functions continues to add integrations with other AWS services, and improve the ease with which you can write and validate your Step Functions.

When you are building your state machine, look at the options for invoking based on factors related to your workflow and payload. For example, when would you want to directly use Amazon API Gateway to initiate the state machine versus having a Lambda function behind API Gateway to initiate the state machine?

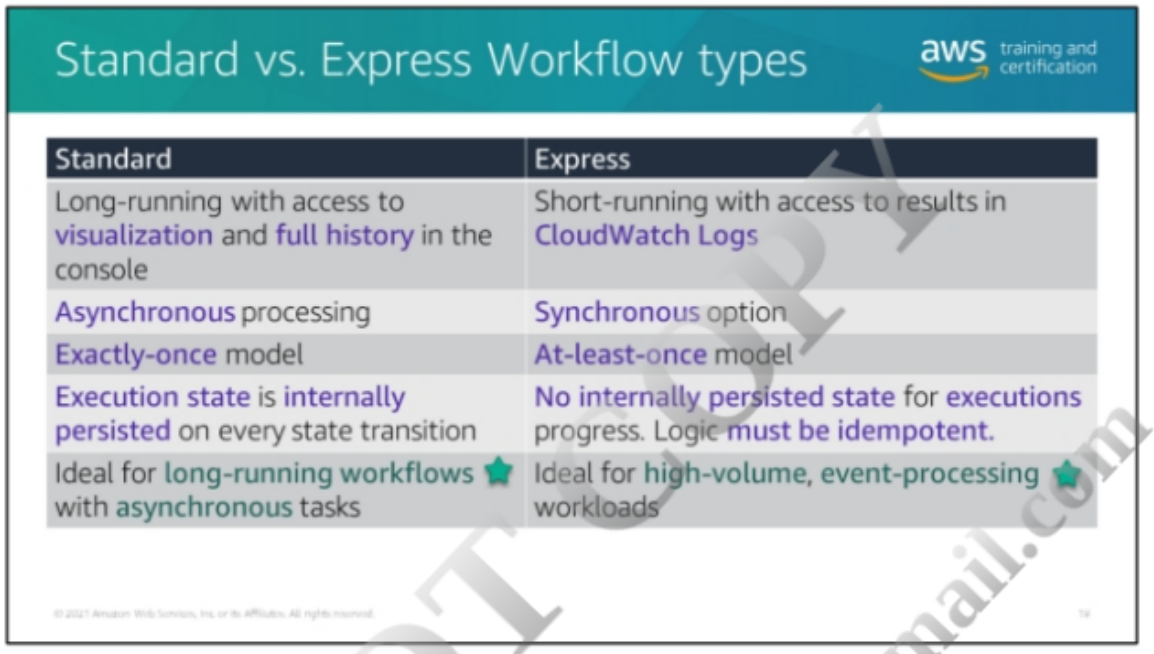
As noted earlier, you can interact with many services directly from a Task state, and integration patterns make it easier to build in logic for waiting on a job to finish or returning a task token with much less coding. Always check on the available service integrations before trying to write your own logic.

For the latest options for invoking Step Functions directly from an AWS service, visit https://docs.aws.amazon.com/step-functions/latest/dg/concepts-invoke-sfn.html.

For the latest integrations you can use from within Step Functions, visit https://docs.aws.amazon.com/step-functions/latest/dg/concepts-service-integrations.html.

Step Functions also supports AWS X-Ray to help you trace what's happening within a state machine. The visualization you have in the console to quickly reference your state machine's flow is available from within the AWS Serverless Application Model (AWS SAM).

For more information about using Step Functions with X-Ray, visit https://docs.aws.amazon.com/step-functions/latest/dg/concepts-xray-tracing.html.
For more information about using Step Functions with AWS SAM, visit https://docs.aws.amazon.com/step-functions/latest/dg/concepts-sam-sfn.html.



For information about Standard versus Express Workflows, see https://docs.aws.amazon.com/step-functions/latest/dg/concepts-standard-vs-express.html.

You have been working with Standard Workflows in your example, but depending on what you are trying to orchestrate, Express Workflows may be a better option.

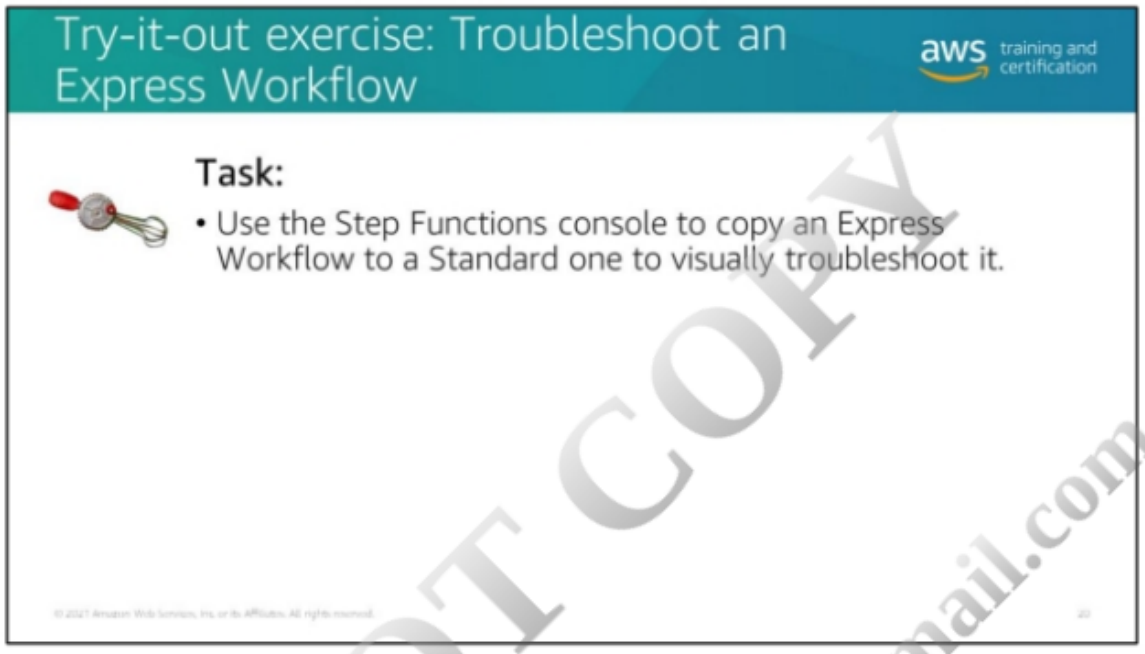
Standard Workflows are best for long-running workflows with asynchronous tasks, whereas Express Workflows were designed for high-volume event processing where the duration is shorter.

Express Workflows also support synchronous request/response workflows. This allows developers to quickly receive the workflow response without needing to poll additional services or write additional code. This is useful for high-volume microservice orchestration and fast compute tasks that communicate via HTTPS; for example, a multi-step sign-up process could be handled synchronously.

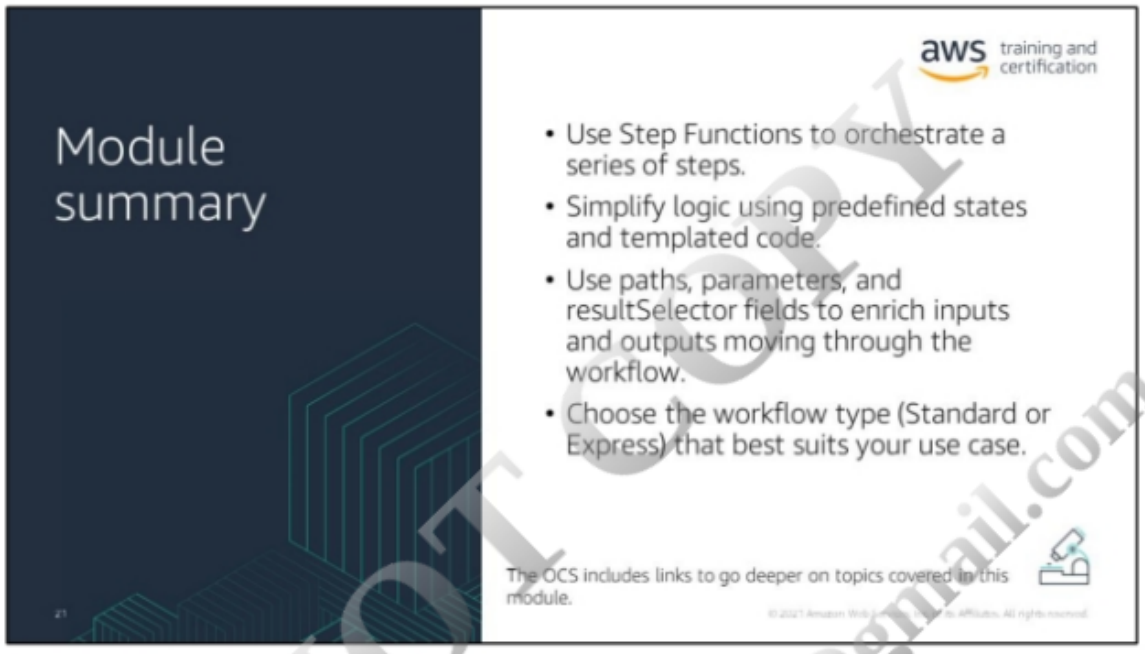
For more information about synchronous workflows, visit https://aws.amazon.com/blogs/compute/new-synchronous-express-workflows-for-aws-step-functions/.

