



AWS Academy Cloud Architecting  
Module 13 Student Guide  
Version 3.0.0

200-ACACAD-30-EN-SG

© 2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

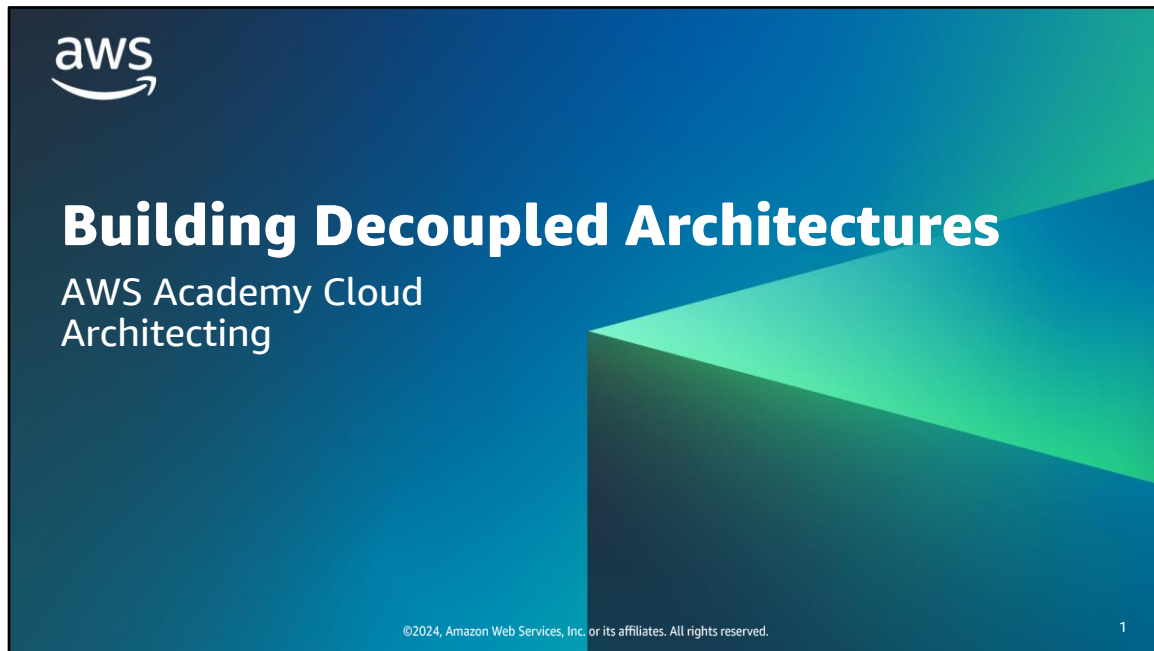
This work may not be reproduced or redistributed, in whole or in part,  
without prior written permission from Amazon Web Services, Inc.  
Commercial copying, lending, or selling is prohibited.

All trademarks are the property of their owners.

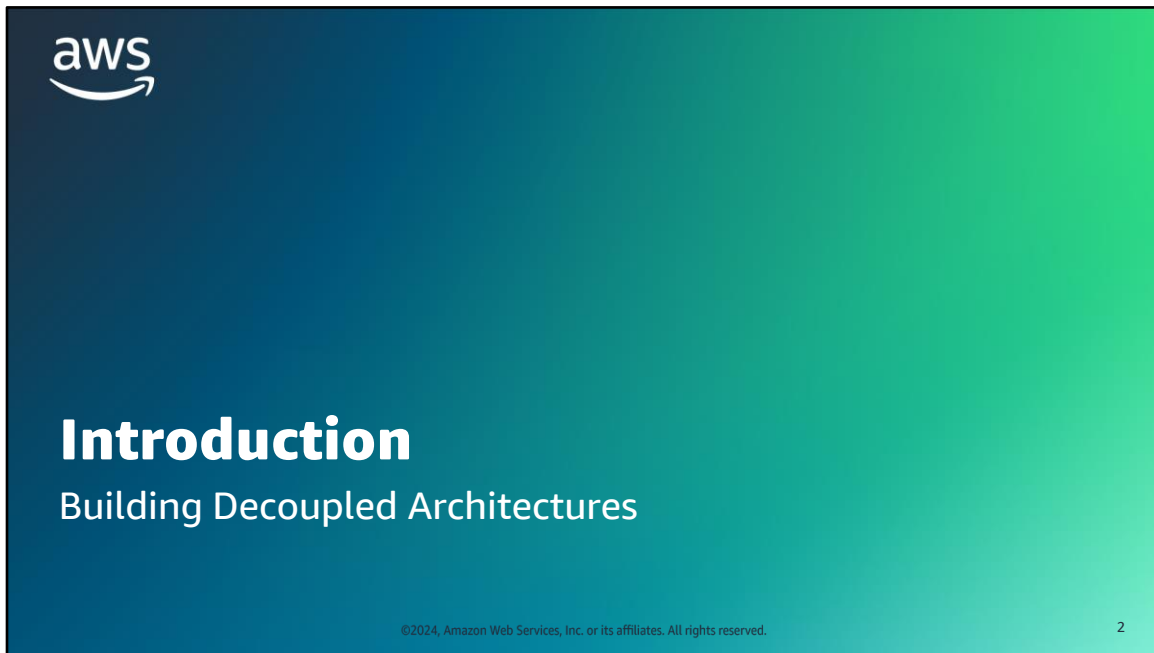
# Contents

Module 13: Building Decoupled Architectures
---

4
---



Welcome to the Building Decoupled Architectures module. This module explains the benefits of decoupling an architecture and describes the techniques and AWS services that you can use to decouple an architecture.



This introduction section describes the content of this module.

## Module objectives



This module prepares you to do the following:

- Differentiate between tightly and loosely coupled architectures.
- Identify how Amazon Simple Queue Service (Amazon SQS) works and when to use it.
- Identify how Amazon Simple Notification Service (Amazon SNS) works and when to use it.
- Describe Amazon MQ.
- Decouple workloads by using Amazon SQS.

©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

3

## Module overview

### Presentation sections

- Decoupling your architecture
- Decoupling applications with Amazon SQS
- Decoupling applications with Amazon SNS
- Decoupling a hybrid application with Amazon MQ

### Knowledge checks

- 10-question knowledge check
- Sample exam question



©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

4

The objectives of this module are presented across multiple sections.

The module wraps up with a 10-question knowledge check delivered in the online course, and a sample exam question to discuss in class.

The next slide describes the lab in this module.

## Hands-on lab in this module

### Guided lab

- Building Decoupled Applications by Using Amazon SQS





©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

5

This module includes the guided lab listed. In this lab, you will create and use an Amazon Simple Queue Service (Amazon SQS) queue to decouple functions within an application. Additional information about this lab and each lab is included in the student guide where the lab takes place, and the lab environment provides detailed instructions.



**As a cloud architect designing application architectures:**



- I need to address potential performance bottlenecks so that the architecture scales as traffic increases.
- I need to limit the impact of failures across the application so that one component's failure does not bring down the application.
- I need to implement architectures in which changes can occur to one application component without affecting the other parts so that I can minimize downtime during maintenance.

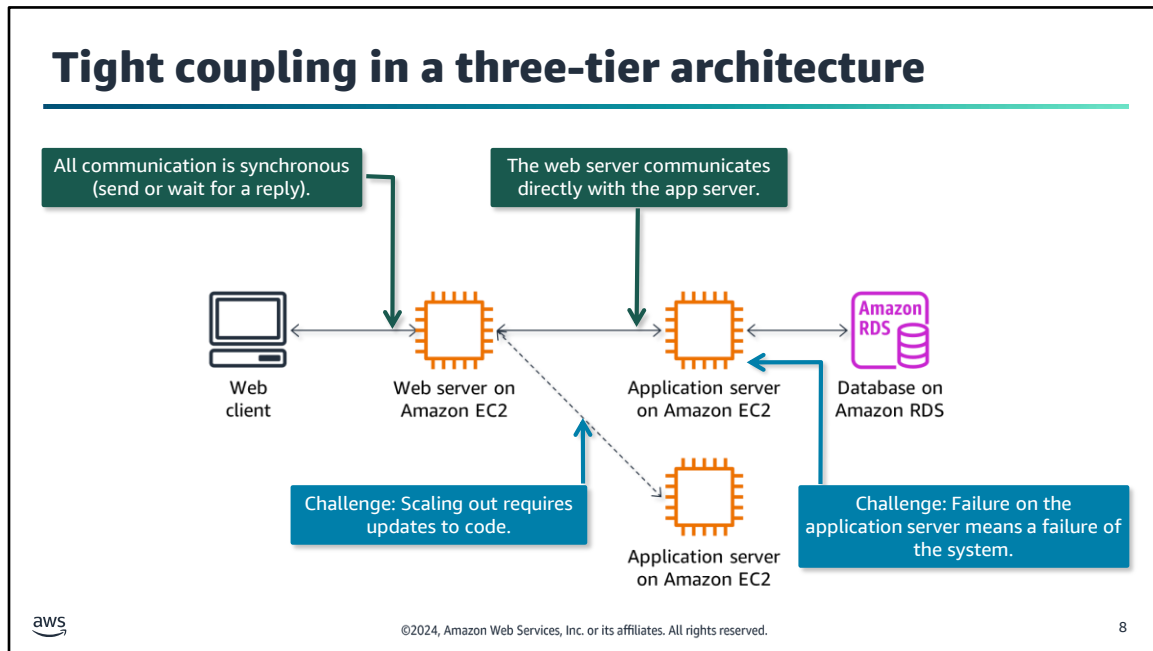
©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

6

This slide asks you to take the perspective of a cloud architect as you think about how to approach cloud network design. Keep these considerations in mind as you progress through this module, remembering that the cloud architect should work backward from the business need to design the best architecture for a specific use case. As you progress through the module, consider the café scenario presented in the course as an example business need, and think about how you would address these needs for the fictional café business.



