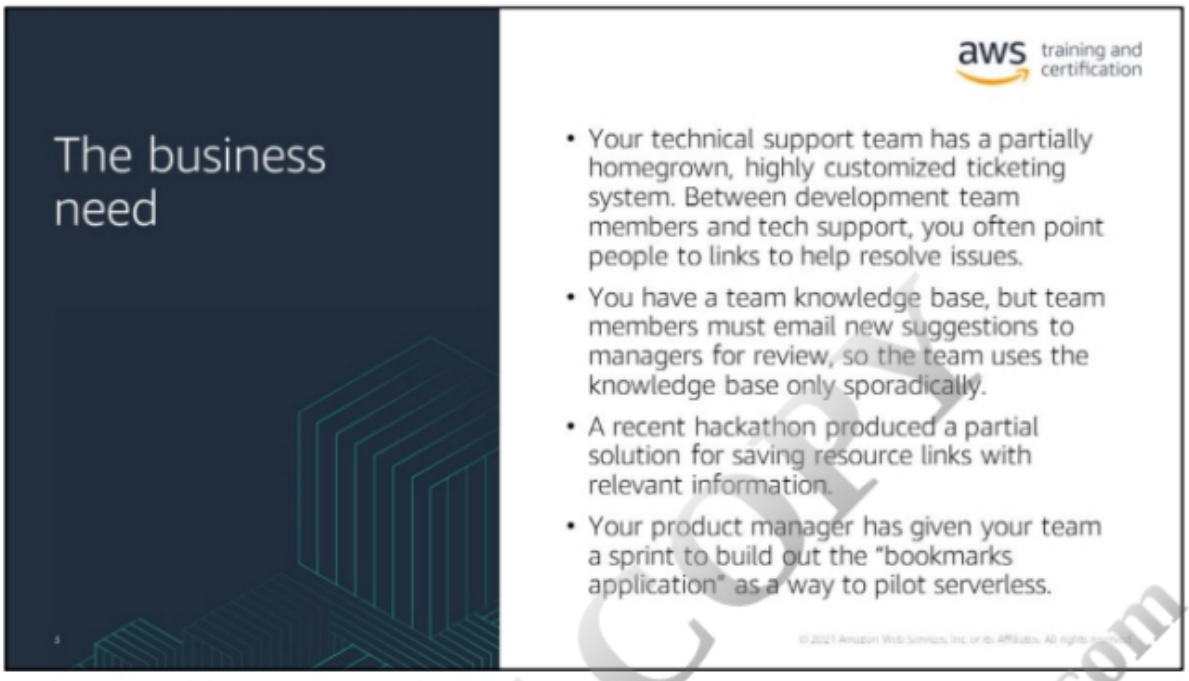
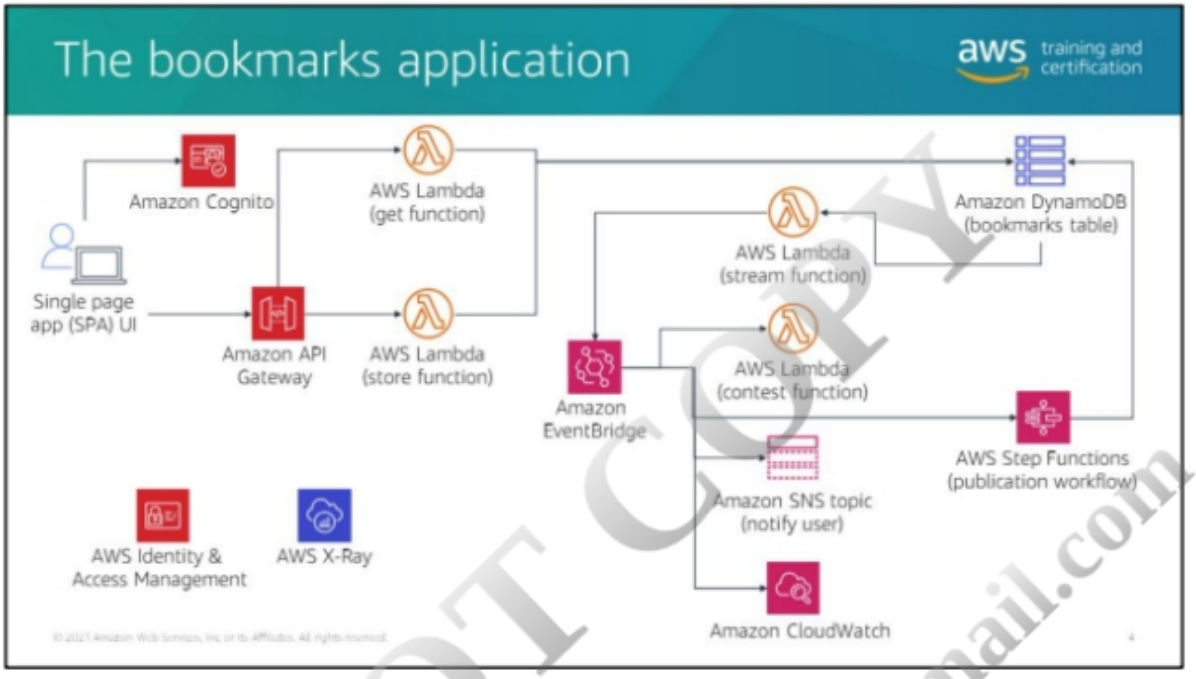
Corrections, feedback, or other questions? Contact us at https://support.aws.amazon.com/#/contacts/aws-training.

Contents

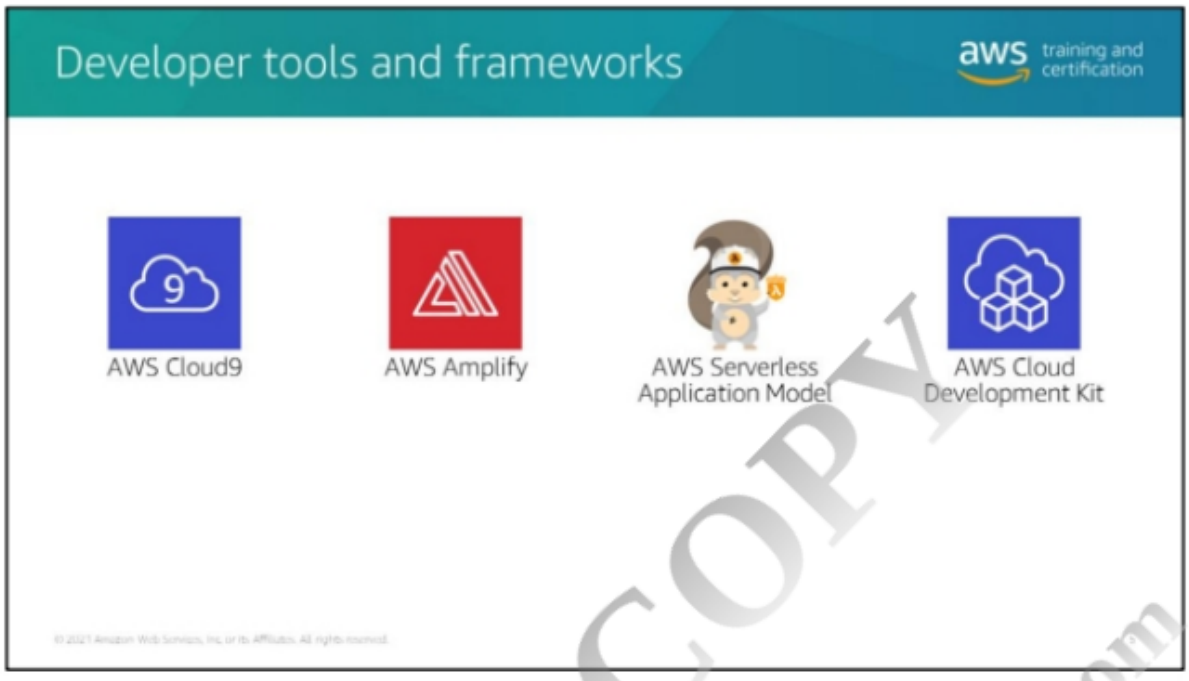




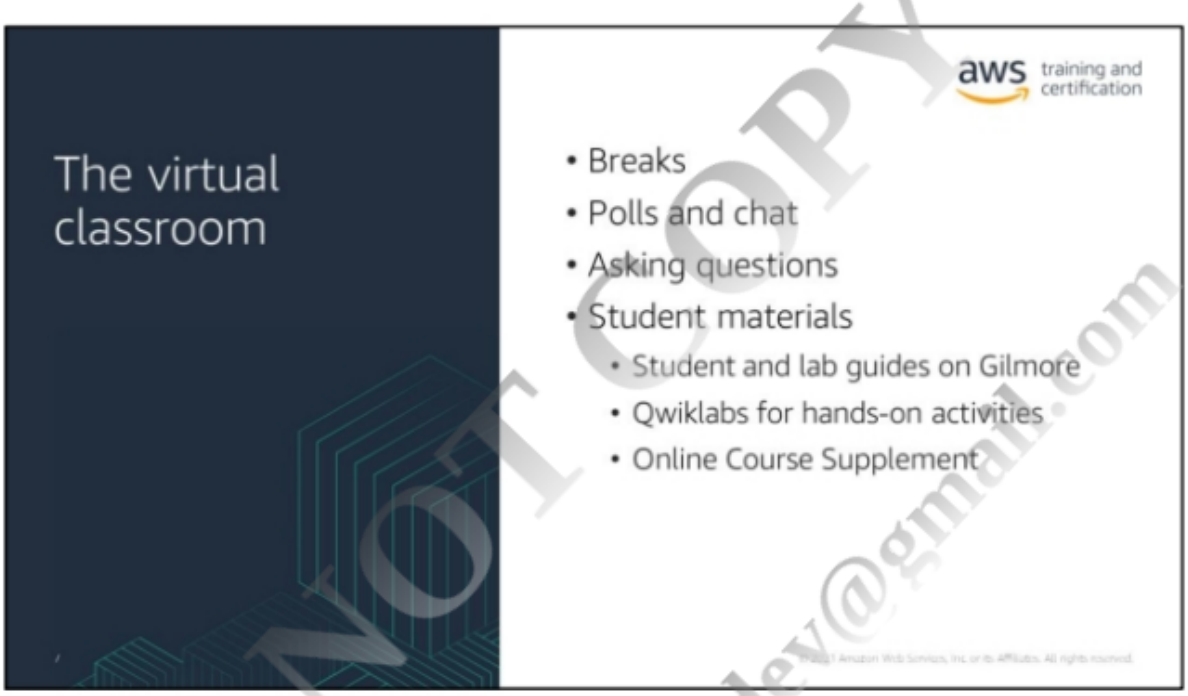
This diagram depicts the components of the application you build in this course.