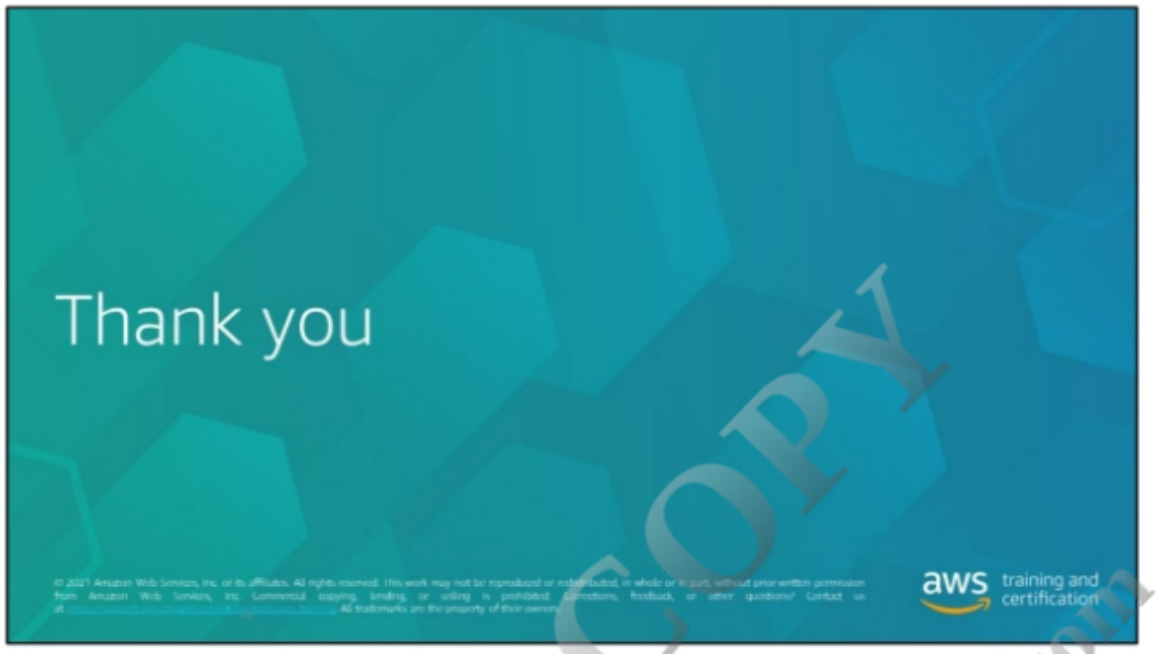
Different service integration patterns are supported. For a table that highlights which patterns are available for each type of workflow, visit https://docs.aws.amazon.com/step-functions/latest/dg/concepts-service-integrations.html.

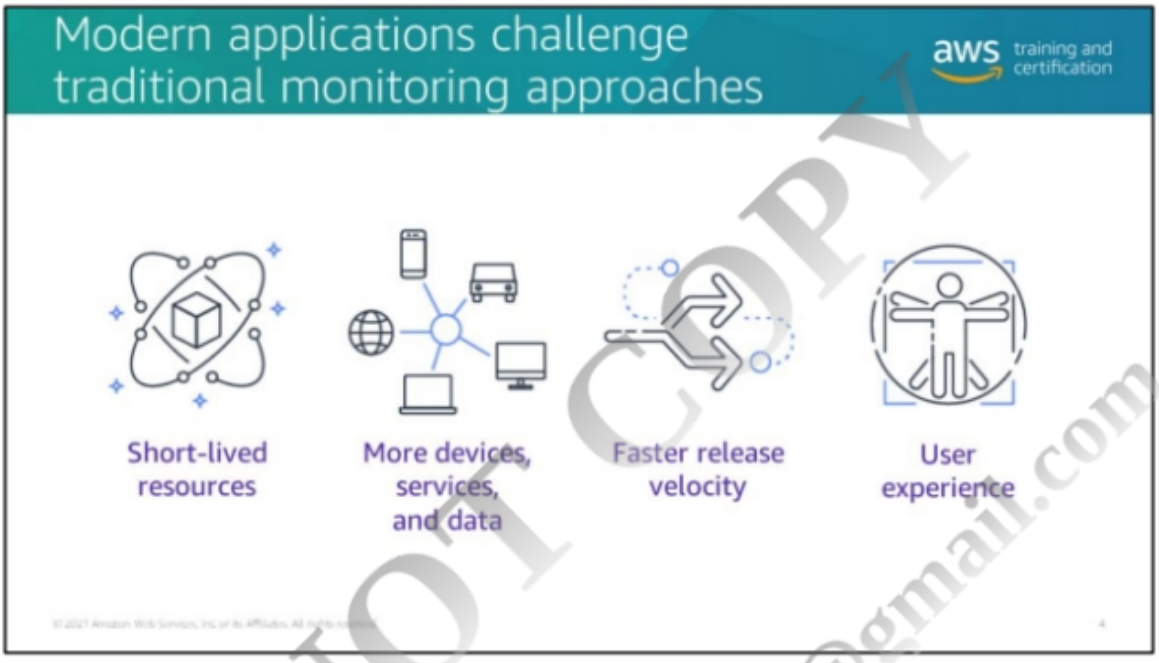
Even when the production workflow may be best suited to an Express Workflow, developers have found it helpful to work on getting their code or workflows right using a Standard Workflow because of its visual representation of errors and the execution history that provides easy-to-access details about each run or each step in the run. Once you have the workflow working as desired, if the workload is more suitable to an Express Workflow, you can copy the Standard Workflow to an Express Workflow.



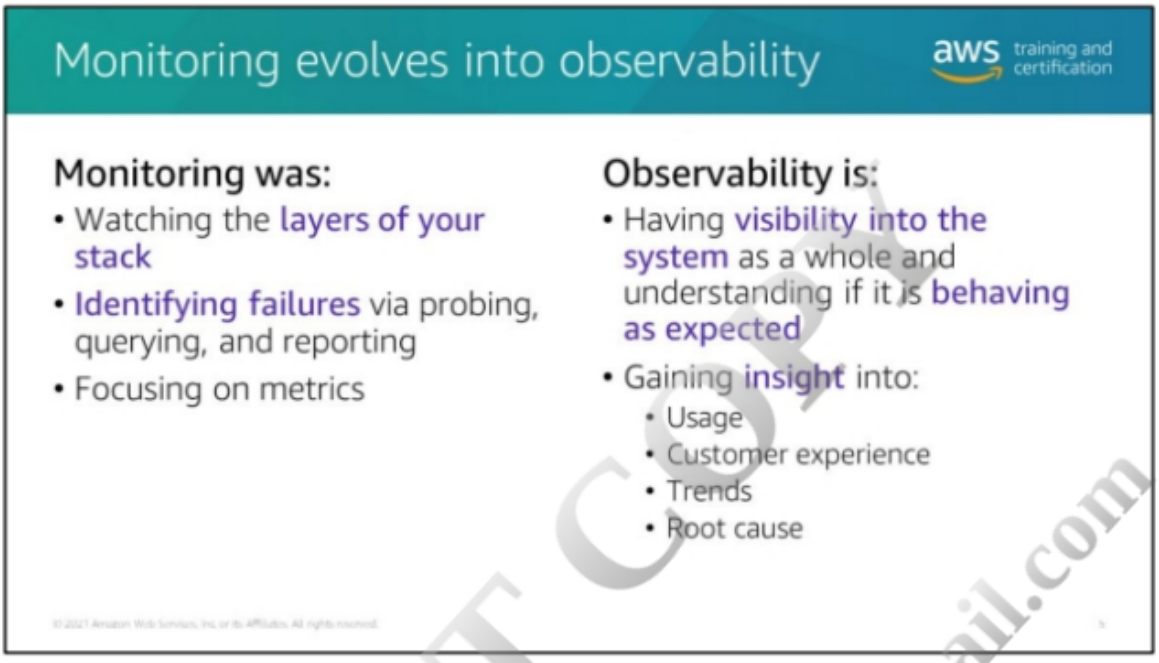
In this task, you use the copy feature to copy an Express Workflow to a Standard one. This is a common developer scenario when you want the functionality of the Express type but want to use the visual modeling and history available in Standard Workflows.







With modern applications, and particularly with serverless ones, you have to deal with short-lived resources, many different services connected in a distributed way, and faster release velocity of independent components. The user experience has also become a more important consideration for the performance of your applications; for example, users don't tolerate slow connections. This type of monitoring requires a different approach than alerting on errors in one of the layers in your stack.