Decoupling refers to the separation of components in a system so that they can operate independently. This section discusses the motivations for decoupling an architecture and describes various techniques that you can use to decouple an architecture.

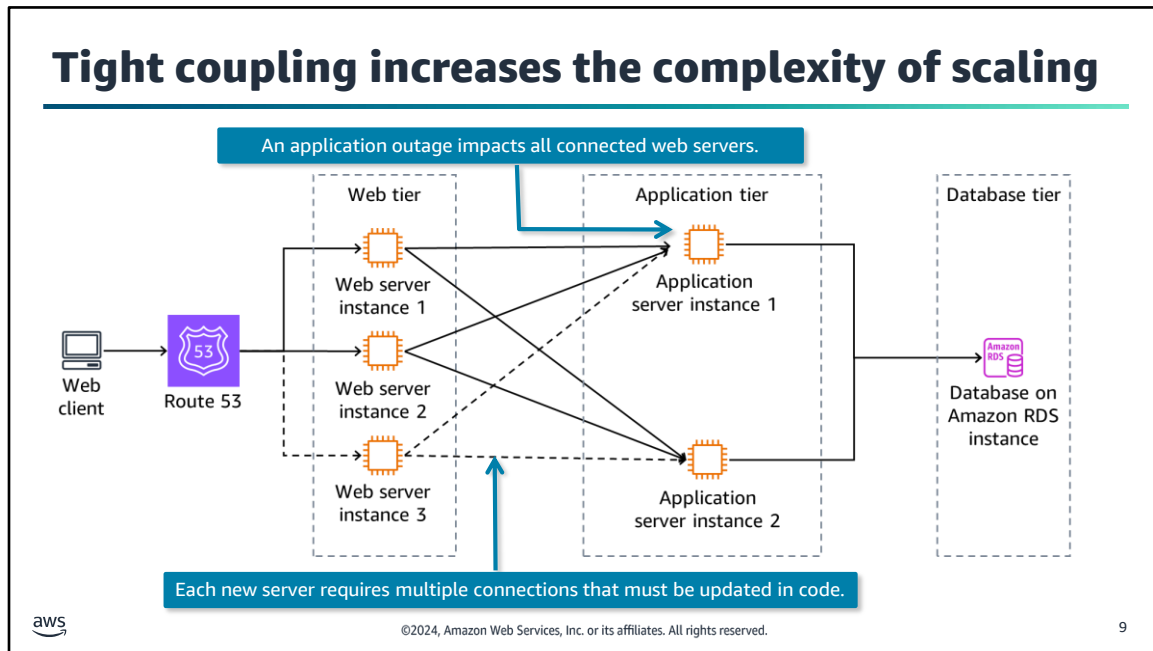


Tight coupling refers to a system in which linked components are dependent on each other to the extent that changes or failures in one component force changes or failures in other components. In a three-tier architecture, the presentation tier (on a web server), business logic (on an application server), and the backend database are developed and maintained as independent components. This architecture provides greater flexibility than earlier architectures when you need to modify one part of the application or scale out the architecture.

However, if those independent components are still tightly coupled with each other, the architecture is still not very resilient. Take an example of a business that has moved its on-premises web application to the cloud and implemented a three-tier architecture. The architecture uses Amazon Relational Database Service (Amazon RDS) for the database server tier and Amazon Elastic Compute Cloud (Amazon EC2) for the web server and application server tiers. Communication between tiers is synchronous, meaning a source component sends a request to the target component and waits for a response before proceeding.

Consider the following challenges:

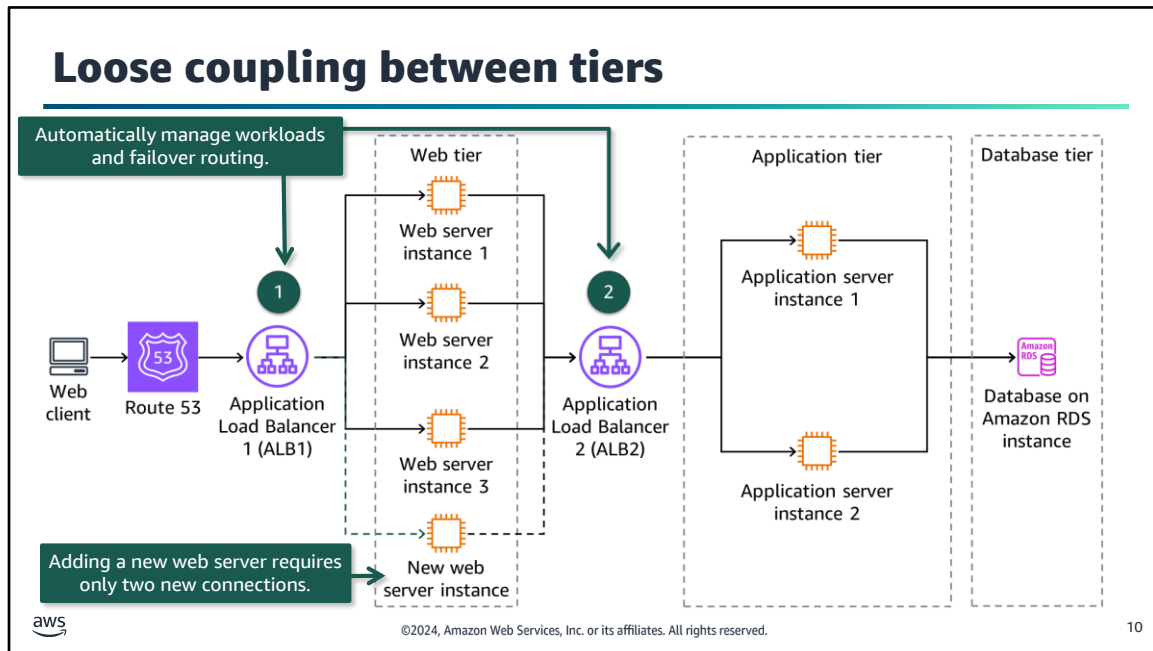
- If the application server fails or the web server goes down, the whole system is down, and errors are returned to the client.
- If a systems administrator needs to update the application server for maintenance or to fix a bug, they have to bring the whole system offline.
- If the systems administrator wants to scale out one of the tiers, they have to update the code to be aware of the IP addresses of the new components and to know when to route traffic to each one.



The challenges of scaling in a tightly coupled system create additional complexity, which can impede the application's ability to scale. In this example, the business has scaled up its application system by adding two additional web servers and one additional application server to provide greater resiliency. They have also introduced Amazon Route 53 as their name service, which lets them know when one of the web servers is unavailable.

However, because related components are directly connected to each other, adding or removing a component to scale out or scale in requires making or removing connections in every related layer. The more components you have tightly coupled, the more connections you have to add or remove. In the tightly coupled architecture example, adding a new web server instance to the web tier requires establishing a total of three new connections: one from Route 53 to the new instance and two from the new instance to the application servers. Likewise, adding a new application server instance to the application tier would require establishing three new connections: one from each web server instance to the new application server instance.

Also, although the number of servers have increased, the systems administrator would have to handle application outages manually. If one of the application server instances goes down, the performance of all of the web servers would be affected because they are directly connected to the instance.



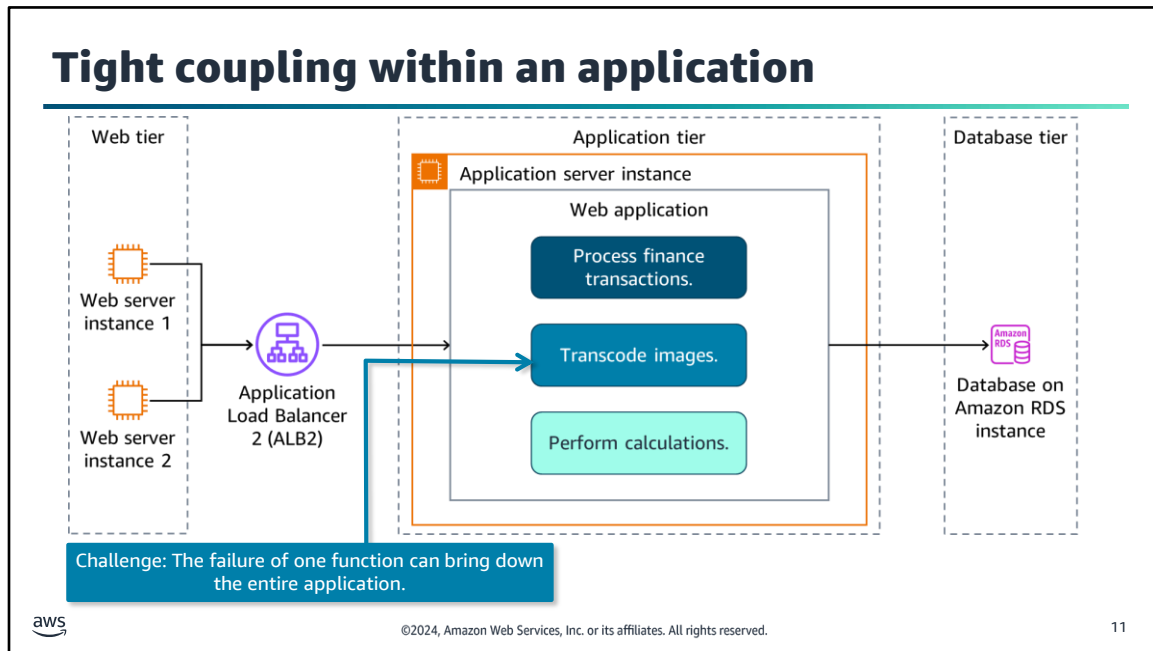
At the infrastructure level, an effective way to alleviate the issues caused by a tightly coupled architecture and implement loose coupling is to introduce an intermediary component between dependent layers. With loose coupling, you reduce dependencies in your system by using managed solutions, such as Elastic Load Balancing (ELB), as intermediaries between the layers of your system. This way, the intermediary automatically handles failures and the scaling of components or layers.