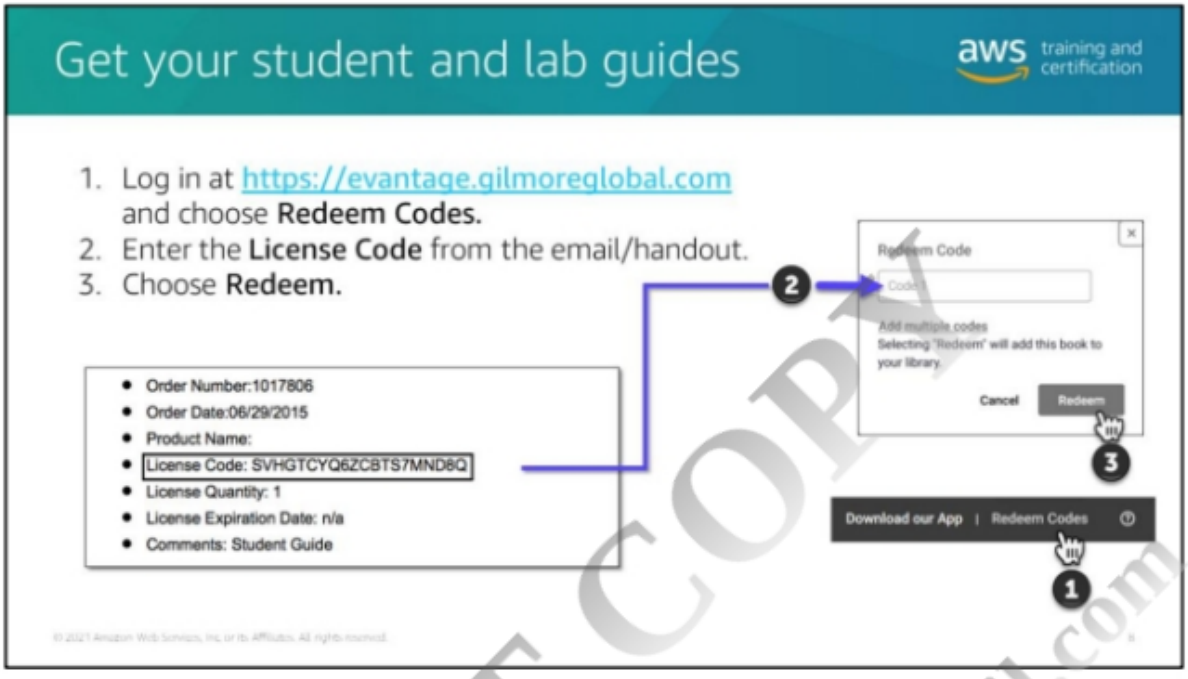
You use Amazon Cognito for signing in, Amazon API Gateway for API management, AWS Lambda functions for your business logic, Amazon DynamoDB for your data store, Amazon EventBridge for routing events, AWS Step Functions to orchestrate a workflow, Amazon Simple Notification Service (Amazon SNS) to notify subscribers of event, and Amazon CloudWatch for logging and metrics. You also use AWS Identity and Access Management (IAM) for security control throughout the application and AWS X-Ray for traceability.

At the end of the course, you should be able to speak to how and why you use each service in a serverless application.



You develop the application using an AWS Cloud9 development environment, and you deploy your user interface using AWS Amplify. You deploy your backend services with the AWS Serverless Application Model (AWS SAM) and create your deployment pipeline with AWS Cloud Development Kit (AWS CDK).

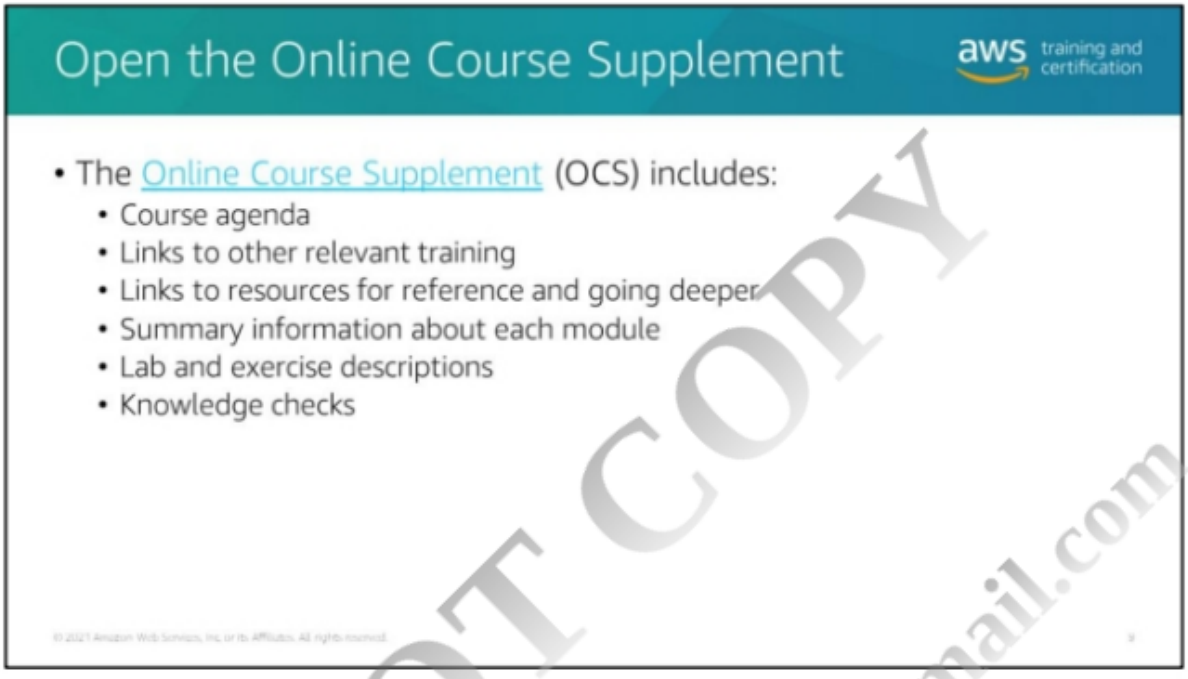




To login to access your student and lab guides, go to https://evantage.gilmoreglobal.com.

Your student guide follows the instructor presentation closely and provides additional details for reference on topics presented in each slide.

Your lab guide provides step-by-step instructions for completing the hands-on activities used during the course. These instructions are the same as those available to you online while you are performing hands-on activities.



The Online Course Supplement (OCS) is an online supplement to the student guide. In it, you'll find a full course agenda as well as links to resources and summaries for each module.

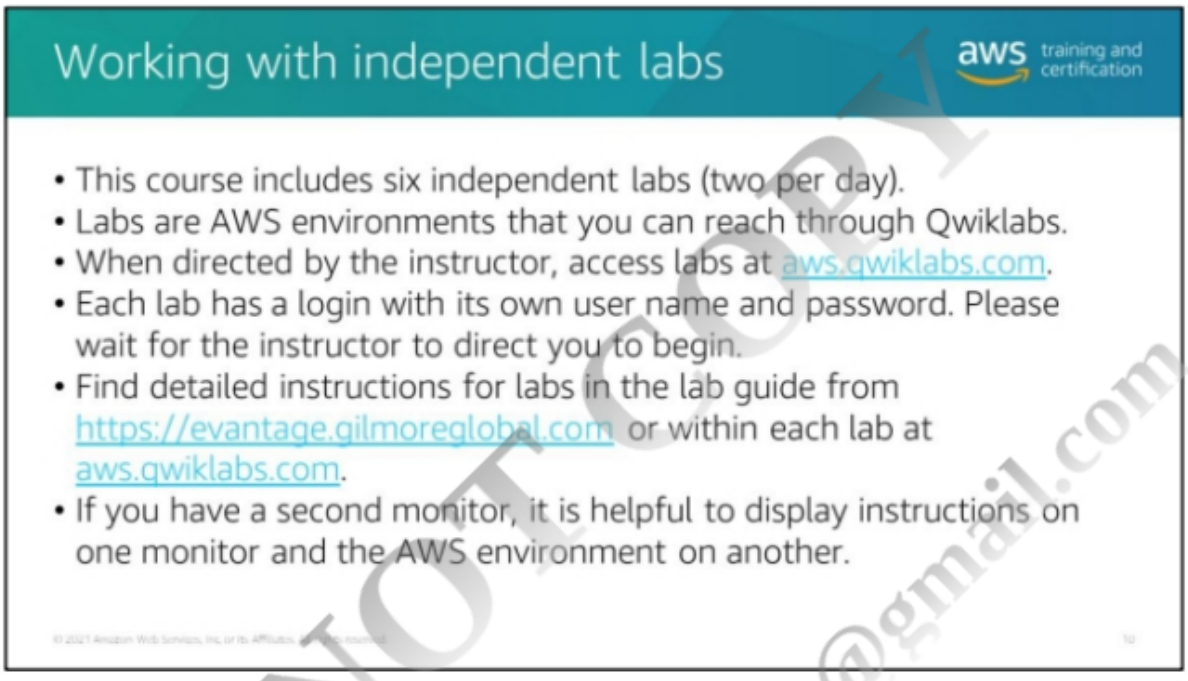
Many of the links included in the student guide are also available in the OCS so that they are easier to find when you need them later.