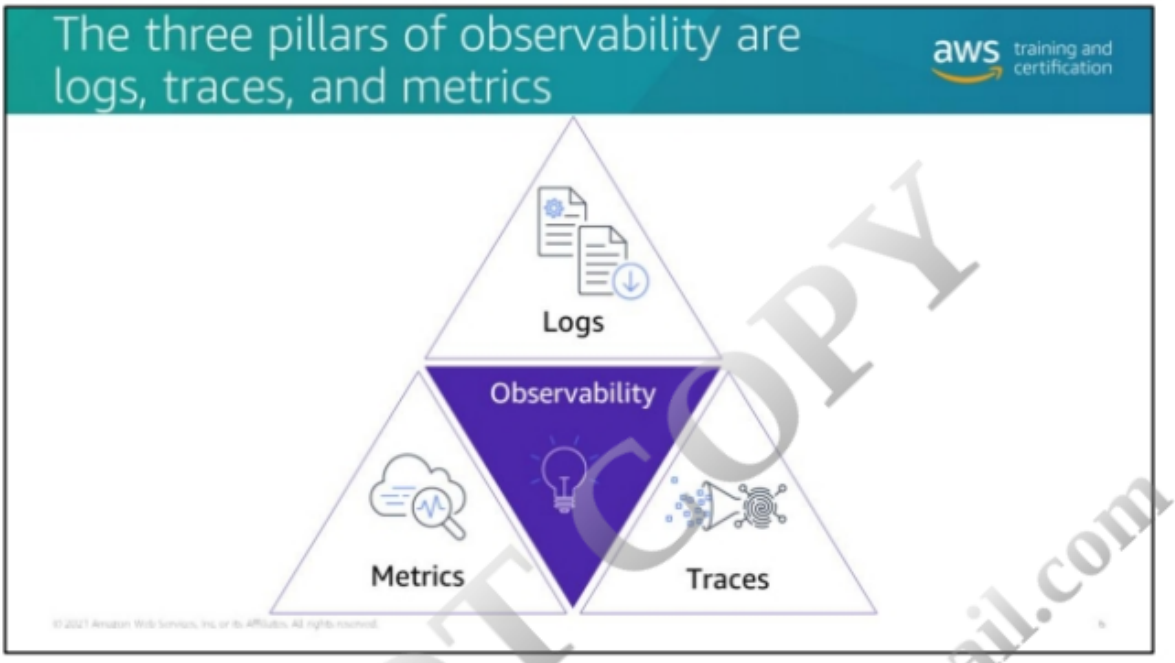
Previously, monitoring focused on identifying failures across the layers of your stack, but a dashboard that only tells you there are no known failures doesn't tell you everything. Is your application performing in line with your expectations? What are the usage and access patterns? What parts of your application are getting used more or less than you designed for?

You also want business information about your system. Is it generating revenue? Where are you seeing growth? What trends can you identify and plan for?

The term *observability* is derived from control theory.
Reference: Aguirre LA, Portes LL, Letellier C (2018) Structural, dynamical and symbolic observability: From dynamical systems to networks. PLoS ONE 13(10): e0206180. https://doi.org/10.1371/journal.pone.0206180

Obserability reflects an approach to organizational culture and development practices that drive how you think about collecting and using data about your system to do more than monitor for failures.

Observability extends traditional monitoring with approaches that address the kinds of questions you want to answer about your applications. Business metrics are sometimes an afterthought, only coming into play when someone in the business asks the question, and you need to figure out how to get the answers from the data you have. But if you build in these needs when you're building the application, you will have much more visibility into what's happening within your application.



Logs, traces, and metrics are the three pillars of observability:

- **Logs** are timestamped records of discrete events, such as a failure, error, or state change. Logs may include metrics.
- **Traces** give you a single transaction of your user's journey through separate applications or parts of an application.
- **Metrics** provide numeric data measured at intervals. These are often service-level indicators like request rate or error rates but can also be business metrics like the number of orders or payments received.

These three pillars must be **linked contextually** to achieve observability in your system. For example, if you are alerted on a metric, you should be able to quickly find the logs related to the metric and quickly dive into the traces associated with the events.

With an observability approach, failure is something that is embraced, not just something that is monitored after the application is built. The failure modes are designed as part of the application during the software development lifecycle.

Make sure that you are collecting enough data to actually understand the system. If data is missing, you could have gaps in your visibility.

Importantly, incorporate the data you want for debugging while you are building your application. Also, incorporate production testing mechanisms like canaries into your deployment processes. A later module will provide more information about this topic.

This module looks at each of these pillars in more detail through the lens of the AWS services that support them.

References:

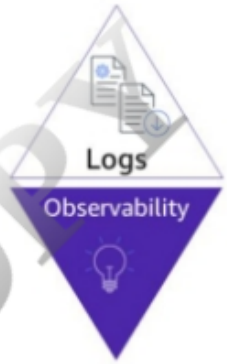
- For more information about distributed systems observability from GitHub, visit https://github.com/keyvanakbary/learning-notes/blob/master/books/distributed-systems-observability.md.
- For more information about distributed systems observability from O'Reilly Inc., visit https://www.oreilly.com/library/view/distributed-systems-observability/9781492033431/.

CloudWatch Logs centralizes logs from applications and AWS services

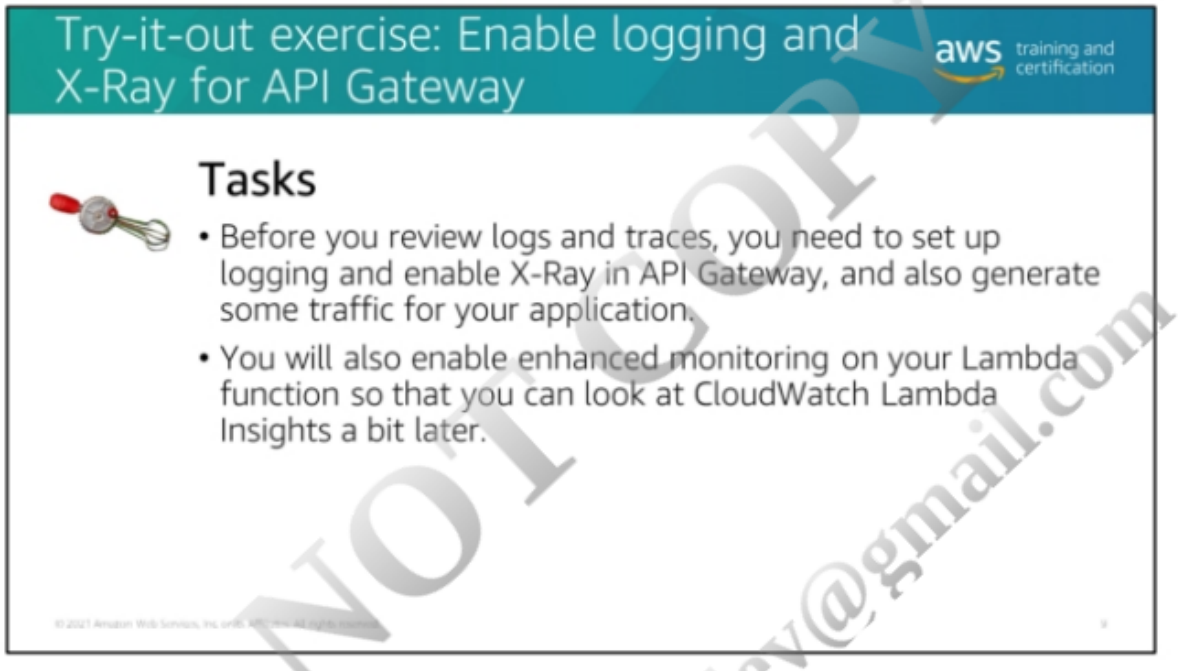
aws training and certification

Developer features

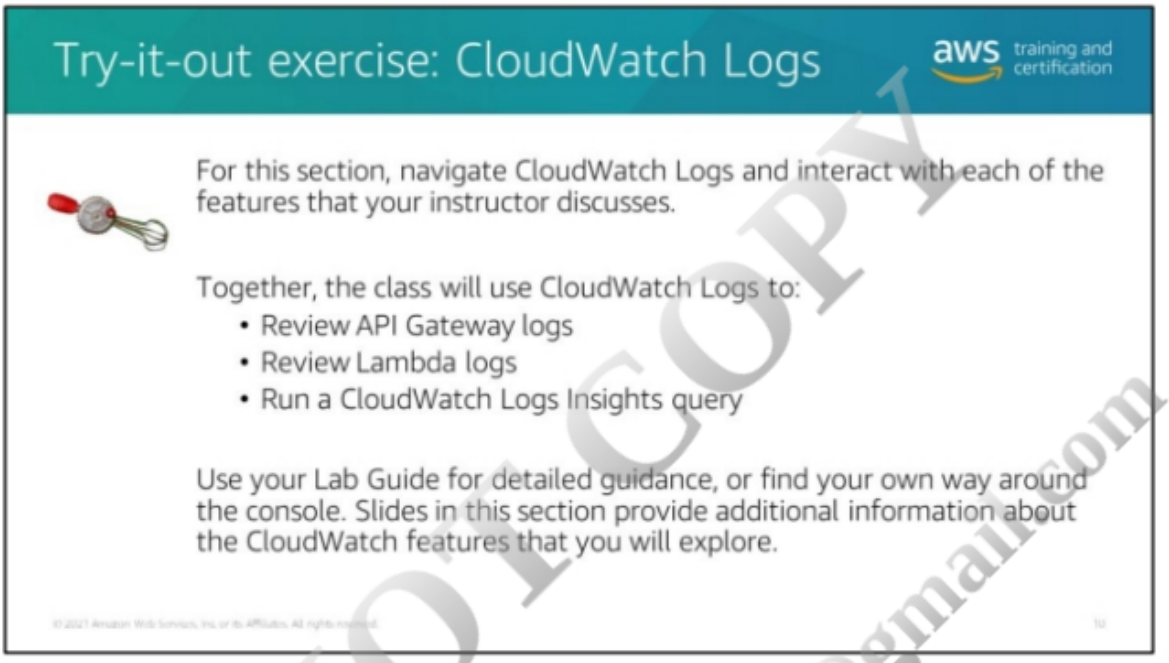
- Review logs as a **flow of events ordered by time.**
- **Query** log data interactively with **CloudWatch Logs Insights.**
- Use log data in **metric filters** and **alarm** on them.
- **Export** log data for additional **analysis** or **processing.**



With Amazon CloudWatch Logs, you can review and query your logs. You can also act on log data via metric alarms or export your data for additional use cases. You might export it to Amazon Simple Storage Service (Amazon S3), stream it via Amazon Kinesis Data Firehose to Amazon Elasticsearch Service (Amazon ES) to use tools like Kibana to analyze your logs, or export it to a third-party tool for analysis or processing.