In the three-tier web application architecture example, loose coupling is implemented at the infrastructure level as follows:

1. Add an ALB load balancer (a type of ELB) in front of the web servers. This load balancer distributes traffic from Route 53 to the web server instances. It monitors the health of all of the instances and sends traffic to only the instances that are healthy.
2. Add an ALB load balancer between the web server instances and the application server instances. Similar to the previous item, this load balancer performs automatic workload management and failover routing functions between the web server and application server instances.

If you now want to scale out and add a new web server instance, you need to make only two connections: one to ALB1 and the other to ALB2.

A detailed description of the ELB service is beyond the scope of this module. For more information on ELB, see the *Elastic Load Balancing User Guide*. A link is provided in your course resources.



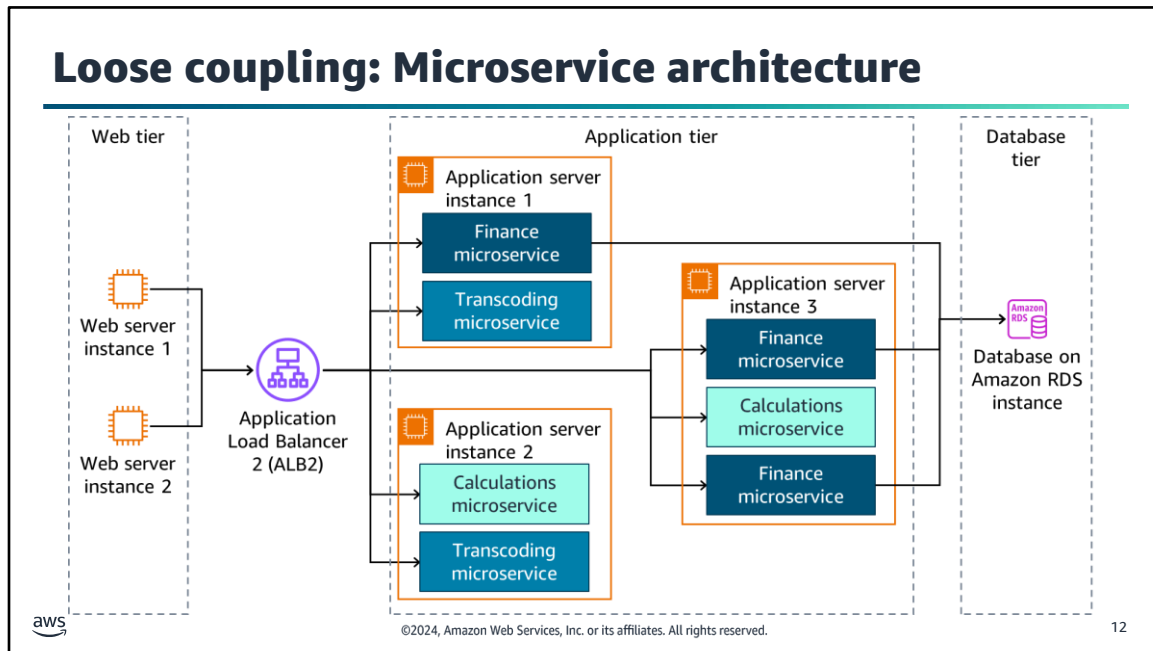
Tight coupling can also occur within an application when the business functions of the application are all implemented and deployed in a single unit, such as in a monolithic application. This design presents the following challenges:

- A problem in a single function can cause all the other application functions to slow down or the entire application to stop.
- Changes to one function require the entire application to be put into maintenance.

In the three-tier architecture example, consider the scenario in which the web application has the three distinct functions to process finance transactions, transcode images, and perform calculations. The application functions are tightly coupled into a single deployment unit and run in the same process in an application server instance.

If a slowdown in one application function occurs, for example in the transcode images function, it can degrade the performance of the entire application or even cause the whole application to be unresponsive. Responses to incoming requests from the web server tier would gradually slow down and eventually stop if the application were unresponsive.

To address these challenges, you can design applications that use loose coupling techniques based on microservice architecture and asynchronous message solutions.



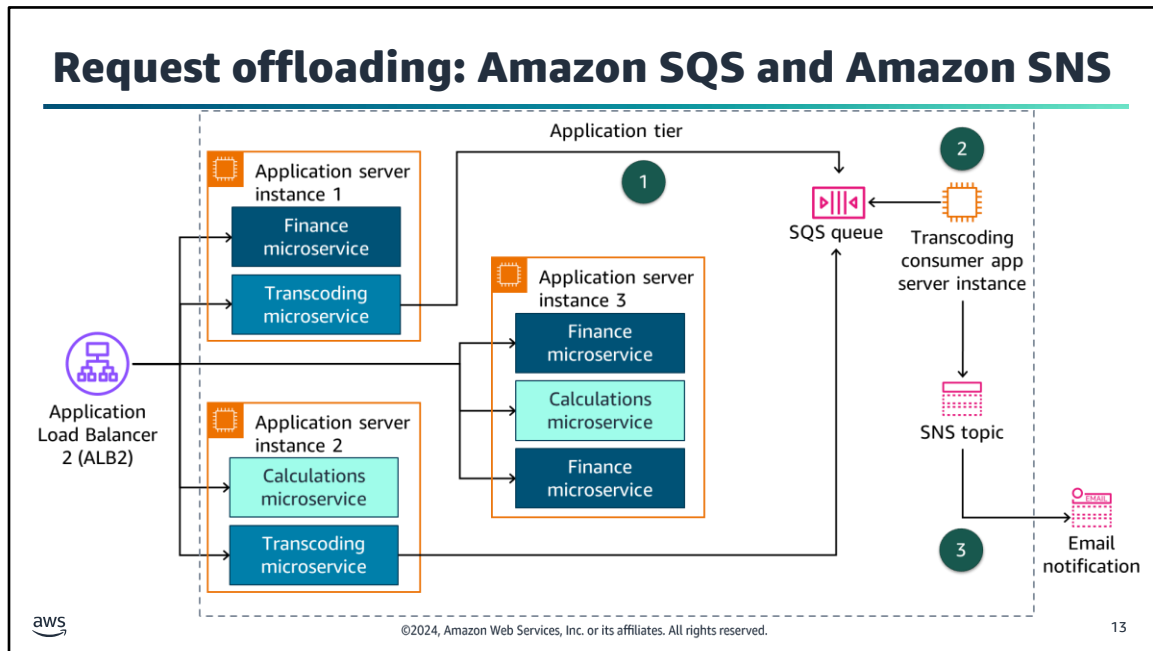
12

A microservice architecture divides application functions into parts that can scale and fail independently. Each microservice runs in its own process, maintains its own data, and exposes its function through a well-defined API. This results in an application architecture with reusable components that are scalable and reliable. The microservices that implement an application are loosely coupled and can communicate with each other by using synchronous or asynchronous communication. This loose coupling gives you the ability to make changes to an application without affecting other parts of the application. You can also add features to a component while minimizing risk to components that depend on it.

In the example on this slide, the tightly coupled functions of the web application have been split into separate microservices: finance, transcoding, and calculations. Each microservice runs in its own process (in this case, its own container) and can scale independently of the other microservices. Notice, for example, that the finance microservice is scaled to run in three containers while the calculations microservice runs in only one. This is because the finance microservice receives and needs to handle more requests than the calculations microservice. In addition, if one of the microservices fails (for example, the transcoding microservice in application server instance 1), the failure does not affect the finance and calculations microservices. It also does not affect the transcoding microservice running in application server instance 2.

A detailed description of microservices is beyond the scope of this module. The module titled Building Microservices and Serverless Architectures discusses microservices.

For more information on microservices, see “Implementing Microservices on AWS” in *Implementing Microservices on AWS*. A link is provided in your course resources.