You'll also find knowledge checks to try on your own or review with your instructor.

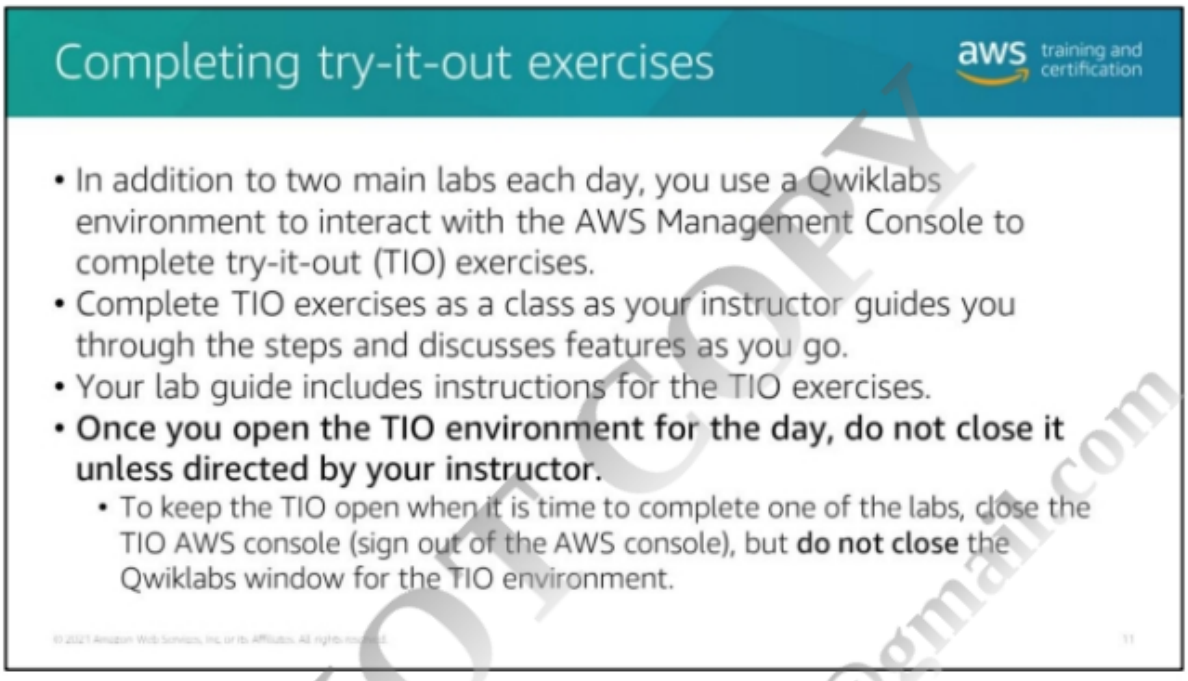
To open the Online Course Supplement, go to https://www.aws.training/Details/eLearning?id=56189.

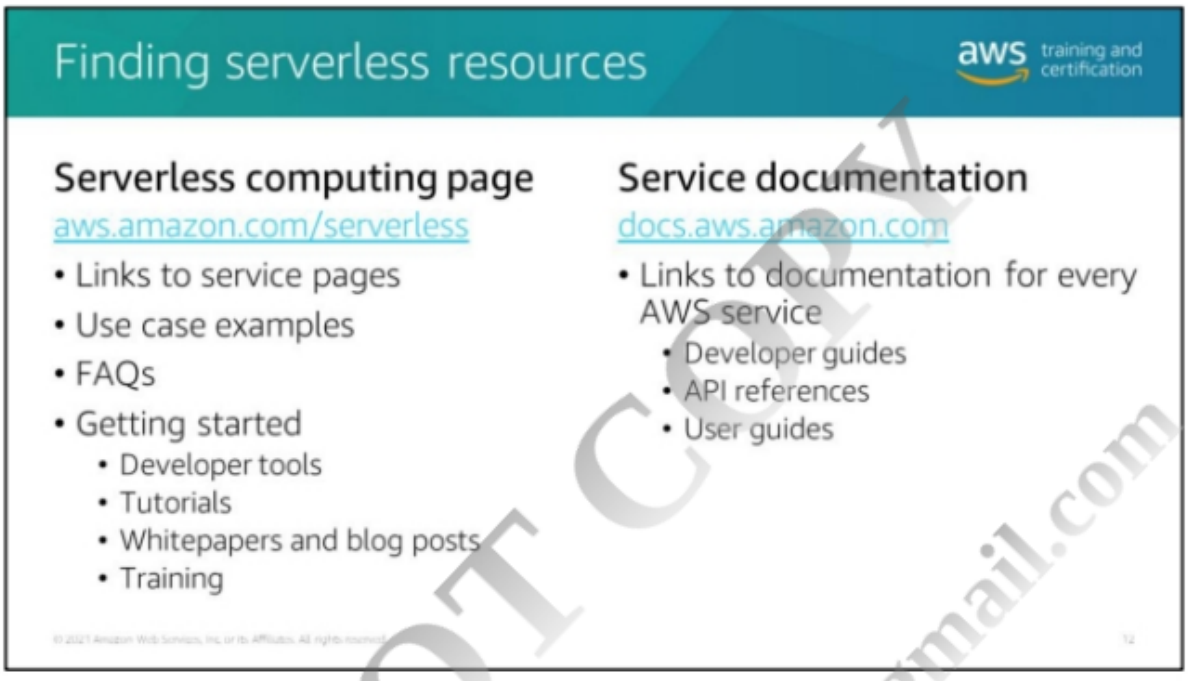
Use the OCS to:

- Navigate quickly between module topics and lab descriptions by day
- Check your understanding of course topics
- Easily find links to resources used throughout the course



Here is the link to Qwiklabs: https://aws.qwiklabs.com/.

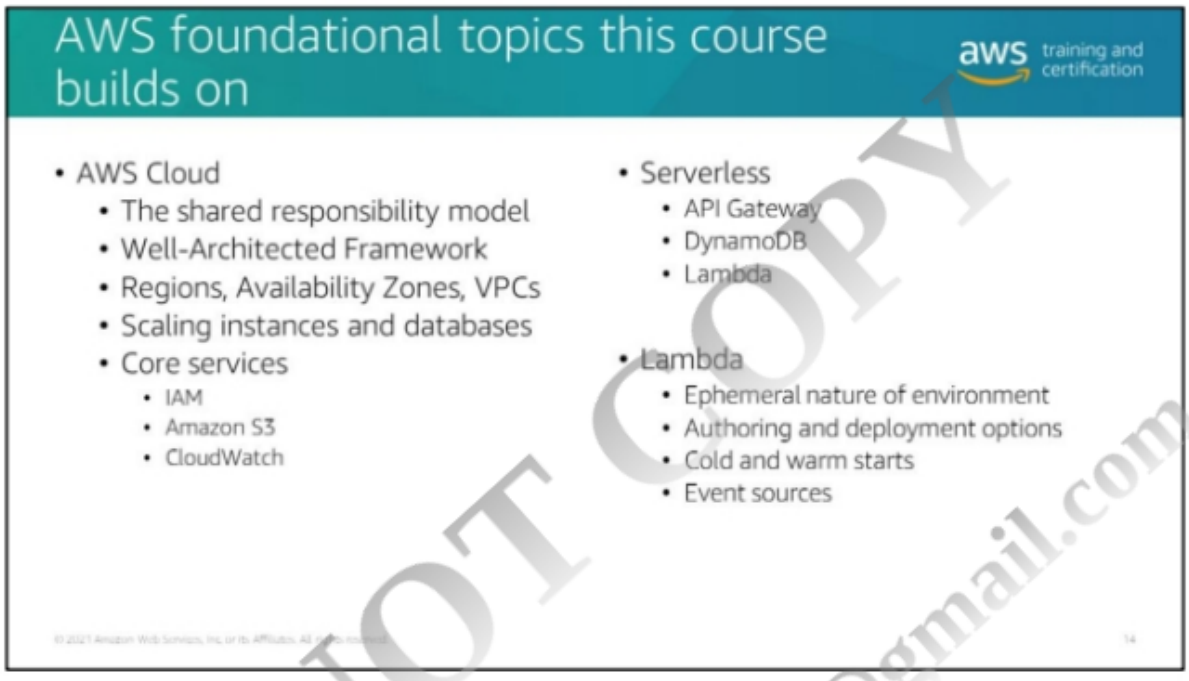




The service pages on aws.amazon.com and guides on docs.amazon.com are an excellent place to start working with any service.

Use the serverless computing page as a starting point to learn about what's available and to go deeper into specific services and use cases.

Bookmark the developer guide or user guide for each service you work with. You will go back to these guides again and again.