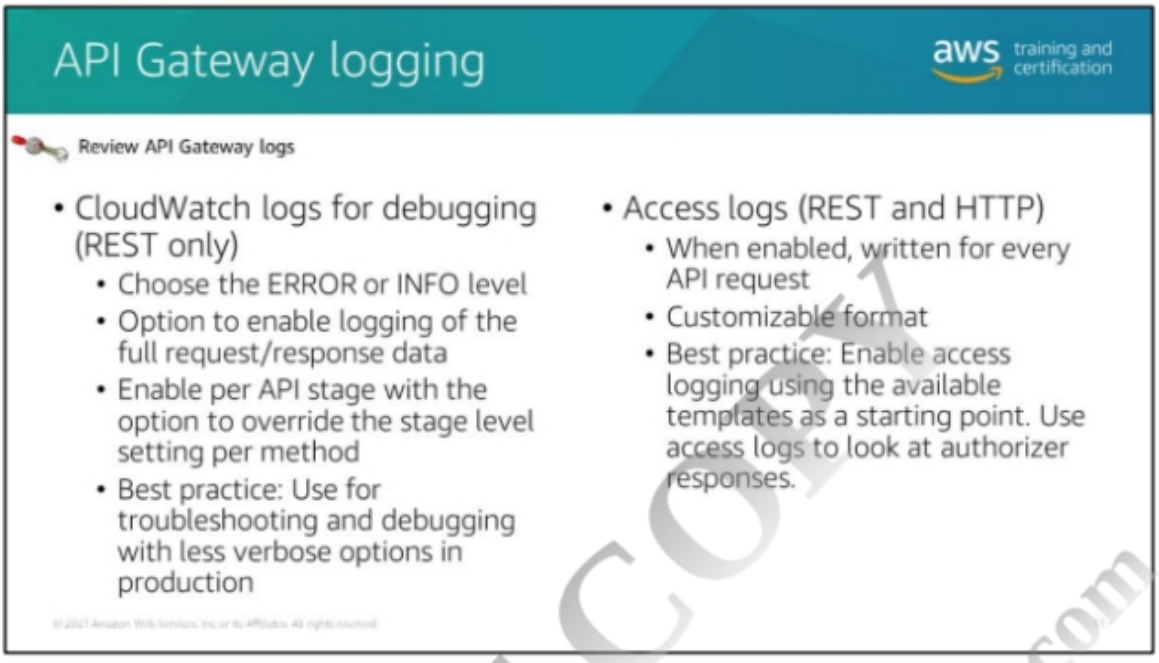


In this task, you will use the serverless backend application, which sends traffic through an Amazon API Gateway to AWS Lambda and interacts with an Amazon DynamoDB table. You will use AWS Cloud9 to send requests to your API.



Use the CloudWatch console to review API Gateway and Lambda logs.

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

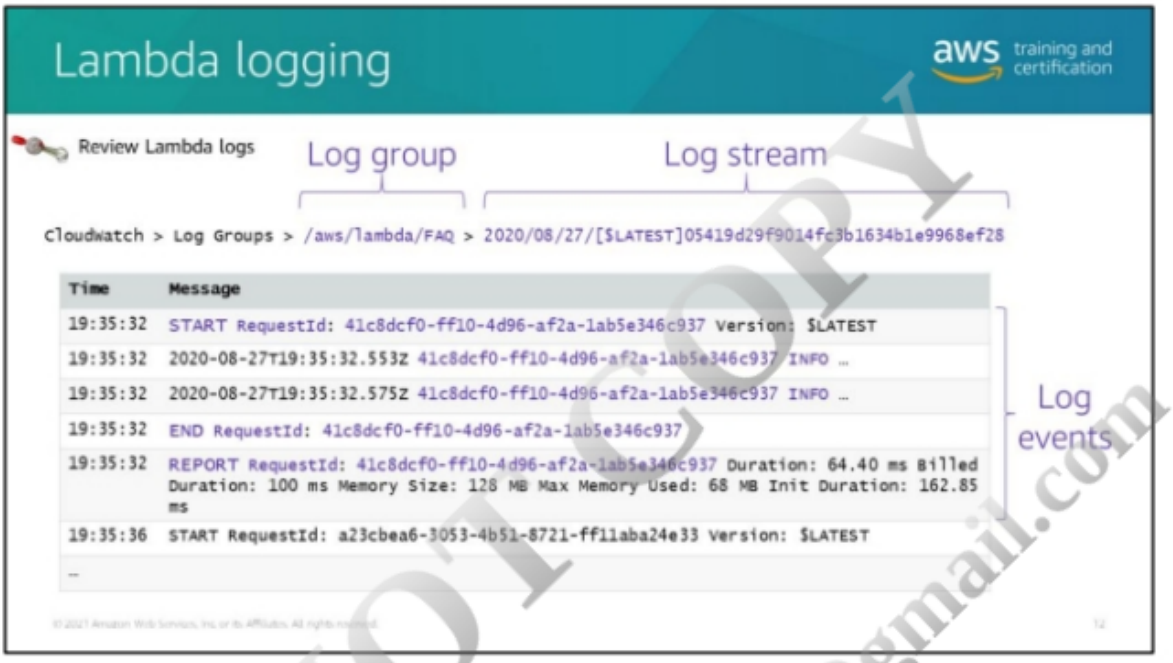


By default, when you create an API in API Gateway, logging is not configured. You must explicitly enable logging as you just did at the start of this try-it-out (TIO) exercise.

Logging works a bit differently for REST versus HTTP types of APIs.

You have not configured access logging in the example application, but it is available for both REST and HTTP APIs. As a general best practice, enable access logging and start with one of the provided templates. At a minimum, you should use logging to look at authorizer responses.

For more information about best practices for using access logs with enhanced observability variables with your APIs, visit https://aws.amazon.com/blogs/compute/troubleshooting-amazon-api-gateway-with-enhanced-observability-variables/.



Lambda logs all requests and results in a CloudWatch log group called /aws/lambda/<**function name**>. A one-to-one relationship exists between function and log group. The first time the function runs, the group is created. If you set up provisioned concurrency, the log group is created when you create the provisioned concurrency. The log group is the entity you would typically use for searching events within your Lambda logs.

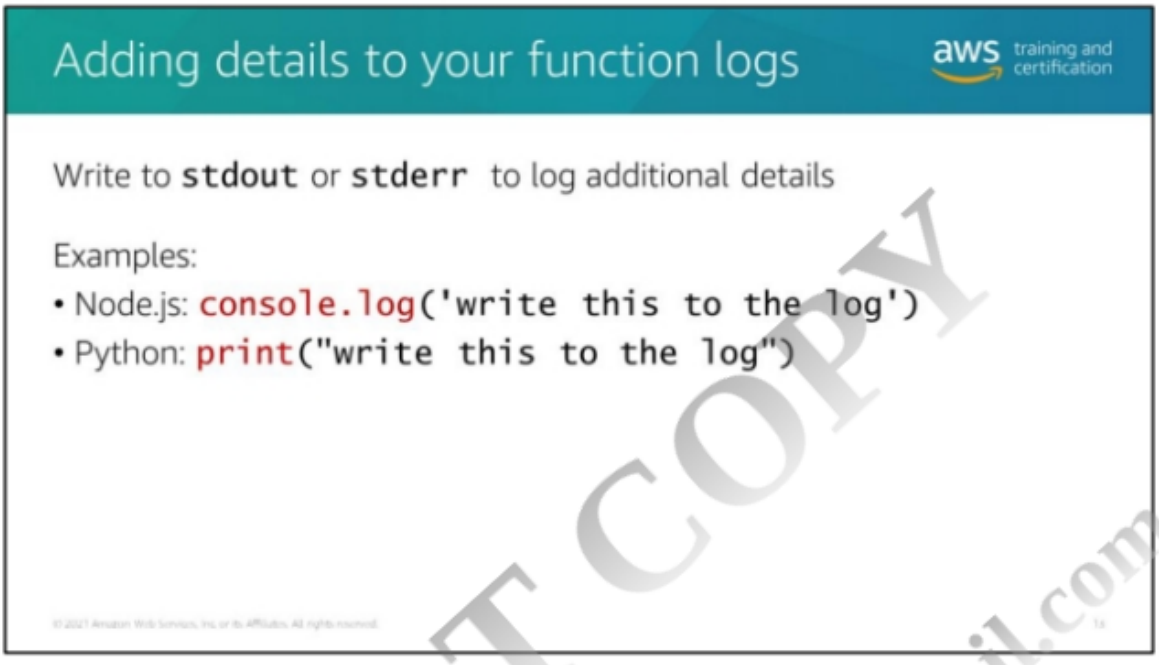
The log group is made up of log streams identified by the date, **function version**, and a unique identifier. Lambda automatically creates new log streams within a group based on internal factors related to how invocations are allocated to environments and the timing of invocations.

Each log stream contains a sequence of events describing Lambda request details. Each request includes a unique **RequestID** and includes a START event, END event, and REPORT event, which quickly tell you summary information about the Lambda request. The RequestID is the primary identifier that you will use to find events related to a particular invocation. The REPORT event is a quick way to see how long the function ran, the billing duration, as well as the memory configured for the function and the memory it used. The "Init" duration in this example tells you that this invocation was a cold start.