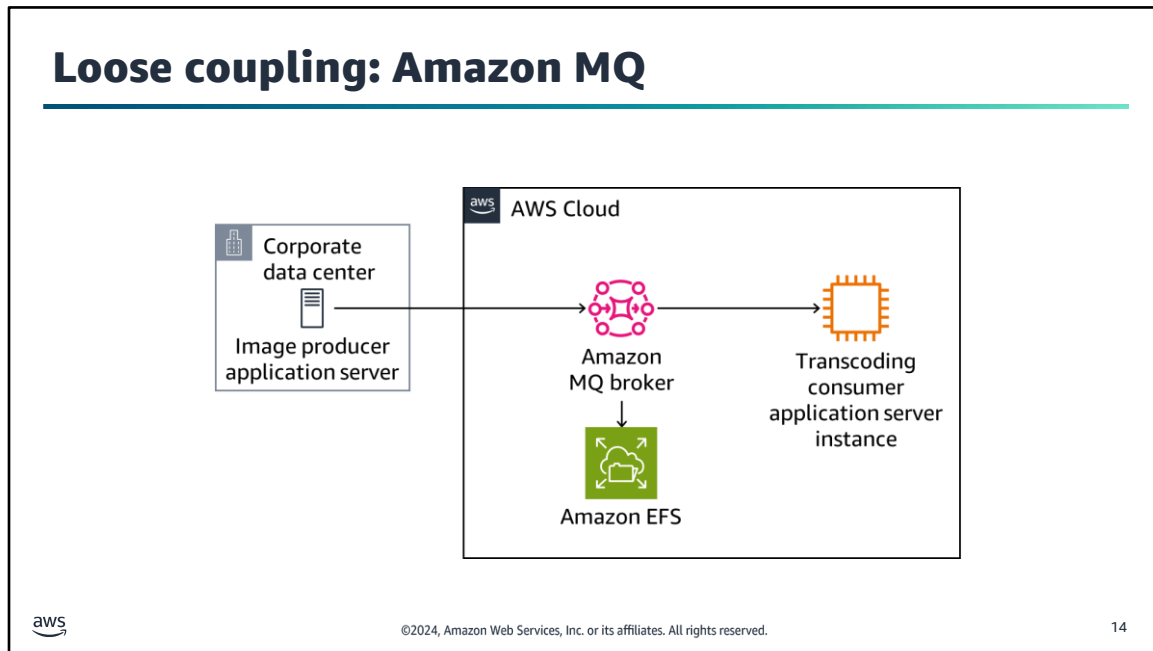
To further improve resiliency through loose coupling, make component interactions asynchronous where possible. This model is suitable for any interaction that does not need an immediate response and where an acknowledgment that a request has been registered will suffice. It involves one component that generates events, the *producer*, and another that consumes them, the *consumer*. The two components do not integrate and interact directly but usually do so through an intermediate durable storage layer, such as a queue or a topic. This decoupled architecture facilitates component integration and reuse. Note that application components that are decoupled through asynchronous messaging can be traditional functions or microservices.

In the example web application, the business notices that the response time for the transcoding microservice is poor. Users are noticing that they have to wait a long time for an image to be transcoded and for one image to be done before they can send the next one. This response time is due in part to the synchronous design of the function, which can process only one image at a time and does not return a response to the user until an image has finished processing. To address this issue, the business redesigns the function to use asynchronous loose coupling solutions.

The diagram illustrates the resulting architecture, which consists of the following changes:

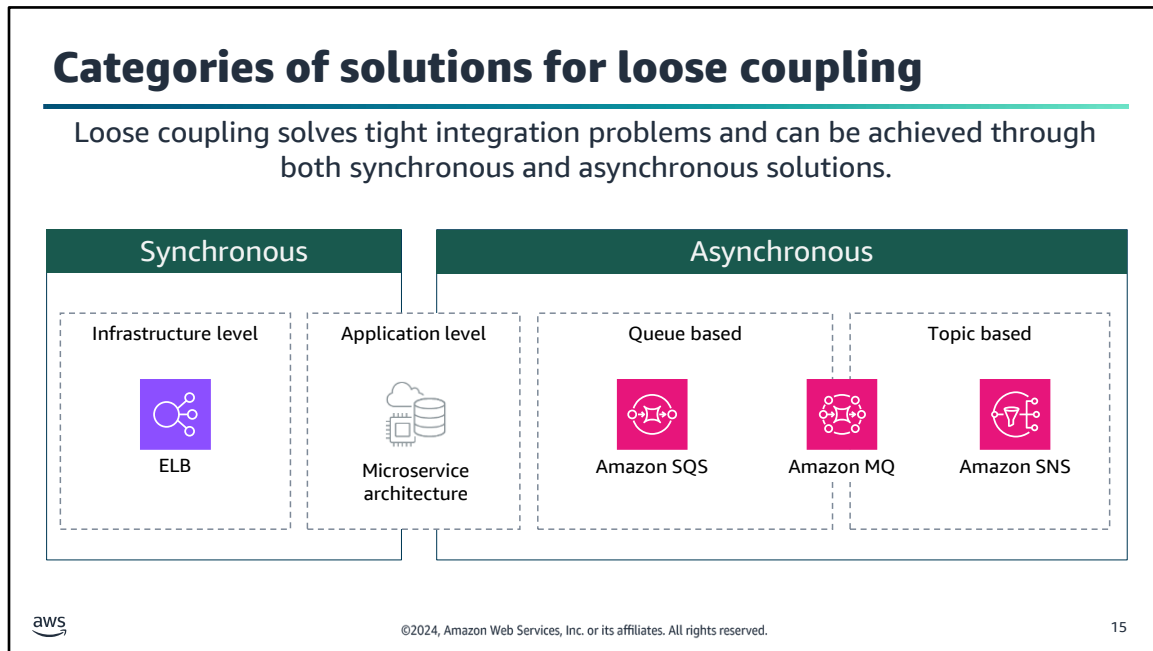
1. The logic that performs the transcoding function moves from the transcoding microservice to a separate consumer application. The microservice becomes a message producer and connects asynchronously to the consumer application by using an SQS queue. For every request that the microservice receives, it stores the uploaded image to a storage system (for example, Amazon Simple Storage Service [Amazon S3]) and puts a message in the queue for the consumer application to process. The microservice immediately responds to the user to acknowledge the receipt of the request and accept the submission of a new request.
2. The transcoding consumer application periodically polls the SQS queue for a message. For each message that it retrieves, the application performs the transcoding function on the image associated with the message. Note that servers for this application can scale out and scale in as needed to achieve the desired performance targets.
3. After the consumer application finishes transcoding an image, it puts a message in an Amazon Simple Notification Service (Amazon SNS) topic to indicate that it has completed this task. The topic sends an email to the user notifying them that the image has been successfully transcoded.





What if an on-premises application wants to use the transcoding consumer application in the AWS Cloud to transcode images? How could this integration be achieved so that the on-premises application does not need to connect directly to the consumer application and become tightly coupled with it? The answer is to use Amazon MQ. This service can loosely couple applications running on premises with applications running in the AWS Cloud through asynchronous messaging.

As the diagram on this slide illustrates, an on-premises application that produces images can send a message to the transcoding consumer application through an Amazon MQ broker. The broker stores messages by using the Amazon Elastic File System (Amazon EFS) or Amazon Elastic Block Store (Amazon EBS) service. By using Amazon MQ to provide a message-based interface for the consumer application, the business can give clients outside of the AWS Cloud the ability to reuse the function in a flexible and consistent way.



To help ensure that your application scales as the load increases and that there are no bottlenecks or single points of failure in your system, implement loose coupling. Loose coupling helps isolate the behavior of a component from other components that depend on it, which increases resiliency and agility. Implementing loose coupling between dependencies isolates a failure in one component from impacting another component.

Loose coupling solutions can be synchronous or asynchronous. In the synchronous category, infrastructure-level solutions usually involve introducing an intermediary component between dependent layers. For example, using an ELB load balancer between a tightly coupled web server and application server layer loosely couples the layers. At the application level, the use of a microservice architecture is an effective way to loosely couple application functions.

Asynchronous loose coupling solutions involve the use of messages, and queues or topics between dependent components. Amazon SQS is an AWS service that support asynchronous messaging by using queues, and Amazon SNS uses topics. Amazon MQ supports both queues and topics.

Note that this module focuses on the asynchronous messaging solutions and identifies and introduces the other solutions only for completeness. The next sections discuss Amazon SQS, Amazon SNS, and Amazon MQ in more detail.

For more information on ELB, see “What is Elastic Load Balancing?” in the *Elastic Load Balancing User Guide*. A link is provided in your course resources.

## Key takeaways: Decoupling your architecture



- Tightly coupled systems are difficult to scale and introduce bottlenecks and single points of failure.
- A loosely coupled architecture removes direct dependencies between related components and permits scalability and resiliency.
- Loose coupling solutions divide infrastructure layers or application functions and typically introduce an intermediary component between them.
- Loose coupling solutions can be synchronous or asynchronous.
  - ELB is an example of a synchronous solution.
  - Amazon SQS, Amazon SNS, and Amazon MQ are examples of an asynchronous solution.

©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

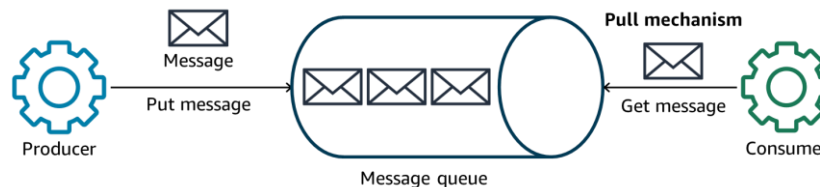
16



This section describes how Amazon SQS works and when to use it.