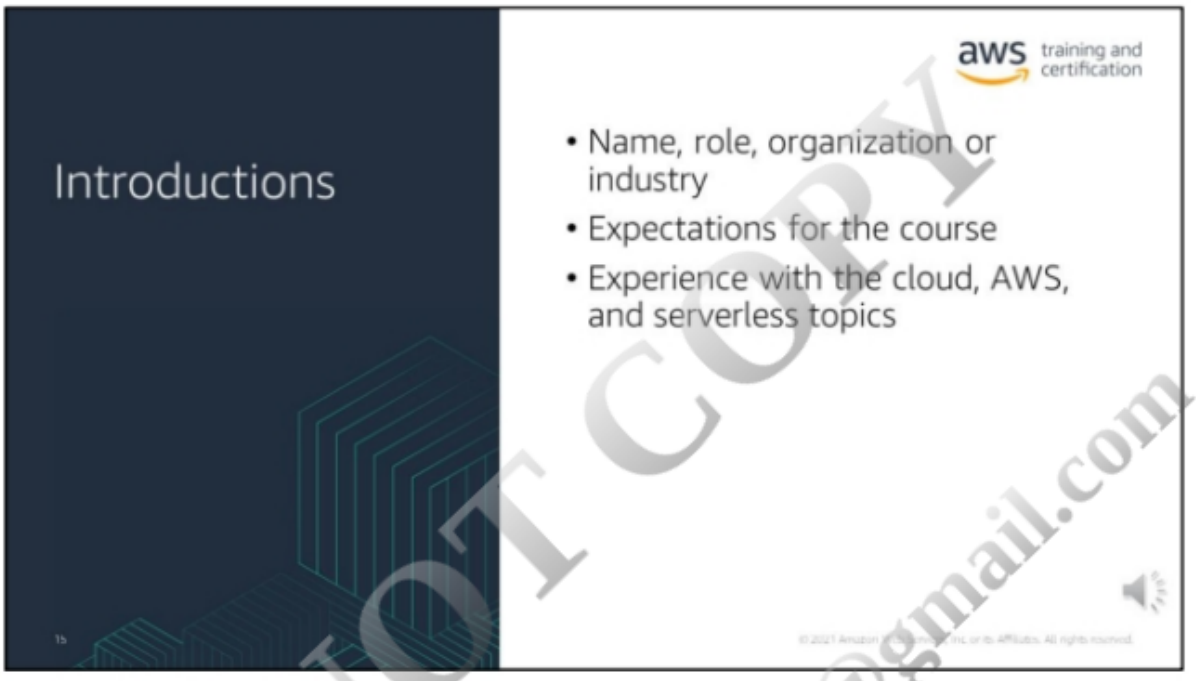


Students are expected to have an existing understanding of cloud architectures.

This course also assumes exposure to AWS services that are introduced in AWS Cloud Practitioner Essentials and expanded upon in Architecting on AWS.

Although the course spends time using Lambda with other serverless services, it does not dive deep into all of the characteristics of the services. This course assumes that you have completed the free digital serverless offerings on Lambda, API Gateway, and DynamoDB or or have equivalent knowledge.

The Online Course Supplement includes links to these three courses and other resources to help you review the prerequisite knowledge.



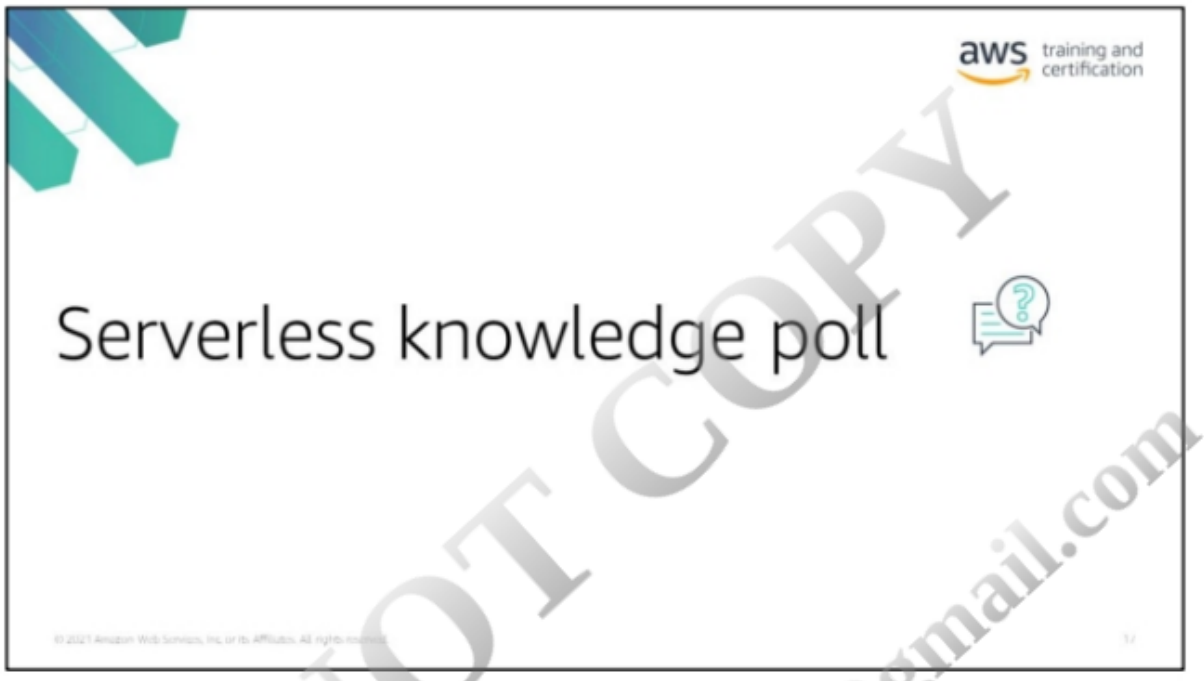
The course often references AWS documentation and resources to help you build your serverless application.

Bookmark the pages and resources for your use after the course ends.

There are a lot of services to talk about, and you might use them many different ways based on your own organization and the applications you need to build.

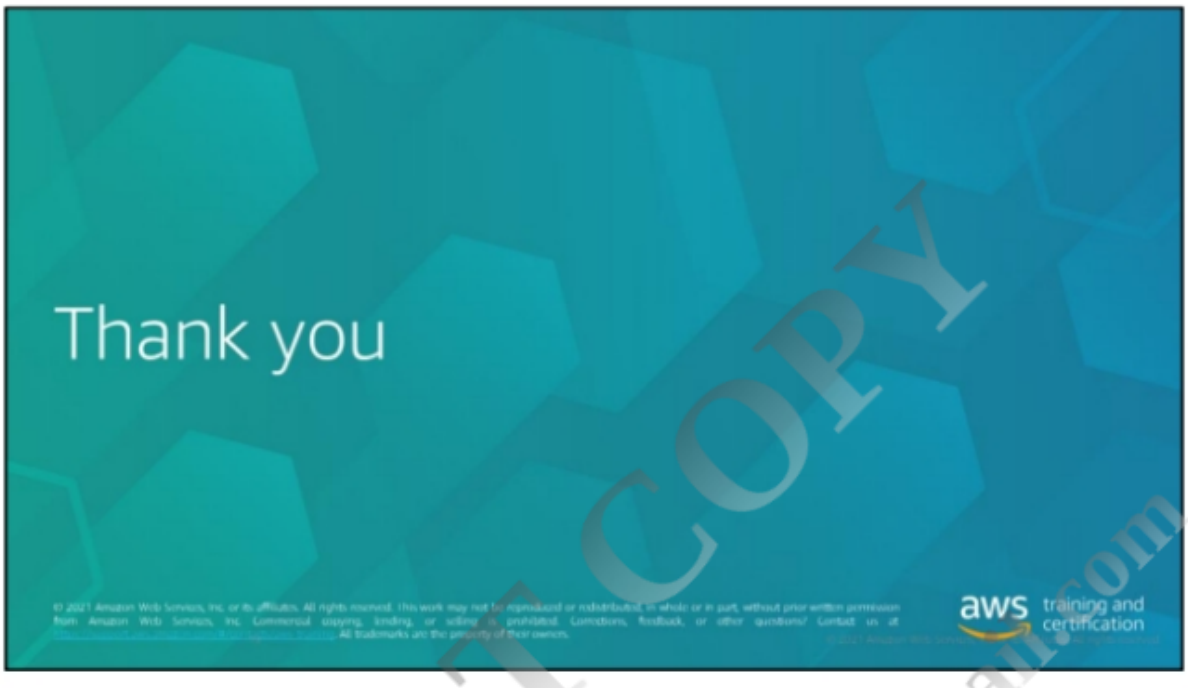
This course is designed to maximize the time that you are actually using the tools versus receiving information about them. This means that the course moves fairly quickly through the features and details of many of the services.

In addition to these primary resource links, many course sections include "going deeper" options on the topics that the Online Course Supplement covers. Use these options to continue your serverless journey beyond the classroom.