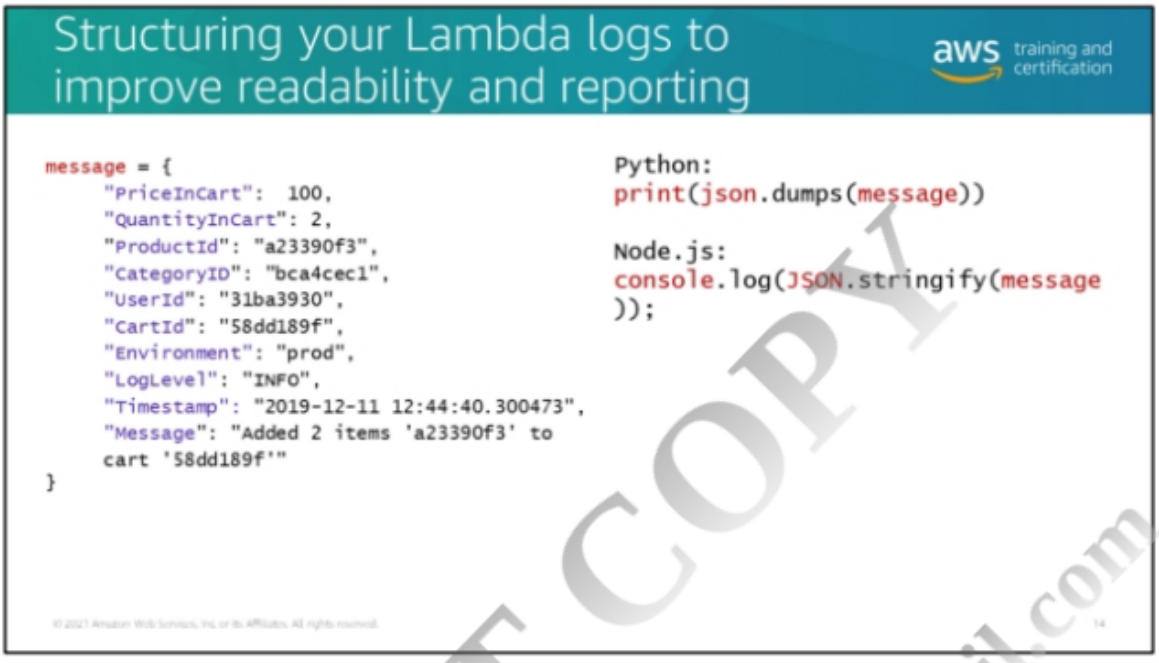
For more information about CloudWatch Logs, visit https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/WhatIsCloudWatchLogs.html.

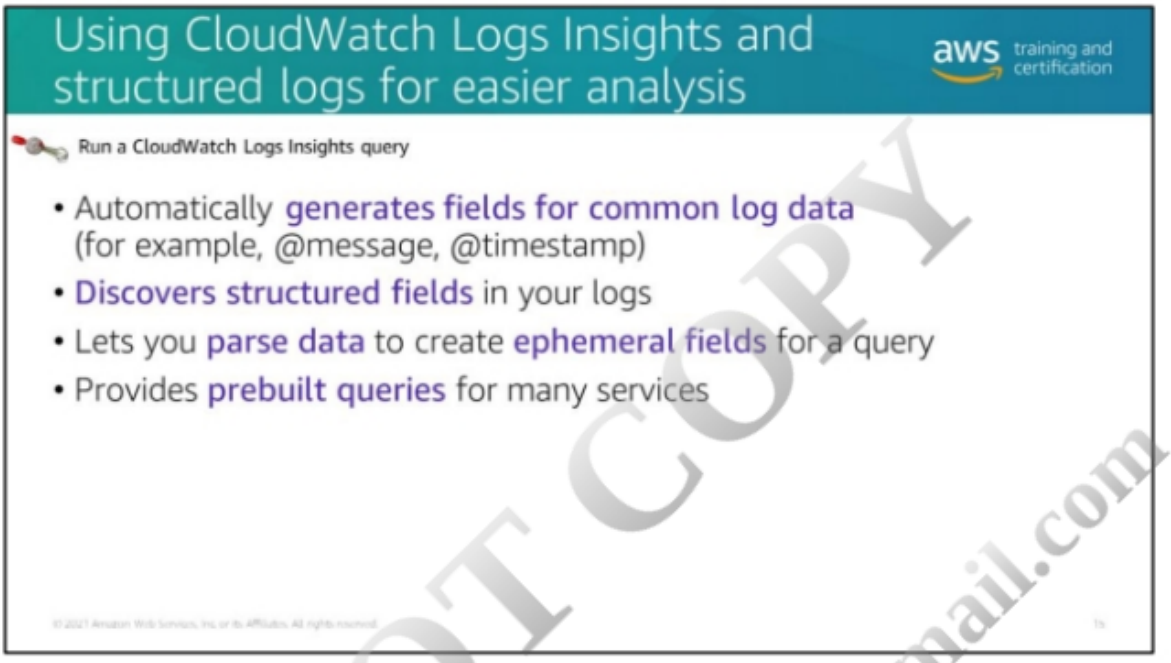
For more information about accessing CloudWatch Logs for Lambda, visit https://docs.aws.amazon.com/lambda/latest/dg/monitoring-cloudwatchlogs.html.

Links are also available in the OCS.



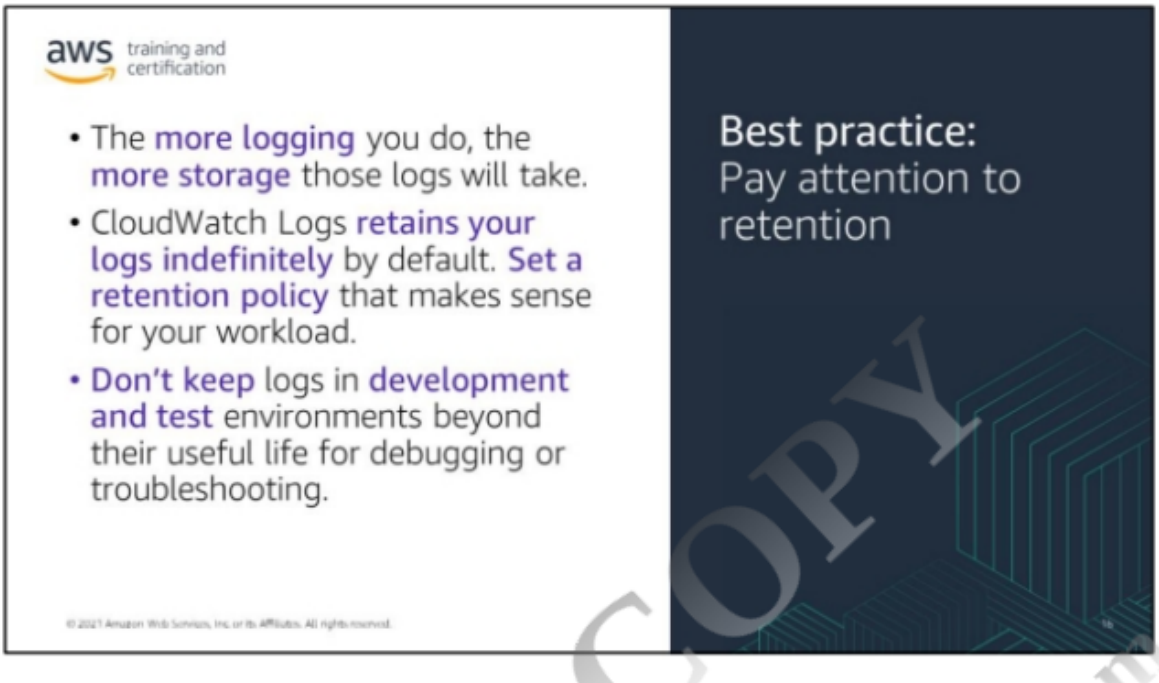
Use the runtime command or any logging library that writes to **stdout** or **stderr**.



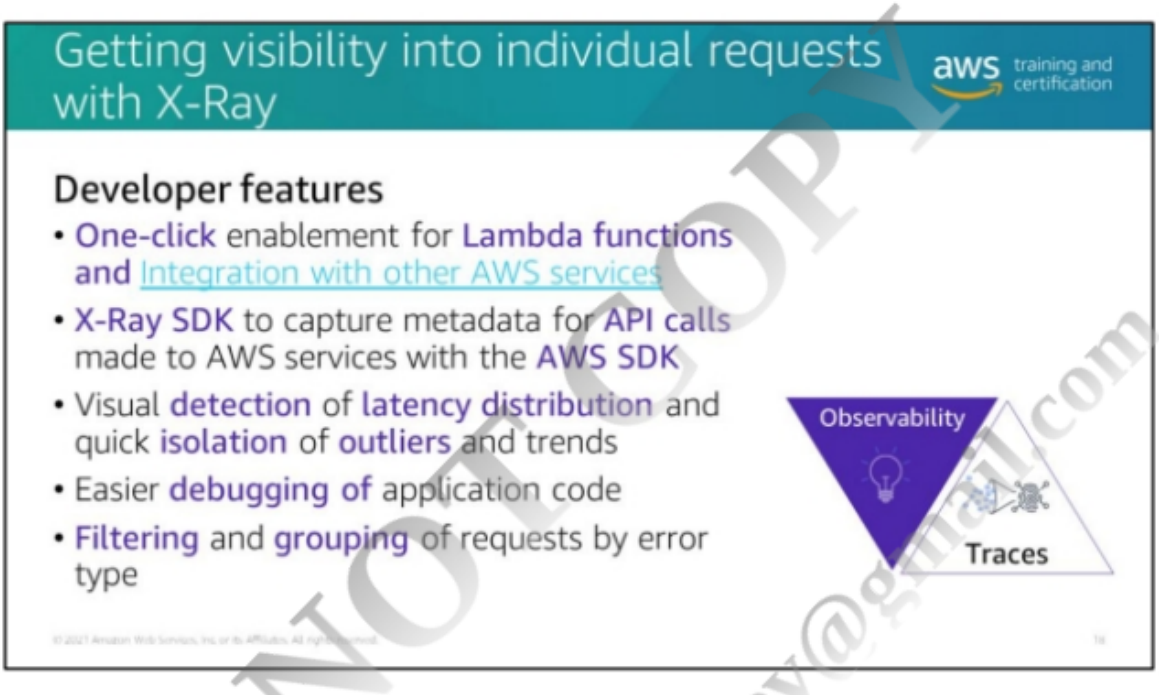


For information about analyzing log data with CloudWatch Logs Insights, visit https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/AnalyzingLogData.html.