## Point-to-point messaging

- You can decouple applications asynchronously by using point-to-point messaging.
- Using point-to-point messaging when the sending application sends a message to only one specific receiving application.
- The sending application is called a *producer*.
- The receiving application is called a *consumer*.
- Point-to-point messaging uses a *message queue* to decouple applications.



©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

18

Point-to-point messaging is an asynchronous messaging model that helps you implement a decoupled architecture. It permits two applications to communicate asynchronously by using messages and a message queue. It is called point-to-point because the sending application knows who the receiving application is and sends messages to only one consumer. The sending application is called a *producer* because it generates messages and puts them in the queue. The receiving application is called a *consumer* because it gets messages from the queue and processes them.

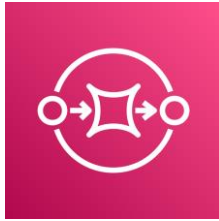
A *message* represents the data sent by the producer to the consumer. A *message queue* is a temporary repository for messages that are waiting to be processed. Messages are usually small and can be things such as requests, replies, error messages, or information. Examples of messages include customer records, product orders, invoices, and patient records.

A consumer retrieves a message from a queue by polling the queue periodically to check if a message is available for retrieval. If so, the consumer retrieves the message and processes it. This approach is called a *pull mechanism*.

The diagram on the slide illustrates the message flow in point-to-point messaging, which is as follows:

1. The producer puts a message in the queue.
2. The queue stores the message until it is retrieved.
3. The consumer gets the message from the queue.

## Amazon Simple Queue Service (Amazon SQS)



Amazon SQS

- Is a fully managed message queueing service
- Helps integrate and decouple distributed software systems and application components
- Provides highly available, secure, and durable message-queueing capabilities
- Provides an AWS Management Console interface and a web services API



©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

19


Amazon SQS is a fully managed, message queueing service that you can use to decouple application components so they run and fail independently. It acts as a buffer between senders and receivers. For example, it lets web service applications queue messages that are generated by one application component to be consumed by another component.

Amazon SQS works on a massive scale and can process billions of messages per day. It stores all message queues and messages within a single, highly available AWS Region with multiple redundant Availability Zones. This redundancy protects the access to messages from any single computer, network, or Availability Zone failure.

You can access Amazon SQS by using the AWS Management Console to create queues and test message communication. You can also access Amazon SQS programmatically by using an API through the AWS SDKs.

## Amazon SQS benefits

Fully managed	Reliability	Security	Scalability
Avoid the need to manage messaging software or maintain infrastructure.	Deliver large volumes of data without losing messages.	Send sensitive data securely between applications.	Scale elastically based on usage.

 ©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved. 20

With Amazon SQS, you do not have to manage a dedicated messaging infrastructure. As a fully managed service, Amazon SQS automatically provisions and manages the compute, storage, and networking resources needed to support messaging. Amazon SQS requires no administrative overhead and little configuration.

Amazon SQS provides reliable communication between distributed software components. You can use it to exchange messages between any number of systems without losing any messages. Amazon SQS provides extremely high message durability because it stores messages on multiple servers. You can reliably deliver large volumes of data at any level of throughput.

In addition, you control who can send messages to and receive messages from a queue. You can also protect sensitive data by encrypting messages in queues with server-side encryption (SSE) and by using keys that are managed by the AWS Key Management Service (AWS KMS). Amazon SQS decrypts messages only when they are sent to an authorized consumer.

You can use Amazon SQS to build applications that are fault-tolerant and scalable. It works on a massive scale and can process billions of messages each day. Messages can be sent and read simultaneously, and multiple producers and consumers can interact with the same queue. You can scale elastically and cost-effectively based on usage so that you don't have to worry about capacity planning or preprovisioning. You can also scale the number of messages you send to Amazon SQS up or down without any configuration.

## Amazon SQS basic components



Message



Queue



Dead-letter queue

- A message can be up to 256 KB in size.
- A message remains in a queue until it is explicitly deleted or exceeds the queue's message retention period.
- Amazon SQS offers two types of queues: standard and first-in-first-out (FIFO)
- You can configure queue parameters, including the following:
  - Message retention period
  - Visibility timeout
  - Receive message wait time (short polling versus long polling)
- You can associate a dead-letter queue (DLQ) with any queue.
- A DLQ stores messages that cannot be consumed successfully.



©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

21

The basic components of Amazon SQS are messages and queues. This slide lists some of their key features.

Amazon SQS messages have a maximum size of 256 KB and can contain XML, JSON, and unformatted text. To send messages larger than 256 KB, you can use the Amazon SQS Extended Client Library for Java. You can use this library to send and receive messages with payloads up to 2 GB. In addition, messages in a queue remain in the queue until the consumer deletes them or they exceed the queue's message retention period. By default, a queue retains messages for 4 days. You can configure a queue to retain messages for up to 14 days.

Amazon SQS supports two types of queues: standard queues and first-in-first-out (FIFO) queues. One major difference between the two types is that a standard queue occasionally delivers messages in an order different from which they were sent. A FIFO queue, however, strictly preserves the order in which messages are sent and received.

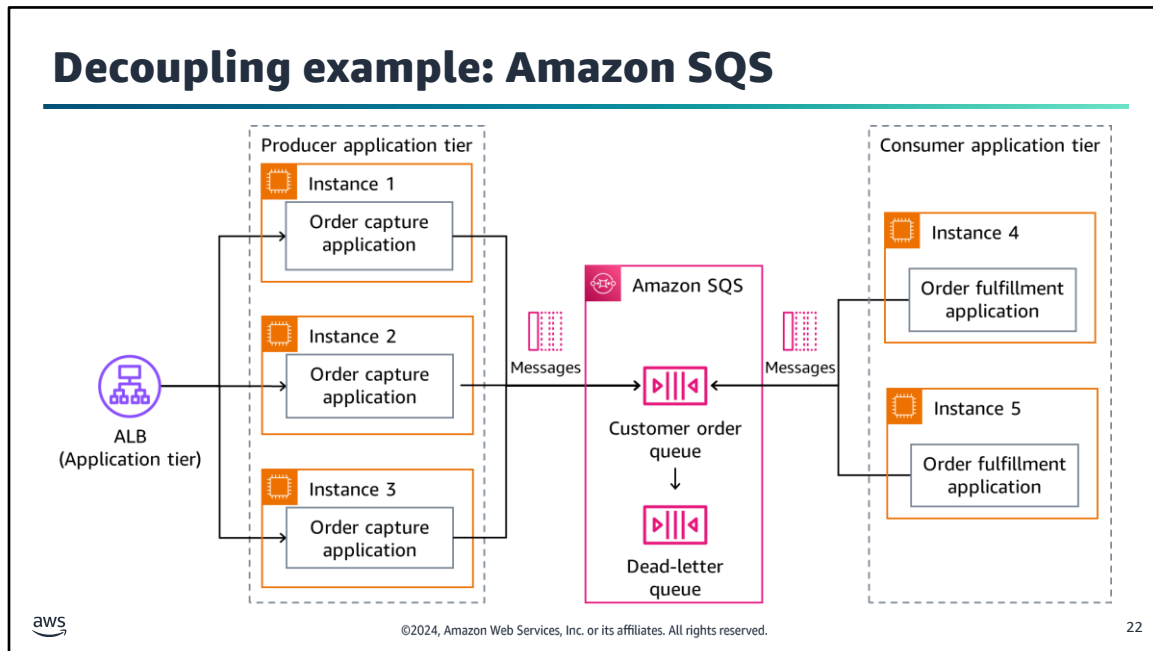
Queues have a number of parameters that you can configure. They include the following:

- **Message retention period:** This parameter is the amount of time that Amazon SQS retains messages that remain in the queue.
- **Visibility timeout:** This parameter is the length of time that a message received from a queue by one consumer is not visible to the other message consumers. This queue parameter can be overridden at the message level when a consumer gets a message from the queue.
- **Receive message wait time:** This parameter is the maximum amount of time that Amazon SQS waits for messages to become available after the queue gets a receive request. This parameter controls the polling behavior of the queue (short polling or long polling).



Amazon SQS also features dead-letter queue (DLQ) support. A DLQ is a queue that is associated with another queue—the source queue—and receives the messages from the source queue that could not be processed. After exceeding the maximum number of processing attempts, Amazon SQS automatically moves the message from the source queue to the associated DLQ. A DLQ is like any other SQS queue: messages can be sent to it and received from it. The DLQ of a standard queue must also be a standard queue. Similarly, the DLQ of a FIFO queue must also be a FIFO queue.

You learn more about queue types and queue parameters in the next slides.