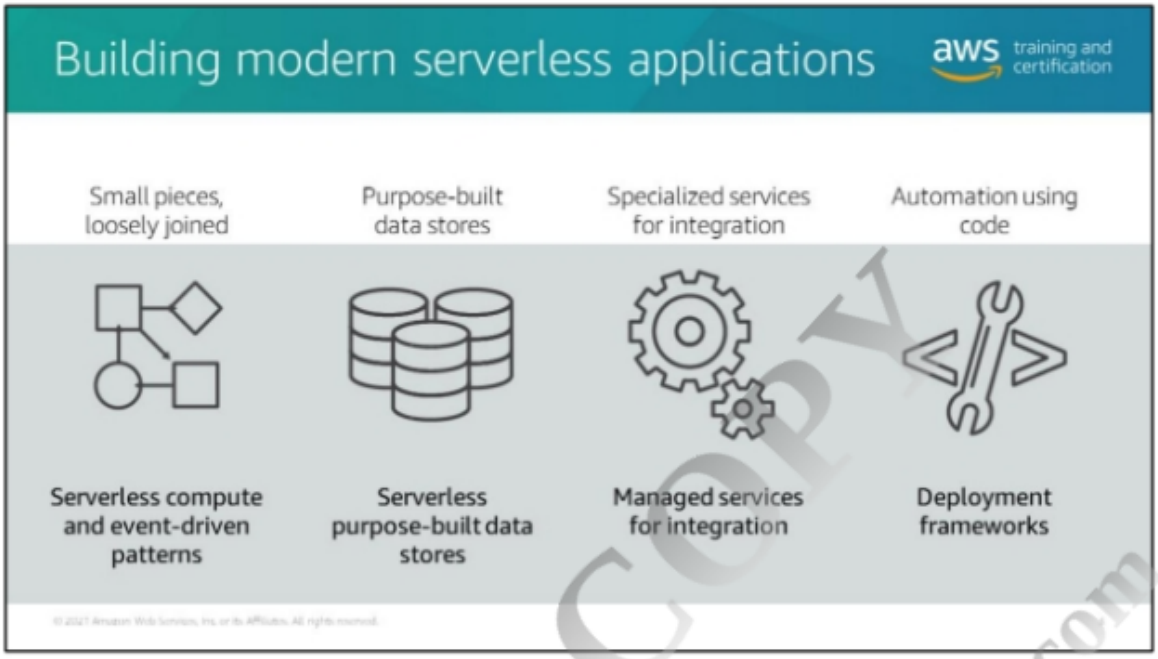


The full course agenda is in the Online Course Supplement.

You take formal breaks guided by your instructor,.



In this module, you'll get into the mindset of a serverless developer. All the things you know about building good software still apply, but you might need to shift your thinking a bit in regard to how you build, integrate, test, and deploy what you build. Let's start with a high-level review of what you as developer want to achieve.



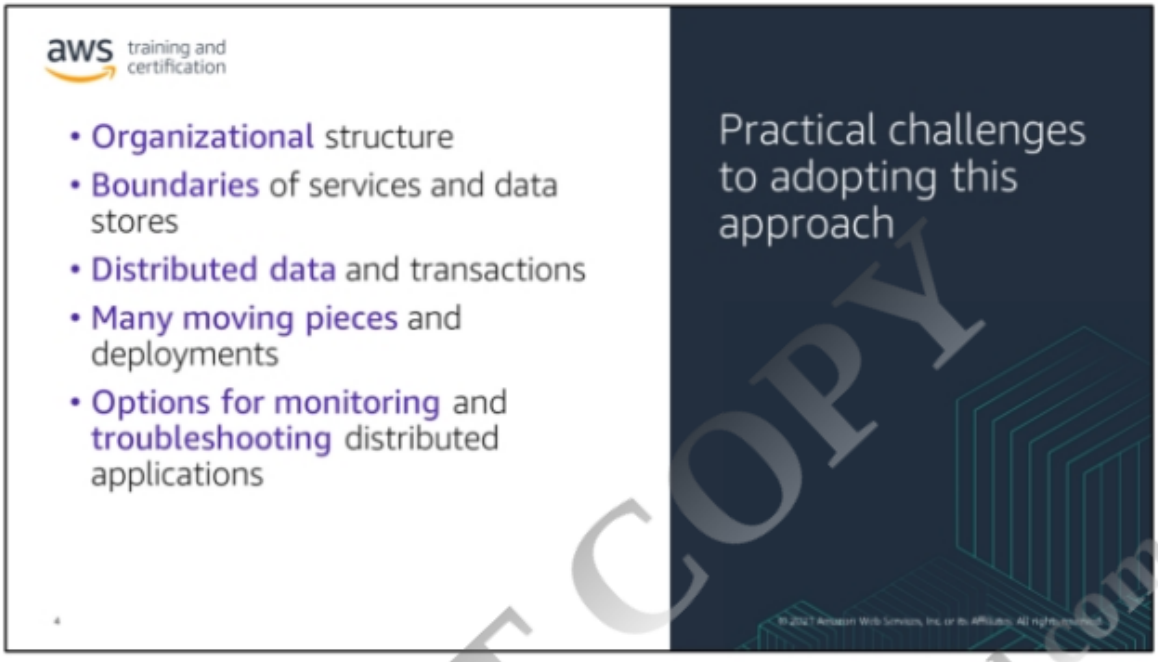
From a builder's perspective, the abstraction of infrastructure into the cloud from virtual servers to containers and now to serverless has coincided with a move away from monolithic applications and a move toward service-oriented architectures and microservices.

The ideas of distributed applications and microservices aren't new, but the evolution of cloud services makes it easier to build applications that follow these principles:

- Use small, loosely joined pieces and purpose-built data stores to provide the flexibility to make changes without impacting other parts of the system, scale and deploy pieces independently, and incorporate reversible decisions.
- Use specialized integration services to increase the speed of delivery and lower the effort of integrations between services.
- Write infrastructure as code to automate as much of the deployment process as you can. This makes it simpler to replicate environments and deploy more frequently with minimal overhead and greater confidence.

Serverless (https://aws.amazon.com/serverless/) is an efficient approach to modern applications.

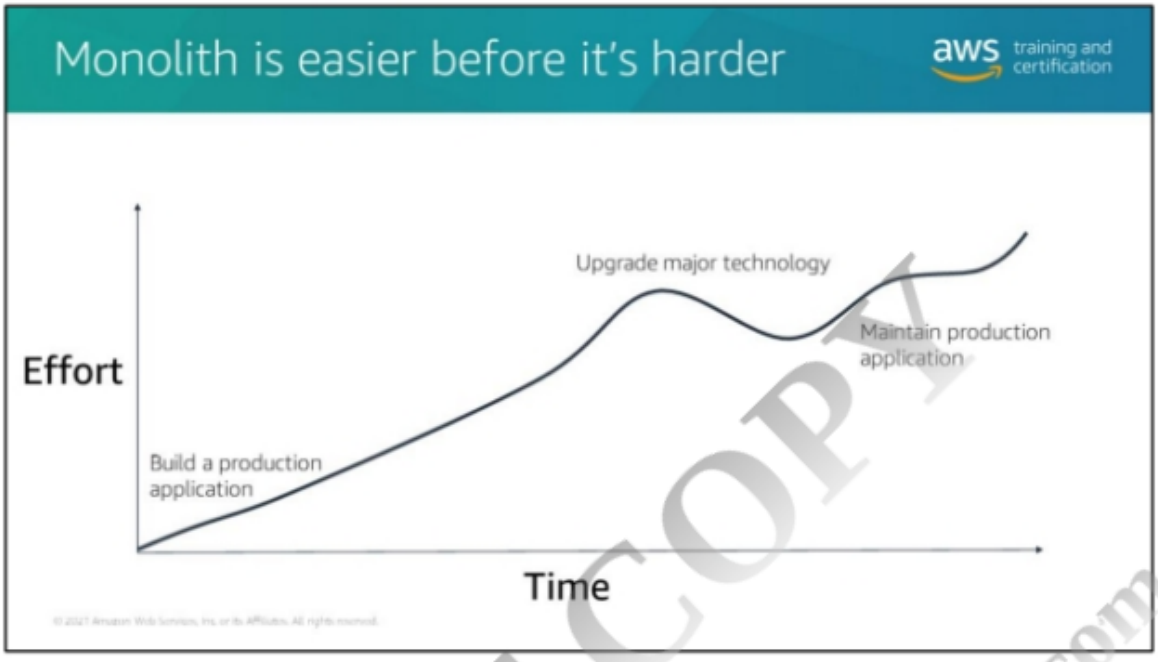
- Serverless compute services let you focus on writing business logic while someone else handles scaling and high availability.
- Purpose-built serverless databases let you choose the features and capacity that suit the type and volume of data each service needs.
- Managed services provide API proxy, messaging, and orchestration and have pre-built integrations with AWS Lambda and other Amazon Web Services (AWS) services. This lets you connect services using asynchronous event-driven patterns to achieve decoupling and flexibility while reducing the undifferentiated lifting of building the integrations to connect your components.
- Serverless deployment frameworks make it easier to use infrastructure as code for deployments and automate the deployment pipeline.



For many organizations, it means changing how you think about your code, structure your teams, deploy your software, and operationalize your applications.

The first three bullets are critical to creating modern applications that your organization can actually maintain and to yielding the benefits of service-oriented architectures and microservices. This course does not cover these topics in detail, but the Online Course Supplement (OCS) provides links to take you deeper on the topics of defining service boundaries and considering all of the impacts of distributed data. A link to the Advanced Developing on AWS course covers these topics in more detail.

This course addresses how you might use AWS services to connect many pieces together and rely on managed services to lighten the heavy lifting of integrating your components or microservices. This course also reviews AWS options for monitoring your serverless applications.



The challenges you just reviewed can make it difficult for organizations to move away from a monolithic approach to building applications.

With a monolithic application, things start out easier. It's simpler to build and simpler to test, and there's only one artifact to deploy. For some less complex applications, a monolithic approach may be the best option for a very long time.

For complex applications that grow over time, the interdependencies among components and the coordination that the development teams need to test and deploy anything become harder and may actually decrease developer productivity.

You've got one technology stack that you have to develop within. You have to hire and train developers in that technology and can't take advantage of skills in other languages.

Any change to the technology stack becomes a major project.

You have a single database. The most demanding use case drives your cost and capacity.