For information about supported logs and discovered fields, visit https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/CWL_AnalyzeLogData-discoverable-fields.html.



By default, logs are kept indefinitely and never expire. You can adjust the retention policy for each log group, keeping the indefinite retention or choosing a retention period between 1 day and 10 years. Find the balance between too much and too little. Consider how much you log but also how long you keep those logs.



For information about AWS X-Ray use cases and requirements, visit https://docs.aws.amazon.com/xray/latest/devguide/xray-usage.html.

Other AWS services integrate with X-Ray using one of these integration methods:

- **Active instrumentation** samples and instruments incoming requests.
- **Passive instrumentation** instruments requests that have been sampled by another service.
- **Request tracing** adds a tracing header to all incoming requests and propagates it downstream.
- **Tooling** runs the X-Ray daemon to receive segments from the X-Ray SDK.

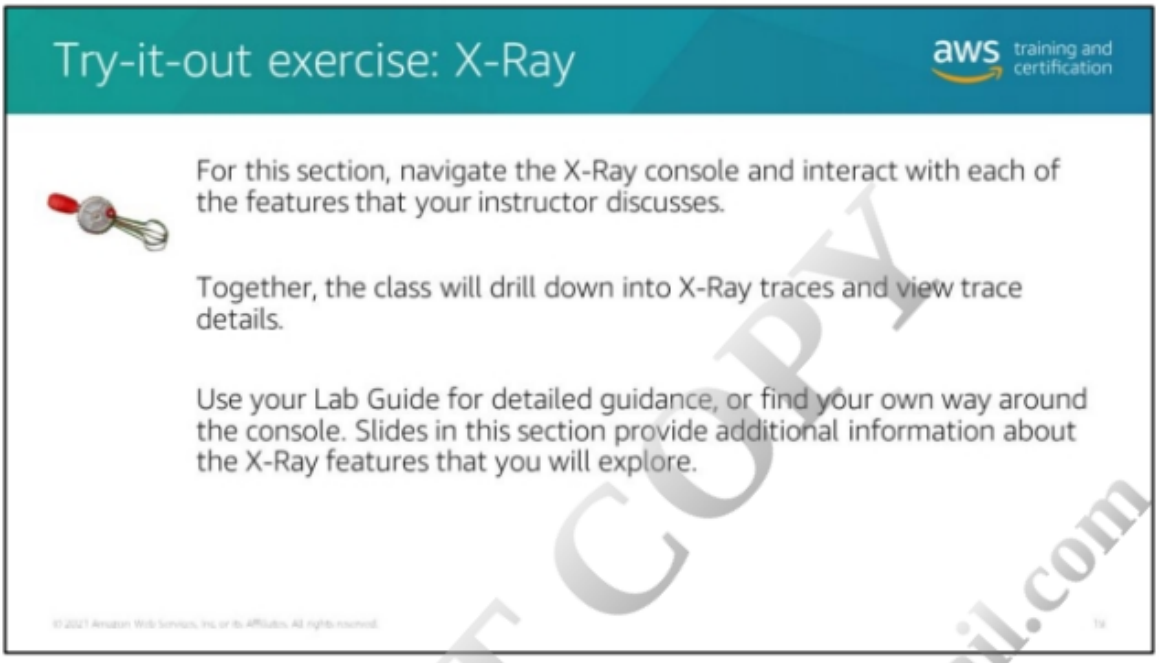
For more information about integrating X-Ray with other AWS services, visit https://docs.aws.amazon.com/xray/latest/devguide/xray-services.html.

For some services like Lambda, you can simply enable X-Ray on the service, and X-Ray will begin to sample new requests and add trace information. This is active instrumentation.

Other services allow you to retrieve the tracing header passed to the service and propagate trace data using the trace ID from the service that generated it. This gives you continuity across the request so that you can follow the trace through downstream services. This is passive instrumentation.

With both active and passive tracing, a new **segment ID** is generated, but with passive tracing the trace ID remains the same. If a trace ID doesn't already exist, no net new trace ID is generated.

You can use the X-Ray SDK to capture metadata for API calls made via the AWS SDK to AWS services

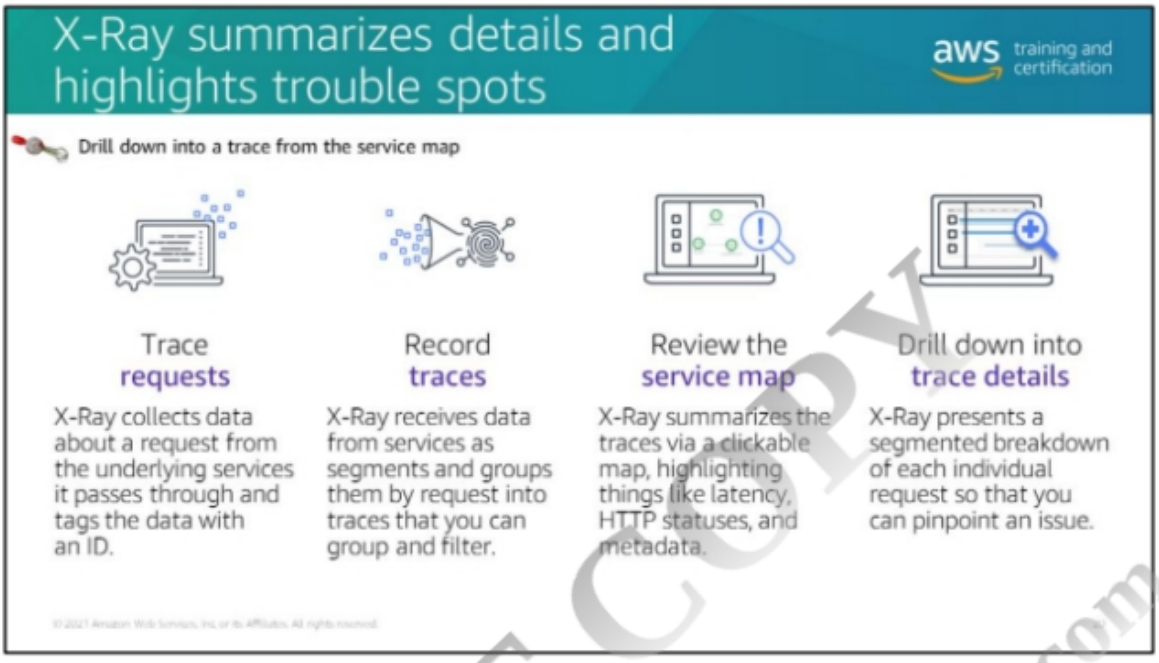


The AWS documentation is a good starting place to walk through X-Ray concepts: https://docs.aws.amazon.com/xray/latest/devguide/xray-concepts.html.

For more information about interacting with traces via the AWS Management Console, visit https://docs.aws.amazon.com/xray/latest/devguide/xray-console.html.

In this section, use the TIO environment to navigate through the X-Ray console and review the features that your instructor is demonstrating. Select the X-Ray service from the AWS Management Console to get started.

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



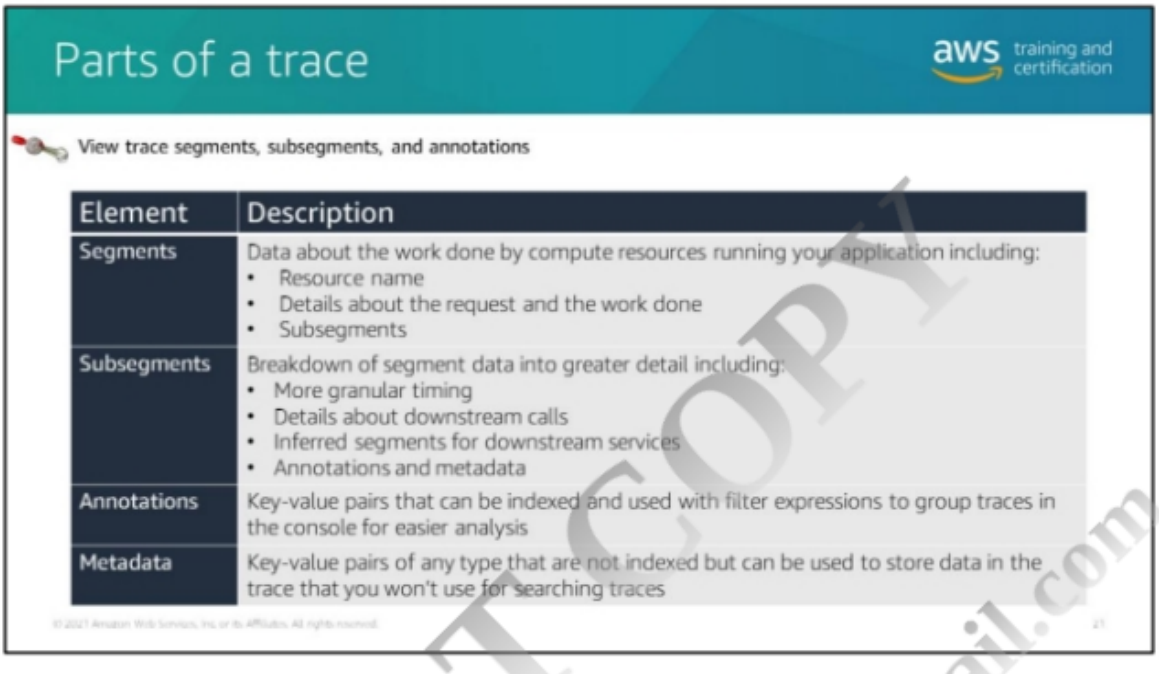
With X-Ray, you can configure services to capture trace data about the requests. For some services, you can directly enable X-Ray.