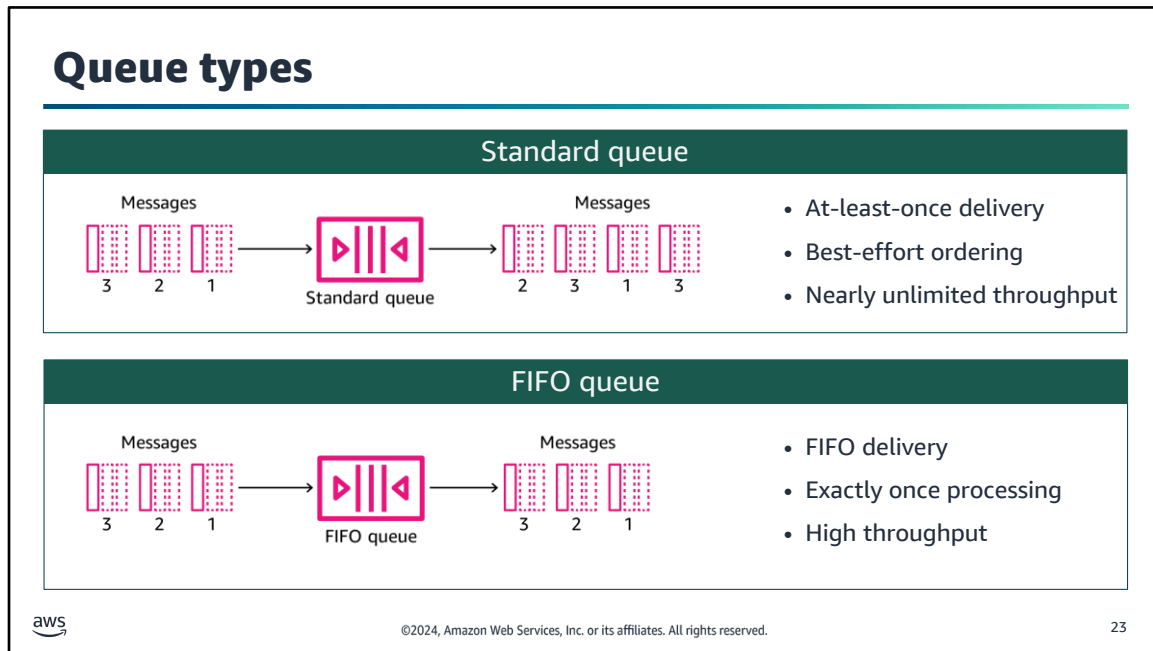


Consider the example of a web application that processes customer orders and has an order capture and order fulfillment function. In a tightly coupled design, as in a monolithic application, both functions would be implemented and deployed in a single unit. You can decouple the architecture of this application by introducing an SQS queue. You can use the queue to isolate the order fulfillment logic into its own component so that it runs in a separate process from the order capture web application.

In this example, the order capture application is the producer. It creates customer orders and sends them as messages to an SQS queue, the customer order queue. The producer application is scaled to run in three instances that are load balanced by an ALB type load balancer. All three producer instances send messages to the customer order queue. The order fulfillment application is the consumer and processes the orders from the producer instances. The consumer application runs on two instances, which can both retrieve messages from the queue. It polls the queue and receives the messages in the queue. After successfully processing a message, the consumer application deletes the message from the queue. If a message cannot be processed, Amazon SQS sends it to the DLQ.

Using SQS queues in this example provides the following benefits:

- The application is more resilient to spikes in traffic. The queue acts as a buffer to absorb spikes, and the order capture and order fulfillment functions can scale independently of each other.
- Work can be performed only as fast as necessary to manage costs. Because orders are temporarily stored in a queue when they are captured, they can be processed asynchronously at a different pace.
- If an application exception occurs, order processing can be retried or redirected to a dead-letter queue for reprocessing at a later stage.



There are two types of SQS queues: standard queues and FIFO queues.

Standard queues offer the following features:

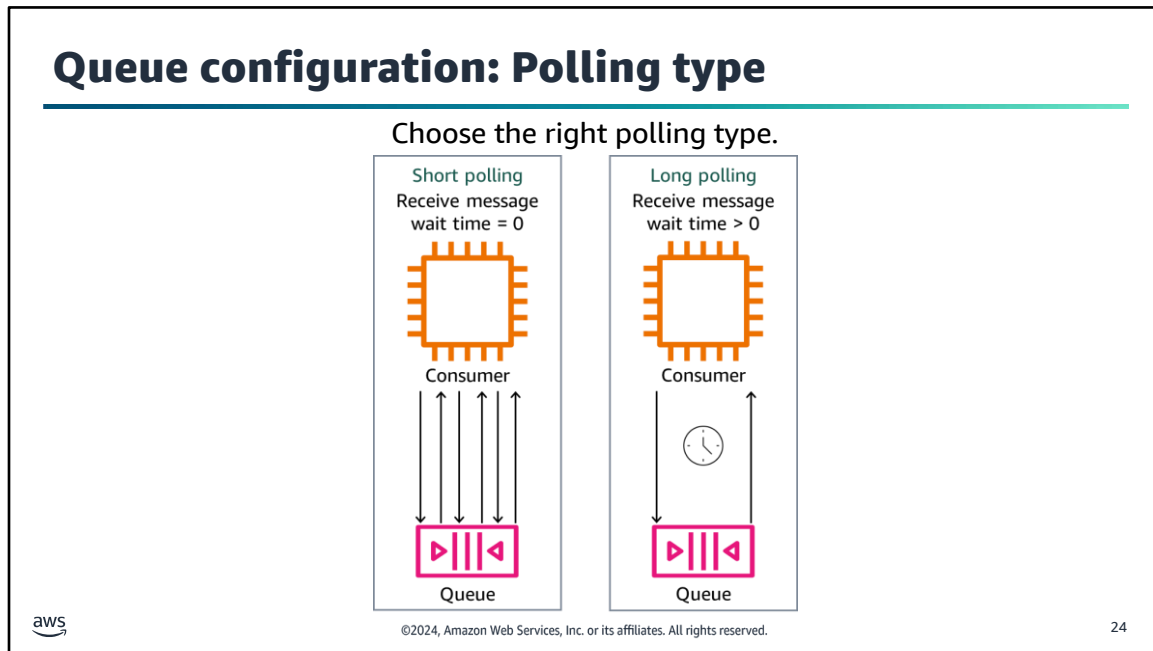
- **At-least-once delivery:** A message is delivered at least once, but occasionally more than one copy of a message is delivered.
- **Best-effort ordering:** Occasionally, messages might be delivered in an order that is different from the order in which they were sent.
- **Nearly unlimited throughput:** Standard queues support a nearly unlimited number of API calls per second for each API operation (SendMessage, ReceiveMessage, or DeleteMessage).

FIFO queues the following features:

- **First-in-first out delivery:** Messages are delivered in the exact order that they are sent.
- **Exactly once processing:** Messages are processed exactly once.
- **high throughput:** FIFO queues support up to 300 API calls per second for each API operation. When you batch 10 messages per operation (maximum), FIFO queues can support up to 3,000 API calls per second.

As a general guideline, use a standard queue when your consumer application can process messages that arrive more than once and out of order. If your consumer application requires that the order be preserved, use a FIFO queue.

In the diagram for the standard queue, three messages enter the queue numbered and in the sequence of 1, 2, and 3. Because the standard queue does not guarantee ordering and exactly one delivery, the messages leave the queue in the sequence of 3, 1, 3, and 2. Note that message 3 is delivered twice. However, in the diagram for the FIFO queue, the messages enter the queue in the order of 1, 2, and 3, and exit the queue in the exact same order. In addition, each message is delivered only once.



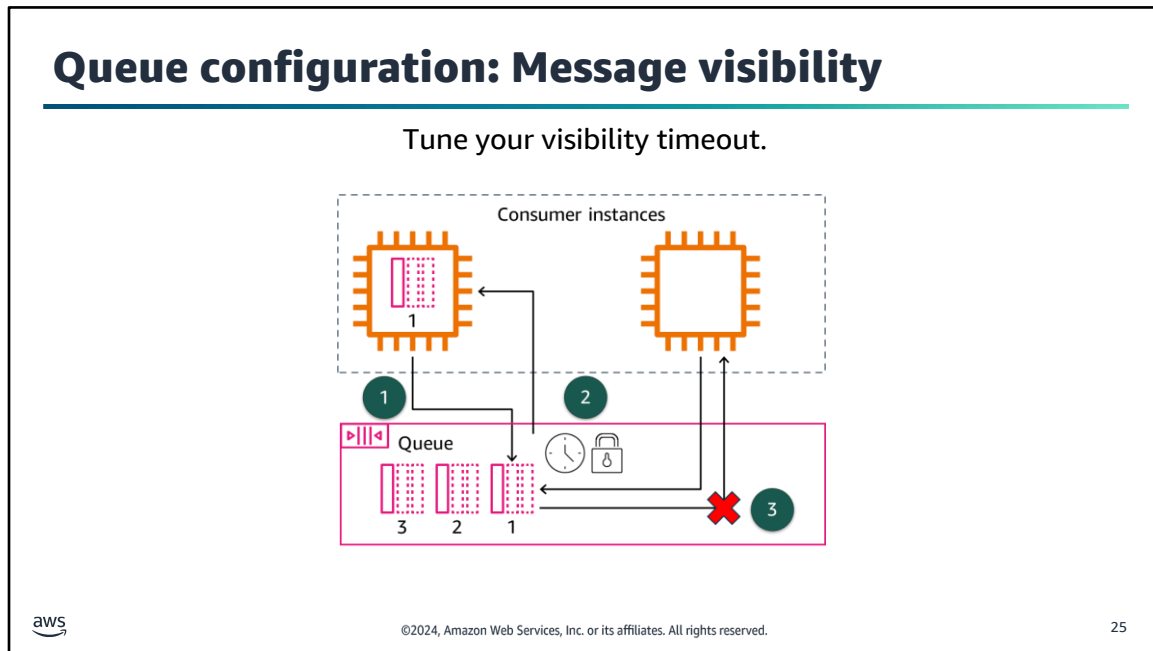
When creating an SQS queue, you need to consider how your application interacts with the queue and configure the queue parameters accordingly. One particular parameter that you should optimize is the *receive message wait time*.