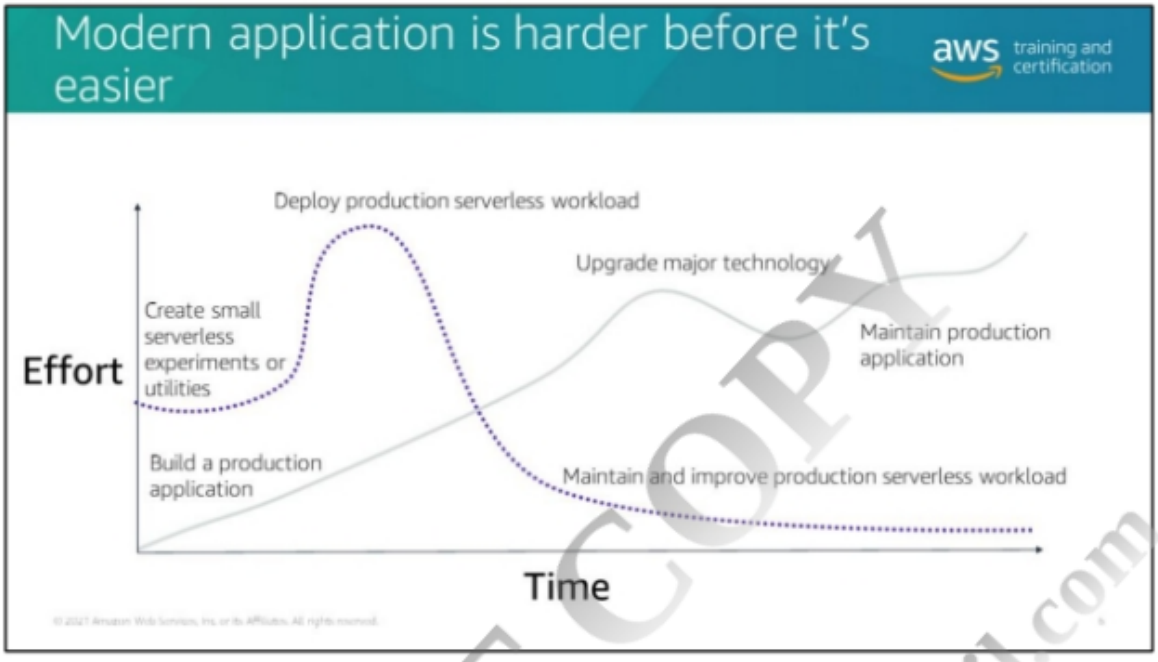
Performance for operations that aren't suited to the database or data structures will keep getting tuned to eke out better performance but will reach a limit that it becomes impossible to get beyond without a redesign.

From an operational standpoint, you have many developers who are all pushing changes through a shared release pipeline, which causes frictions at many points of the lifecycle.

- If you want to upgrade a shared library to take advantage of a new feature, you need to convince everyone else to upgrade at the same time.
- If you want to quickly push an important fix for your feature, you still need to merge it with everyone else's in-process changes. This leads to "merge Fridays" or, worse yet, "merge weeks" when all the developers have to compile their changes and resolve any conflicts for the next release.
- After development, you also face overhead when you're pushing the changes through the delivery pipeline. You need to rebuild the entire application, run all of the test suites to make sure there are no regression issues, and redeploy the entire application.

Every new feature adds to the complexity of the coordination, deployment, and ongoing operations effort. Technical debt stacks up, and it becomes nearly impossible to untangle code.

To give you an idea of this overhead, before migrating to a microservices approach, Amazon had a central team whose sole job it was to deploy its monolithic application into production.



With a modern serverless application, the effort to get started is higher.

Moving to the modern application characteristics described requires a significant surge in effort as everyone learns and make mistakes. Often everyone is new to the approaches required.

Moving from experiments and utilities into full-blown production applications requires a surge of effort that includes both technology and organizational changes. The energy that it takes to get over that initial hurdle can hold organizations back unless there is some initiating event or tipping point.

After the initial learning curve, the agility that this approach provides starts to accelerate the speed with which you can put out new features and the effort it takes to keep them running.

- Scale individual services independently.
- Choose the best technology stack and data store for each service.
- Deploy changes independently as often as desired.
- Let teams focus on specific services and innovate within that service.

Finding a serverless learning project

aws training and certification

- Look for opportunities to introduce serverless through common use cases.
- Pick something authentic, iterate on it, learn from it, and expect to keep working on it in production.
- Expect to change your mind about what you thought was the best solution when you started. Keep learning and updating your approach.
- The OCS includes links to resources to help you get started.

This course does not dive deep into the right way to design and group the functions and services that make up a serverless application, but let's take a few minutes to talk about the primary things that you have to plan.

- In its simplest terms, a service should do one thing well.
- You want small, easily identifiable pieces with well-defined interfaces.
- Your structures should represent the business. Service boundaries equal business boundaries, and related behaviors sit together.
- Your application is a cohesive system made of many small parts that work together.

In the food truck, you do it all. You take the order on paper, take cash-only payments, make the food for each order, and keep track of inventory. You've got a grill, a fryer, one small oven, and one cold storage, and you can reach them all while standing in one place.

Expansion so far has meant hiring a couple of other people to help on busy days, and everyone does a bit of everything. You have very informal communications methods. It works well, but it's not really scalable.

Now you need to adapt to an environment that handles more customers and more product lines.

What are some factors that will influence how you organize the work?

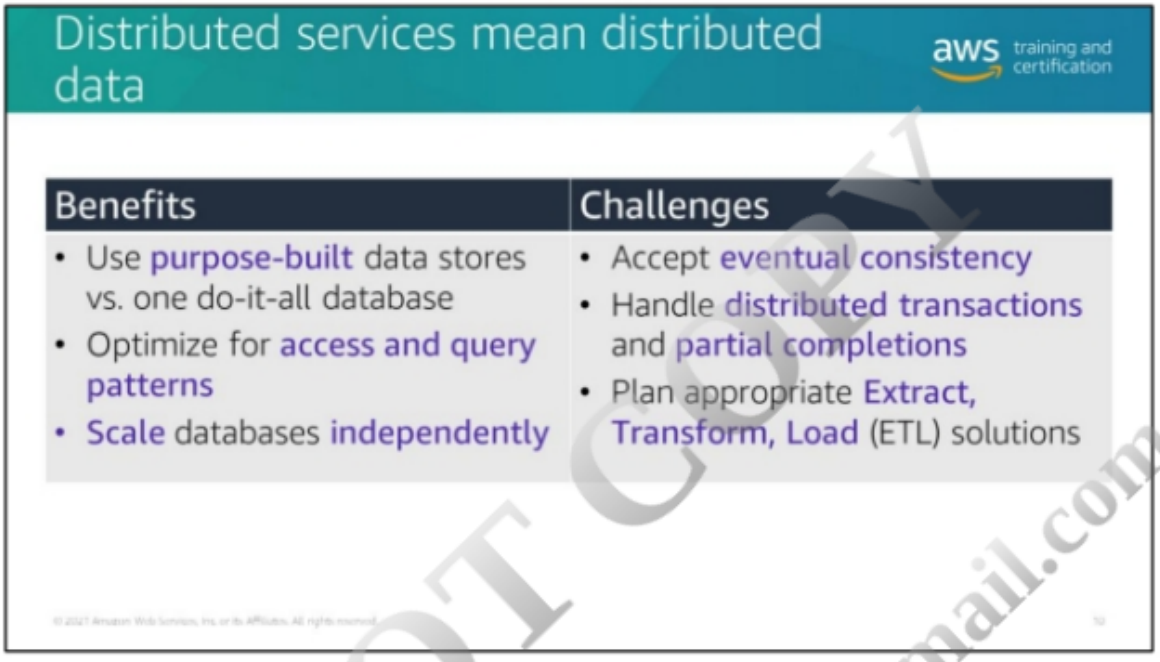
What are some boundaries and groupings to consider?

Business boundary examples:

- Organizational structure
- Permissions
- Natural separations between entities
- Transactional attributes

Technical boundary examples:

- Scaling requirements
- Shared deployment criteria
- Synchronous versus asynchronous processes
- Replaceability
- Data needs (for example, data type, access patterns, and persistence)



A key benefit of distributed data is being able to choose a purpose-built data store that makes sense for the service instead of settling for one do-it-all database. This lets you match each data store to the access and query patterns to optimize for that pattern. It also lets you scale database choices independently rather than buy capacity for the most demanding component.

The challenge is that now you don't have one database to read from and write to; you have a lot of distributed stores to deal with and must find ways to share data as needed. This means you have to accept eventual consistency as described in the CAP Theorem.

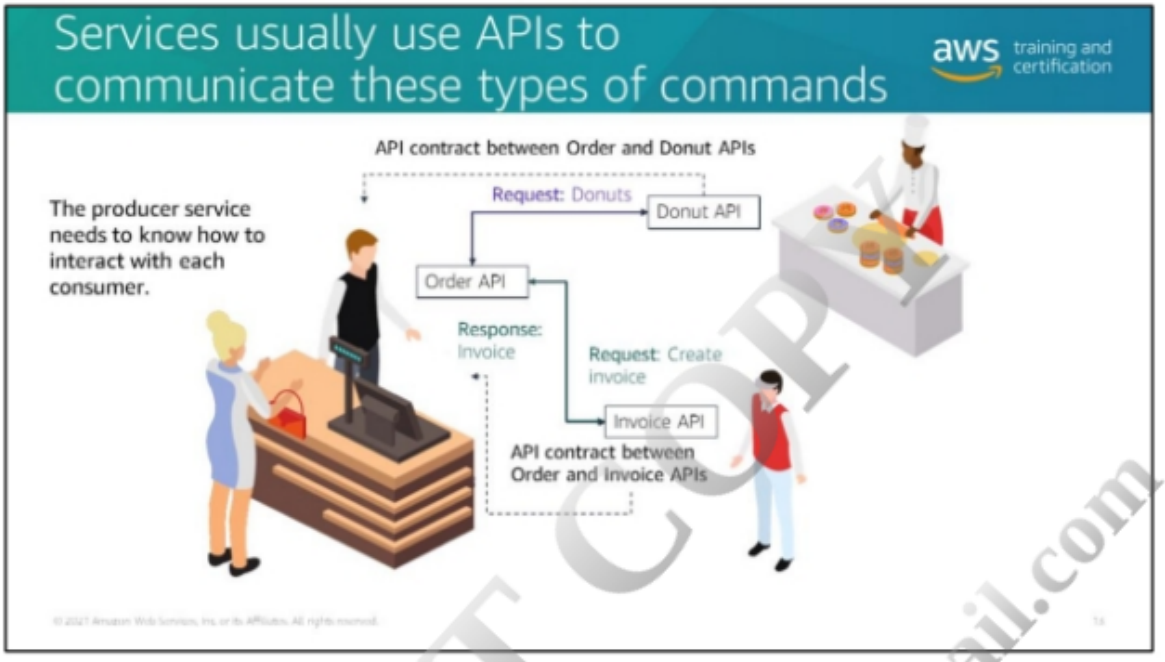
You might need to use unfamiliar methods to deal with transactions that cross multiple data stores and in particular transactions that fail part way through. You also have to determine how you are going to share data that gets written to one data store with other components that need it: for example, transactional data that is needed for reporting.