X-Ray combines the trace data from each service involved in serving a request into a unit called a *trace*.

The X-Ray *service map* gives you an overview of requests moving through the system and highlights trouble spots. From the service map, you can then drill down into a noted trouble spot to review data about each trace segment that X-Ray captured. You can see how long each activity took, and X-Ray highlights segments that have errors.

X-Ray tracks three types of errors in your code and from downstream services:

- **Error:** Client errors (400 series)
- **Fault:** Server faults (500 series)
- **Throttle:** Throttling errors (429 Too Many Requests)



For information about segments, visit https://docs.aws.amazon.com/xray/latest/devguide/xray-concepts.html#xray-concepts-segments.

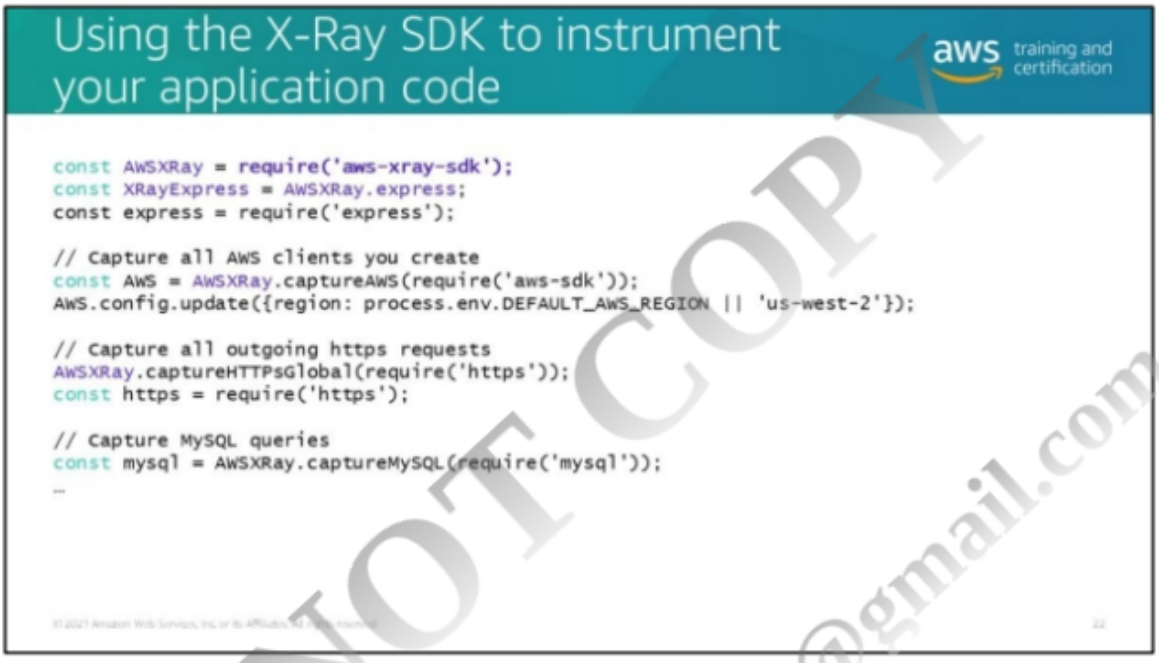
For information about X-Ray segment documents, visit https://docs.aws.amazon.com/xray/latest/devguide/xray-api-segmentdocuments.html.

A *segment* provides the resource's name, details about the request, and details about the work done.

A segment can break down the data about the work done into *subsegments*. Subsegments provide more granular timing information and details about downstream calls that your application made to fulfill the original request. A subsegment can contain additional details about a call to an AWS service, external HTTP API, or SQL database. You can even define arbitrary subsegments to instrument specific functions or lines of code in your application.

For information about annotations and metadata, visit https://docs.aws.amazon.com/xray/latest/devguide/xray-concepts.html#xray-concepts-annotations.

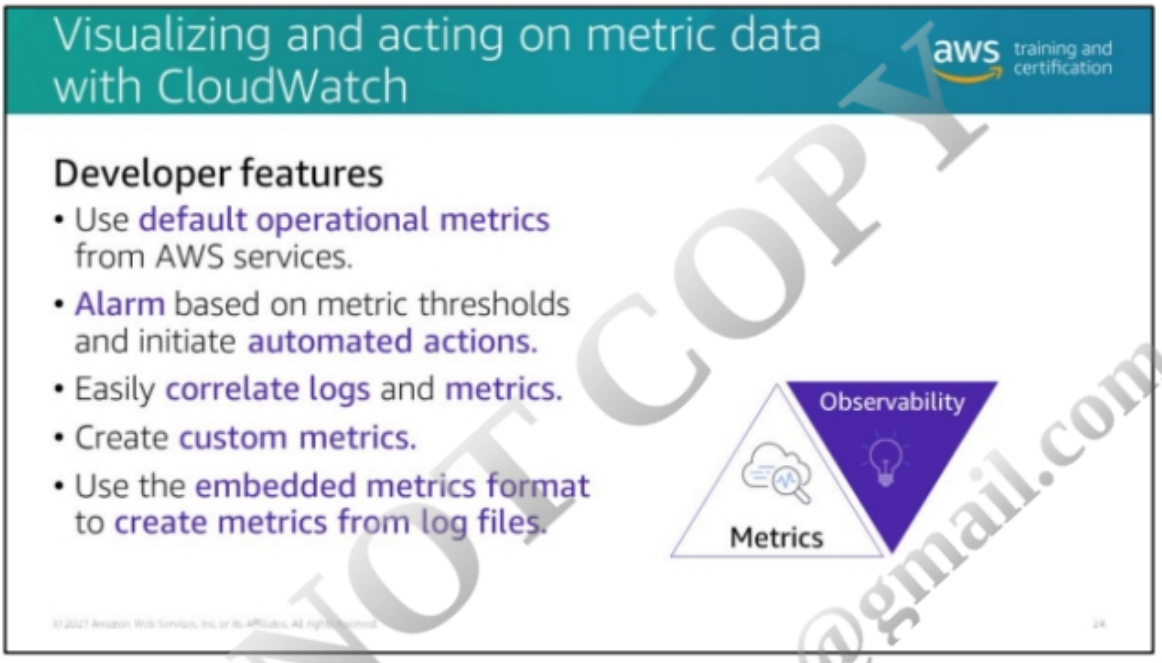
This additional data makes it easier to use filter expressions and look at your traces in logical groups. For more information about using filter expressions to search for traces in the console, visit https://docs.aws.amazon.com/xray/latest/devguide/xray-console-filters.html.



To instrument your application code, use the X-Ray SDK. The SDK records data about incoming and outgoing requests and sends it to the X-Ray daemon, which relays the data in batches to X-Ray.

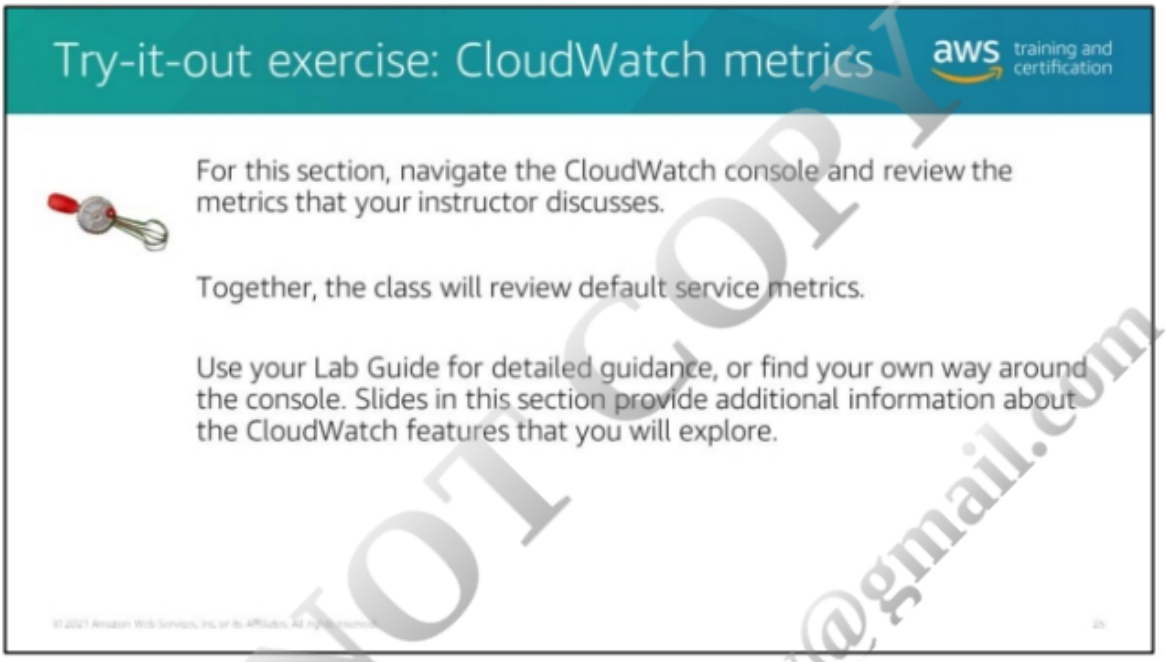
The X-Ray console and X-Ray daemon both use the AWS SDK to communicate with X-Ray. The AWS SDK for each language has a reference document for classes and methods that map to X-Ray API actions and types.