The receive message wait time parameter controls the polling type for a queue. Amazon SQS supports both short polling and long polling to retrieve messages from a queue. By default, queues use short polling, which is indicated by a value of zero for the receive message wait time (polling wait time) parameter. Any nonzero value sets long polling, and the maximum allowable value is 20 seconds.

To provide durability, Amazon SQS redundantly stores the messages in a queue on multiple servers. When a consumer requests a message, short polling queries only a subset of the servers (based on a weighted random distribution) and returns messages from only those servers. Amazon SQS then sends a response to the consumer immediately even if the query found no messages. This results in a faster response but increases the number of responses and therefore increases costs. In contrast, long polling queries all of the servers for messages, and Amazon SQS sends a response to the consumer when the polling wait time expires. The resulting effect is that Amazon SQS waits until a message is available in the queue before sending a response. Long polling provides less frequent responses but decreases costs because it reduces the number of empty receives.

Depending on the frequency of messages arriving in your queue, many of the responses from a queue that uses short polling could be reporting an empty queue. In almost all cases, Amazon SQS long polling is preferable to short polling. One exception is if your application requires an immediate response to its poll requests: for example, as in the case of an application consuming real-time stock prices. Another exception is if your application uses a single thread to poll multiple queues. Using long polling will probably not work because the single thread will wait for the long-poll timeout on any empty queues, delaying the processing of any queues that might contain messages.

The diagram shows that with short polling, the frequency of responses from the queue to the consumer is high. However, with long polling, the frequency is lower because the queue has to wait for the polling wait time to elapse before returning a response to the consumer.

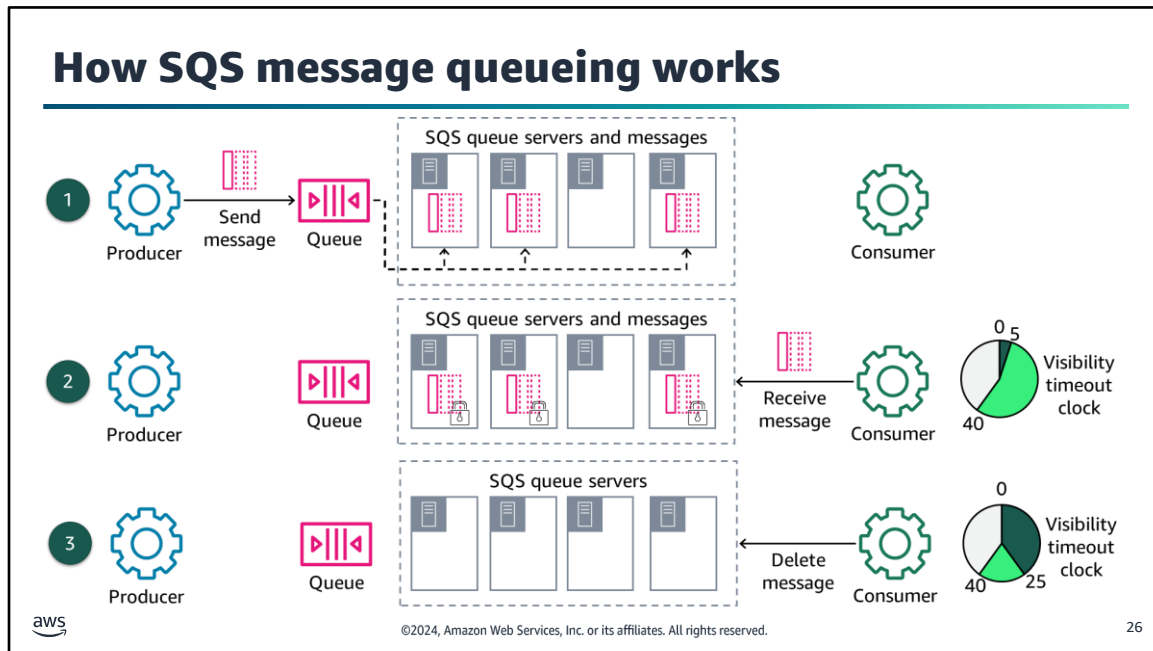


Another parameter that you should configure appropriately when creating an SQS queue is the message *visibility timeout*.

Visibility timeout is the period of time when Amazon SQS prevents other consumers from receiving and processing the same message. The timeout helps ensure that a message does not get processed multiple times and cause duplication. During the visibility timeout, the consumer that received the message should process it and then delete it from the queue. If the consumer fails to process and delete the message before the visibility timeout expires, the message becomes visible to other consumers and might be processed again. Typically, you should set the visibility timeout to the maximum time that it takes your application to process and delete a message from the queue. The default visibility timeout for a message is 30 seconds. The maximum is 12 hours. As an example, if it takes your application 15 seconds at the most to process and delete a message, you should set the visibility timeout to 15 seconds.

The diagram illustrates how the visibility timeout affects the access to the same message in a queue. The icons that are labeled 1, 2, or 3 are messages. The flow of events is as follows:

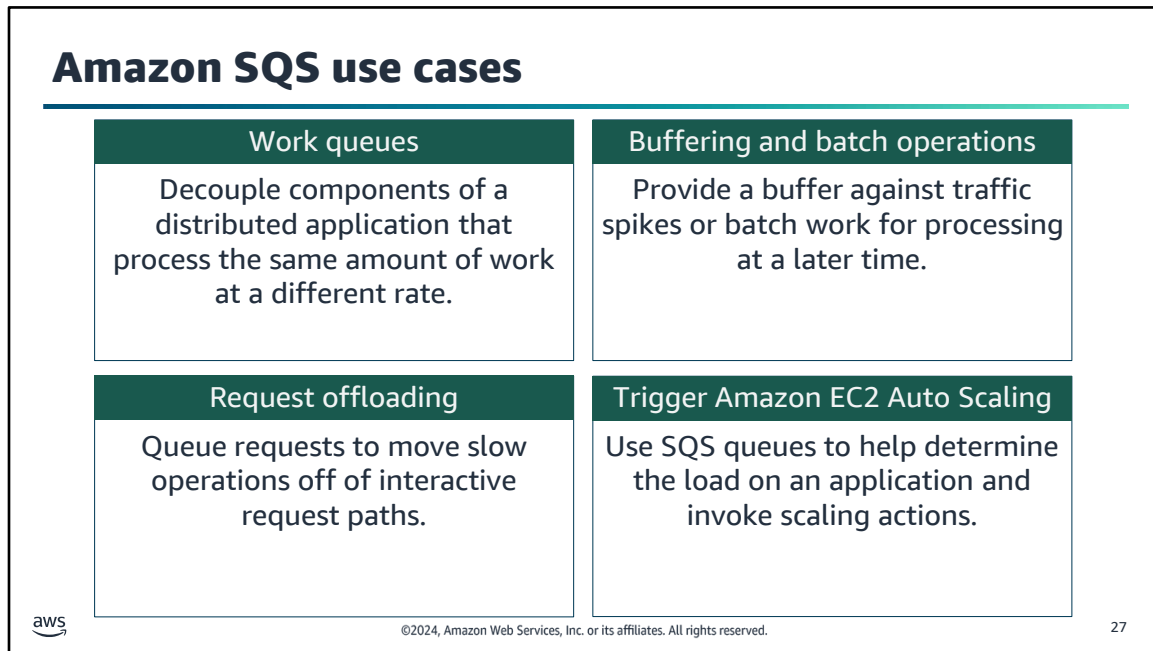
1. A consumer request the next message, message 1, from the queue.
2. The queue returns message 1 to the consumer, starts the visibility timeout countdown, and makes the message invisible to other consumers.
3. A second consumer requests the next message from the queue which is still message 1. Because the visibility timeout has not been reached yet, message 1 is still invisible to the second consumer and not delivered.



The following scenario illustrates the lifecycle of a message in an SQS queue and how Amazon SQS messaging works:

1. First, a producer sends a message to a queue, and the message is distributed across the queue's servers redundantly.
2. When a consumer is ready to process the message, it retrieves the message from the queue, and the visibility timeout period starts. While the message is being processed, it remains in the queue. During the visibility timeout, other consumers cannot process the message. In this example, the visibility timeout is set at 40 seconds.
3. After processing the message, the consumer deletes the message from the queue during the visibility timeout period. This action prevents the message from being received and processed again when the visibility timeout expires. Amazon SQS doesn't automatically delete the message. Because Amazon SQS is a distributed system, there is no guarantee that the consumer actually receives the message (for example, because of a connectivity issue or an issue in the consumer application). Therefore, the consumer must delete the message from the queue after receiving and processing it.

If the visibility timeout expires and the consumer that received the message didn't delete it, the message becomes visible again to be consumed.