Reference:
"Perspectives on the CAP Theorem," Seth Gilbert, National University of Singapore, and Nancy A. Lynch, Massachusetts Institute of Technology. Retrieved from https://groups.csail.mit.edu/tds/papers/Gilbert/Brewer2.pdf on November 20, 2020.

Now that you're thinking in terms of microservices and all these smaller bits of work, you need to think about how to connect them in a way that best serves the application's purpose and best suits your goals to limit time spent on busy work.

With this approach, each command is explicitly issued to a specific recipient for a response.

In this example, the counter clerk or manager takes the order, directs the cashier to ring up the order and create the bill, and tells the baker or chef to prepare the order.



The communication between the services may be synchronous or asynchronous.

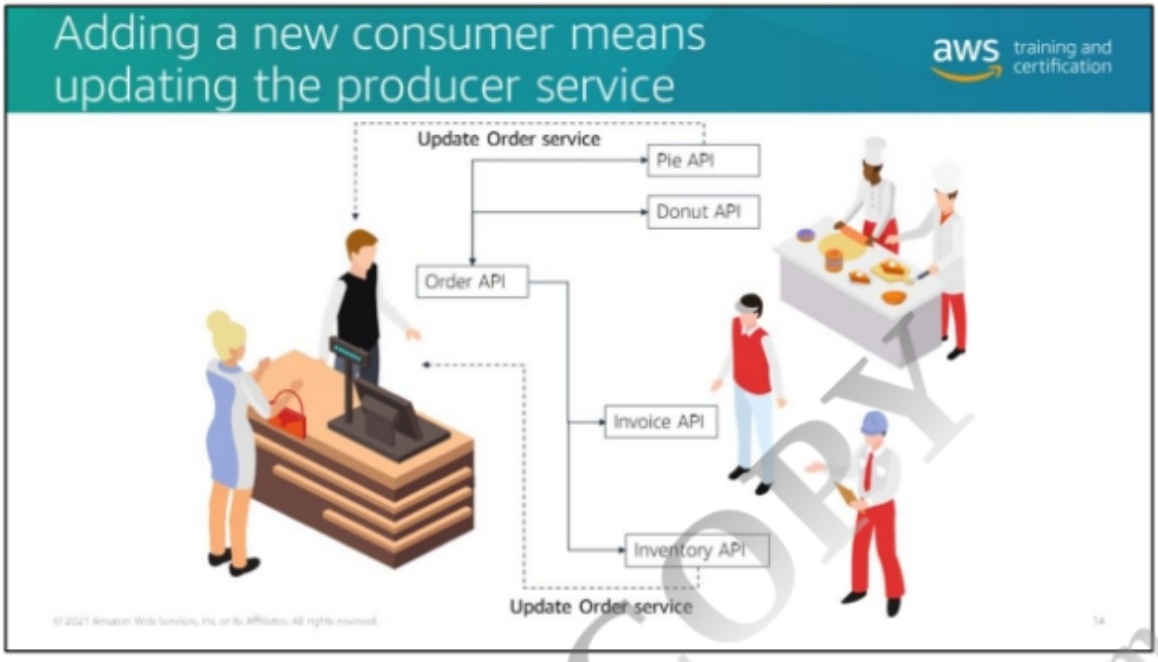
The following might be a synchronous example: When the order service sends a request to the invoice service, the invoice service creates the invoice and immediately responds with the completed invoice for the customer. The order service doesn't continue until the invoice service completes its work and returns the response.

The following might be an asynchronous example: When the order service requests donuts from the donut API, the API doesn't reply with the completed donuts. There are downstream steps that have to happen to retrieve the donuts from the shelf or perhaps make the donuts to fulfill the order. The order service doesn't have to wait for those steps to continue with its next order.

Note that the order service probably receives an acknowledgement from the donut service right away with some identifier that it can use in the order process to be able to check when the donuts are ready. The work that completes the order is happening asynchronously and doesn't prevent the order service from moving on to the next order.

The APIs provide the contract between the product and the consumer of the service. This establishes how the producer communicates with the consumer, what information will be passed between them, and what to do if the consumer doesn't respond as expected.

Even if each of the consumer service has a really well-built API, the order service is still responsible for talking to each of them.



For each new consumer, you need to update the producer service to handle its responsibilities for the new API. The producer service needs to know the endpoint for the consumer API, it needs to understand the retry behaviors it's supposed to use, and it has to bundle in a software development kit (SDK) for each of them. This can slow down the addition of new services that want to consume the data, and it adds complexity to the order service that is not directly related to the business logic of the service.

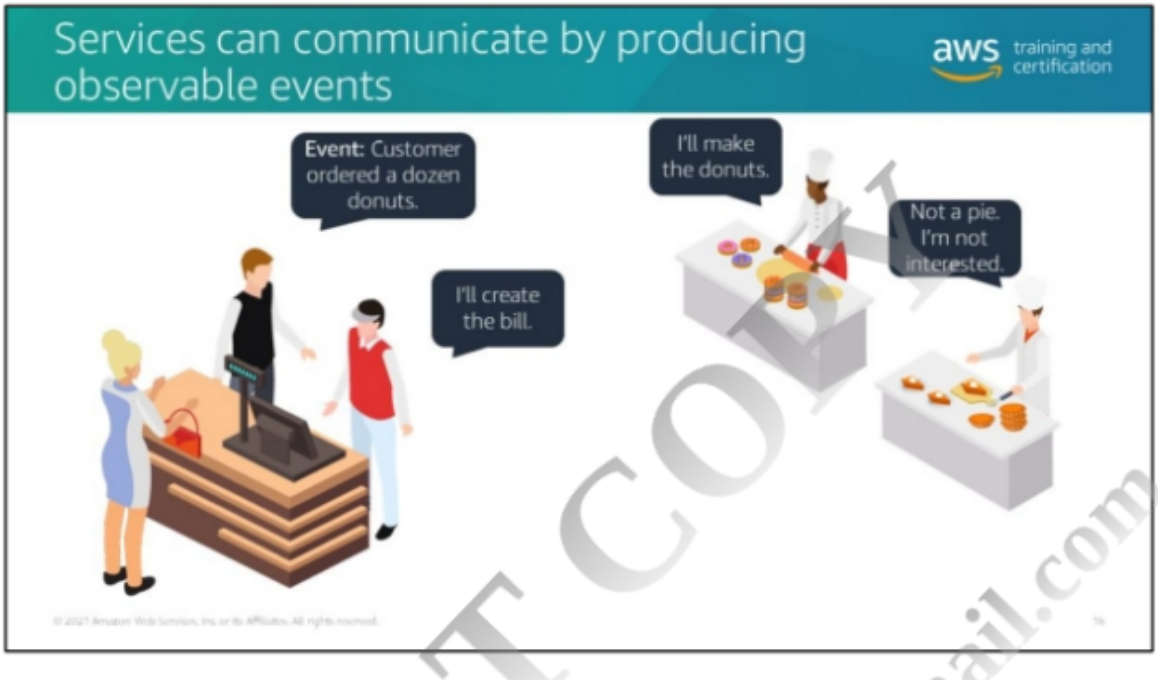
If the APIs need changes, the order service team is responsible for making the update.

How can you reduce the busy work for the order service?

In event-driven design, events are the primary mechanism for sharing information across services.

Events are observable rather than directed.

In event-driven design, event processing usually occurs asynchronous.



In an event-based approach, a producer emits an observable event (for example, order information), and interested consumers take action on the event asynchronously.

The producer puts the event message somewhere that consumers can access it (for example, a known endpoint) but doesn't specify what the consumers will do or which consumers will take action, and there is no direct route back to the producer.

In this example, the introduction of the kitchen manager between the front of the house and the kitchen acts as a messaging service between the front end and the backend.