For more information about X-Ray use cases and requirements, visit https://docs.aws.amazon.com/xray/latest/devguide/xray-usage.html.



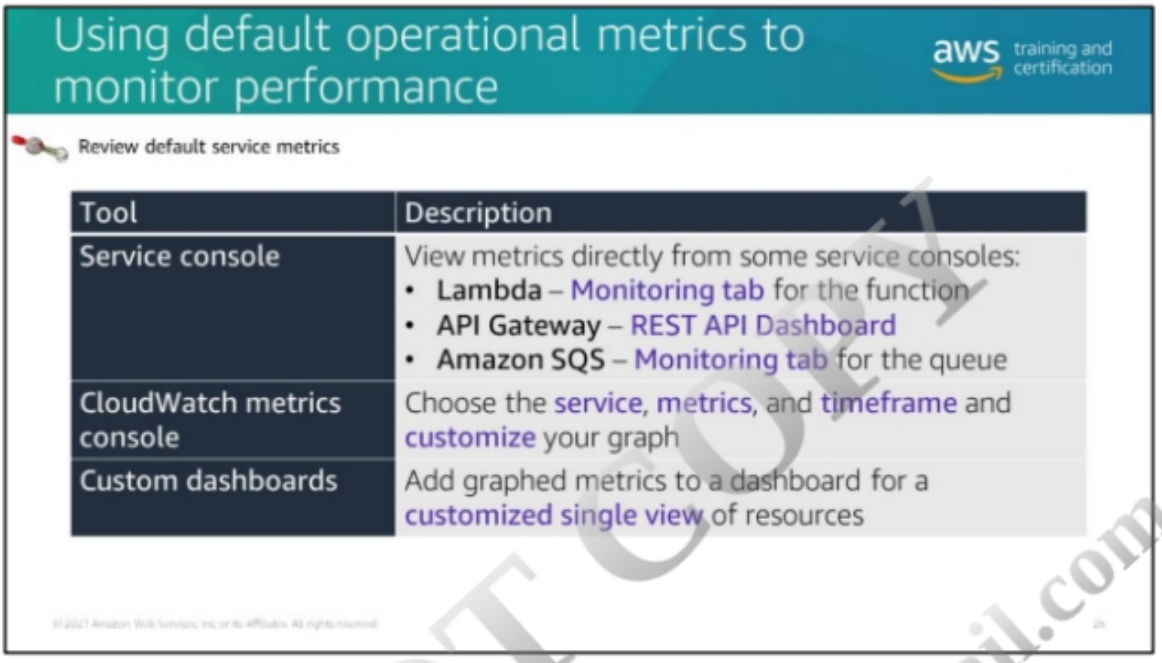
As the CloudWatch Logs section highlighted, you can create metrics based on your log data, and you can easily move between the logs and metrics related to a service via the CloudWatch console and in CloudWatch Logs Insights.

You can create custom metrics to address your business needs (for example, customer activity), and you can set alarms to notify you or take other actions when a threshold is exceeded.



In this section, use the TIO environment to navigate through the CloudWatch console and review the metrics that your instructor is demonstrating. 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 this exercise.



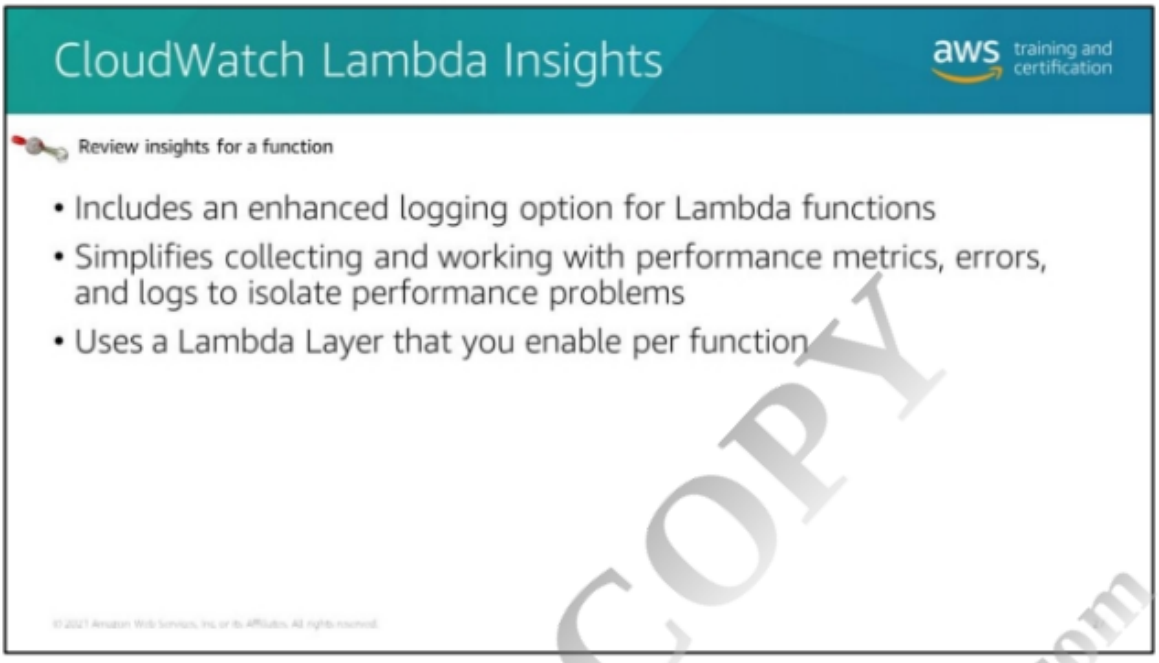
Many built-in metrics are available for AWS services, and some services have metrics dashboards directly on the service console.

When you review metrics from the service console, use the "..." icon next to the metric graph title to jump to the CloudWatch metrics console view of that same metric.

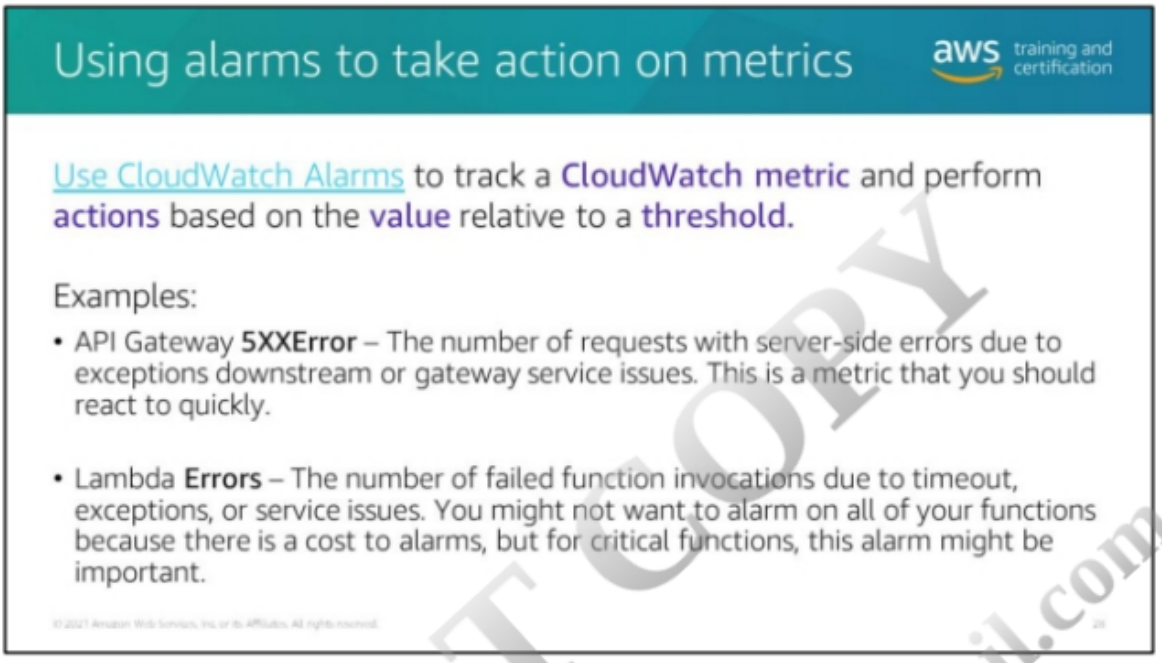
From the CloudWatch metrics console, you can choose a service, specific metrics you want to graph, and a timeframe and can then customize the graph style so that it makes sense for the metric (for example, summary values versus a line chart showing trends).

If you create a view that you want to use again, you can add it to a dashboard. You can add multiple views and views of different services to your custom dashboard and can share dashboards with others.

For more information about using CloudWatch dashboards, visit https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/CloudWatch_Dashboards.html.



For more information about using and configuring CloudWatch Lambda Insights, visit https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/Lambda-Insights.html.



For information about using Amazon CloudWatch Alarms, visit https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/AlarmThatSendsEmail.html.

With alarms, you can track a CloudWatch metric and perform actions based on the value relative to a threshold. For example, you can use an Amazon Simple Notification Service (Amazon SNS) topic to send emails when the alarm state changes, or you can use Amazon EventBridge rules (previously Amazon CloudWatch Events) to take actions based on alarms. You can set static alarms that use a value as the threshold, or you can use anomaly detection to use modeled data to determine the threshold.

For more information about setting Amazon SNS notifications, visit https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/US_SetupSNS.html.