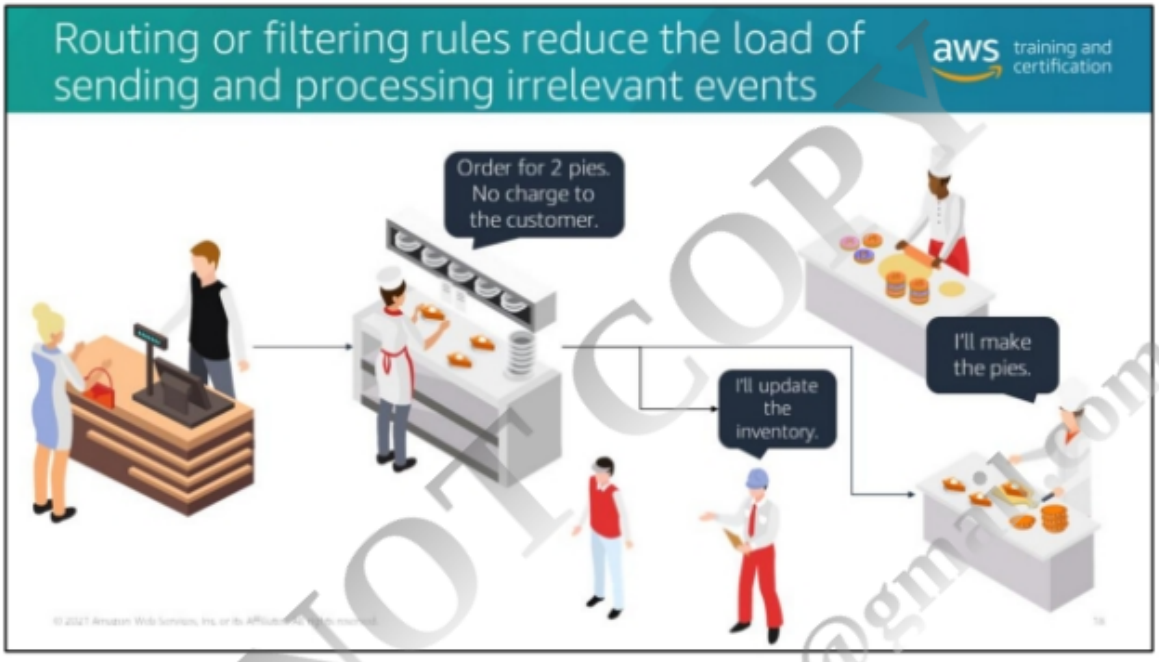
The messaging service handles sharing the events with the appropriate downstream consumers.

This provides further decoupling between the producer and its consumers.

A common example is a pub/sub model in which the producer publishes the message to a known endpoint, and subscribers get messages from that endpoint.

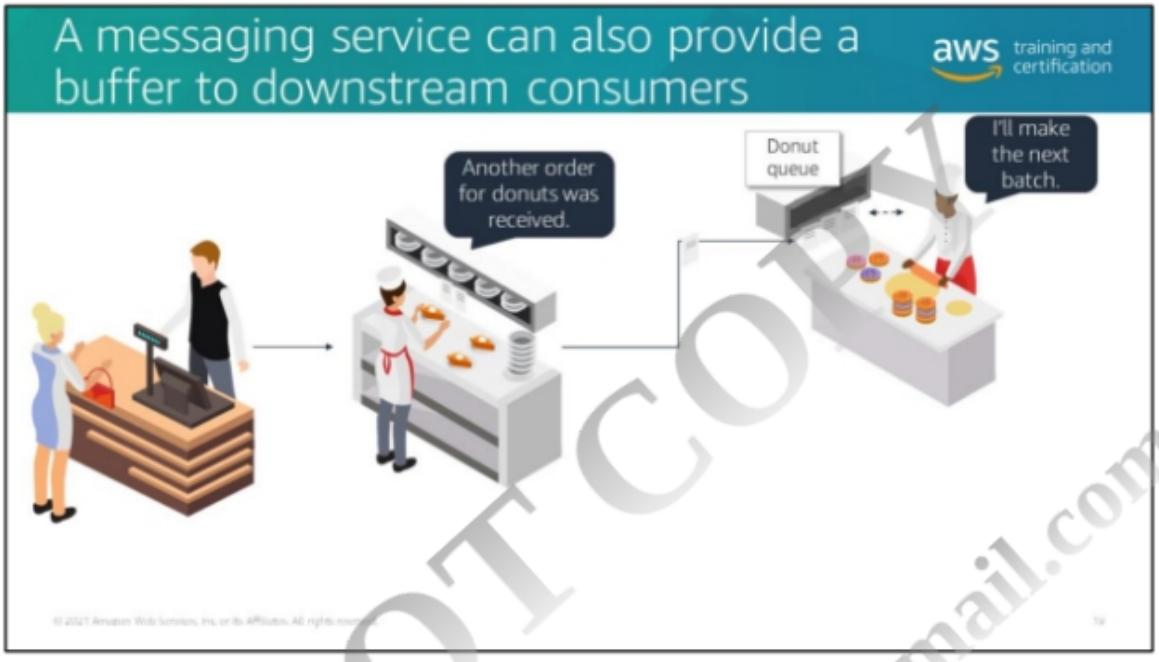
For example, the pie, donut, and invoice services would subscribe to messages about orders.

This decoupling also simplifies the addition of new consumers. In this example, you add the inventory service as a new consumer without impacting the producer.



This example would benefit from using a messaging service that supports routing or filtering rules to reduce unnecessary traffic and to reduce the work of the consumer to filter out unwanted messages.

For example, rather than sending the "free pie" order to all the consumers and letting the donut service and the invoicing service identify that they have no work to do, the messaging service (the kitchen manager) uses its rules to send the pie event to only the pie service and the inventory service. This option reduces the number of messages that the messaging service needs to send and reduces the busy work for the consumers.



An example of this is a queue or stream that stores events that the consumer can pick up at their own pace.

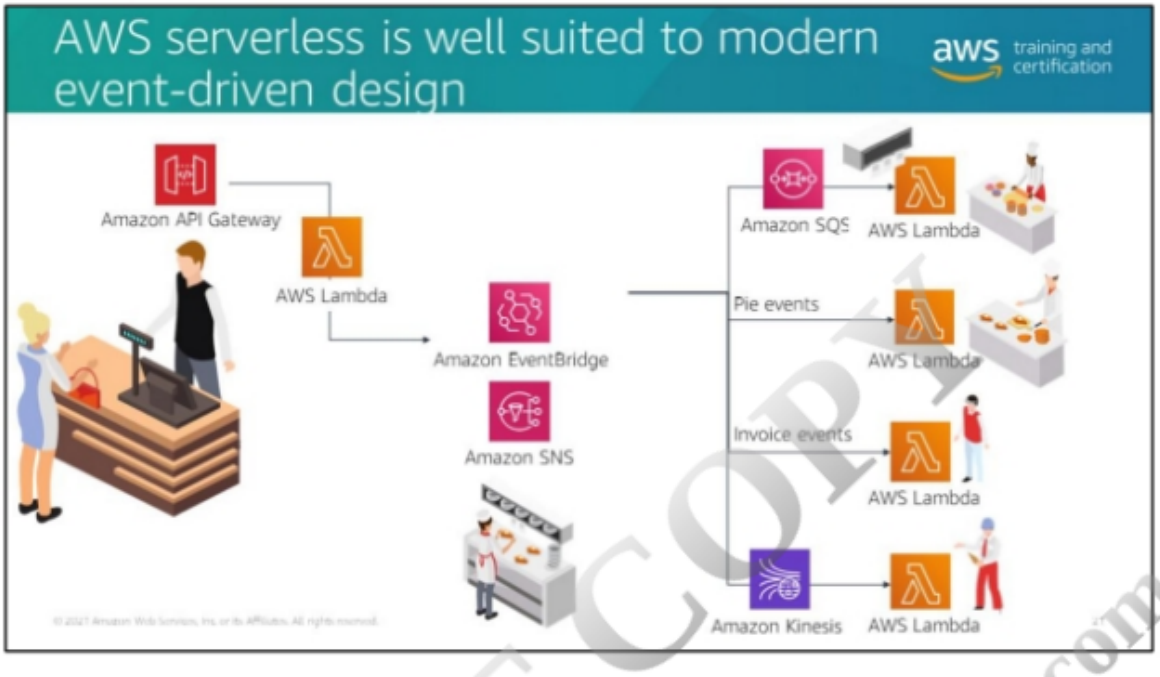
For example, maybe the donut service has its own order queue that collects the incoming order events and then hands them off to the donut makers in controlled batches.

Let's go back to the AWS services and see how they fit into this event-driven approach.

The serverless landing page is a good place to start in regard to resources.

Over the next few minutes, the course introduces serverless services and illustrates in the diner analogy where they fit in to an event-driven application. You might choose to follow along via the slides or to review these services on the https://aws.amazon.com/serverless/ page.

This link is also available in the OCS.



Lambda provides serverless compute and is where your business logic lives.

- Lambda functions run based on an event source and receive data through the event object.
- Lambda functions produce events that other services can consume.
- Lambda has native support for events that AWS messaging and streaming services produce. These integrations can handle communication between services so you can connect things without knowledge of the services' API.

You look more at Lambda in module 7.

You can use Amazon EventBridge and Amazon Simple Notification Service (Amazon SNS) for messaging and fan-out:

- EventBridge is an event router that can abstract producers from consumers and filter events using rules.
- Amazon SNS is a pub/sub service that can filter notifications to subscribers.
- Both services are asynchronous event sources for Lambda and can be Lambda targets. Both provide built-in error handling.

You look at these services in module 5.

You can use Amazon Simple Queue Service (Amazon SQS) and Amazon Kinesis Data Streams for buffering data:

- Amazon SQS can queue requests to prevent a service from becoming overwhelmed. It provides a durable store for events.
- Amazon Kinesis can act as a streaming event source for Lambda. A common use case is data processing where the stream contains data about upstream events. For example, send events that impact inventory to Kinesis Data Streams, and analyze inventory to dynamically offer discounts in the afternoons.

You look at these services in module 6.

Amazon API Gateway is a synchronous event source and is well suited to be the front door to your application. You might need synchronous connections if:

- A client requires a synchronous response.
- You need to direct how the response comes back.
- You need transactional boundaries.
- You need the absolutely lowest latency.
- The message flows within a single service's bounded context.

You look at API Gateway and its role in API access in the next two modules.