Using a message queue is a great fit for asynchronous service-to-service communication. Some specific use cases for Amazon SQS include the following:

- **Work queues:** Decouple components of a distributed application that might not all process the same amount of work at the same rate. You can place work in a queue that multiple consumers (workers) in an Auto Scaling group can process. The consumers can scale up and down based on workload and latency requirements. For example, consider a car navigation system that collects data from thousands of cars to provide up-to-date road condition and routing information. This information is stored in a map database in the cloud. Because the sensors on a car send the data in real time and with a high-frequency, the data is first stored as messages in a queue. The application component that updates the database retrieves and processes the messages from the queue at its own pace. It can scale up or down based on the desired targets for the performance of the database refresh.
- **Buffering and batch operations:** Add scalability and reliability to your architecture, and smooth out temporary volume spikes without losing messages or increasing latency. Batching message requests gives a consumer the ability to process them in bulk and at a later time than when they were created. For example, a stock trading application can separate the capturing of real-time trade transactions from the update of the corresponding customer portfolio balances. During the day, the transactions are captured and batched into a queue. At night, the batch is processed to update customer balances.
- **Request offloading:** Queue requests to move slow operations off of interactive request paths. For example, in a banking application, separate the frontend application that a customer can use to make a bill payment from the backend application that processes it. In this way, the customer gets an immediate response, and the bill payment is processed in the background.



- **Trigger Amazon EC2 Auto Scaling:** Use SQS queues to help determine the load on an application. When they are combined with Amazon EC2 Auto Scaling, you can scale the number of EC2 instances out or in depending on the volume of traffic measured by the number of messages in the queue. For example, consider an order fulfillment application that retrieves and processes order messages from a queue. The application runs in an Amazon EC2 Auto Scaling group. If you want to provide a service level such that there are no more than 10 orders waiting in the queue at any given time, you can create an Amazon CloudWatch alarm that triggers the addition of another instance to the Auto Scaling group when the queue's *NumberOfMessagesSent* metric is greater than 10.

There are scenarios when using a queue is not recommended. They include the following:

- **Selecting specific messages:** You might want to selectively receive messages from a queue that match a particular set of attributes or match a one-time logical query. For example, a service requests a message with a particular attribute because it contains a response to another message that the service sent out. In this scenario, the queue can have messages in it that no one is polling for and are never consumed.
- **Managing large messages:** Most messaging protocols and implementations work best with reasonably sized messages (in the tens or hundreds of KBs). As message sizes grow, it's best to use a dedicated storage system (such as Amazon S3) and pass a reference to an object in the store that is in the message itself.

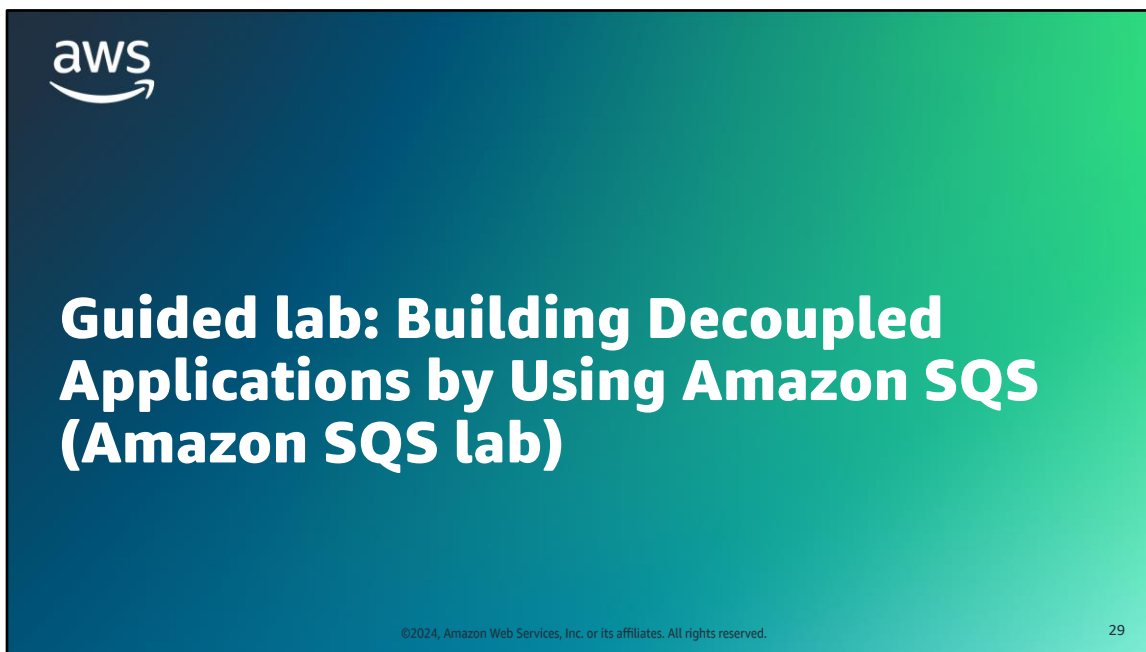
## Key takeaways: Decoupling applications with Amazon SQS



- Amazon SQS is a fully managed message-queuing service that you can use to decouple application components so that they run independently.
- Amazon SQS supports standard and FIFO queues.
- Messages that cannot be processed can be sent to a dead-letter queue.
- Long polling helps reduce the cost of using Amazon SQS by reducing the number of empty responses to a consumer's receive message request.
- A producer sends a message to a queue. A consumer processes and deletes the message during the visibility timeout period.



©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

28



You will now complete a lab. The next slides summarize what you will do in the lab, and the lab environment includes detailed instructions.

## Amazon SQS lab tasks



In this lab, you will perform the following main tasks:

- Create and configure an SQS queue.
- Use the queue in an application designed with decoupled functions.

©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

30

Access the lab environment through your online course to get additional details and complete the lab.

## Debrief: Amazon SQS lab

---

- How did you configure the SNS topic to send messages to the desired locations?
- How did you configure Amazon S3 to notify Amazon SNS when a new image is uploaded?



©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

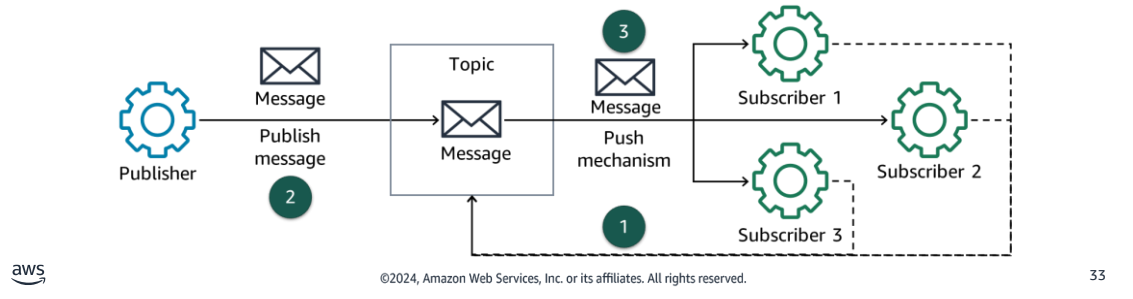
31



This section describes how Amazon SNS works to decouple tightly coupled architectures and when to use this service.

## Publish/subscribe messaging

- You can decouple applications asynchronously by using publish/subscribe (pub/sub) messaging.
- Use pub/sub messaging when the sending application sends a message to multiple receiving applications and has little or no knowledge about the receiving applications.
- The sending application is called a *publisher*.
- The receiving application is called a *subscriber*.
- Pub/sub messaging uses a *topic* to decouple applications.



Publish/subscribe (pub/sub) messaging is a messaging model that can be used to decouple applications asynchronously. It provides instant event notifications for distributed applications and permits messages to be broadcast to different parts of a system asynchronously.


The sending application is called a *publisher* because it sends messages to one or more recipients and does not necessarily know anything about them. The publisher publishes the messages to an interim destination called a *topic*. The receiving applications are called *subscribers* because they express interest in receiving messages from the publisher by subscribing to the topic. The messages in a topic are automatically broadcasted to the subscribers asynchronously. This approach is called a *push mechanism* and eliminates the need to periodically check or poll for new messages.

Unlike message queues, which store messages until they are retrieved and deleted, message topics transfer messages with no or little queueing and push them out immediately to all subscribers. The subscribers often perform different functions and can each do something different with the message in parallel. The publisher doesn't need to know who is using the information that it broadcasts, and the subscribers don't need to know who the message comes from. This style of messaging is a little different from message queues, where the component that sends the message often knows the destination it is sending to.

The diagram on the slide illustrates how pub/sub messaging works, which is as follows:

1. A recipient that is interested in receiving messages from a topic subscribes to the topic. In this case, the recipients are Subscribers 1, 2, and 3.
2. The publisher publishes a message to the topic.
3. The topic pushes the messages to the subscribers of the topic.

## Amazon Simple Notification Service (Amazon SNS)



Amazon SNS

- Is a fully managed pub/sub messaging service
- Helps decouple applications through notifications
- Provides highly scalable, secure, and cost-effective notification capabilities
- Provides an AWS Management Console interface and a web services API

aws

©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

34

Amazon SNS is a web service that you can use to set up, operate, and send notifications from the cloud. The service follows the pub/sub messaging model, where notifications are delivered to clients by using a push mechanism.

Amazon SNS is designed to meet the needs of the largest and most demanding applications, and it permits applications to publish an unlimited number of messages at any time. With no maintenance or management overhead and pay-as-you-go pricing, Amazon SNS gives developers a mechanism to incorporate a powerful notification system with their applications.

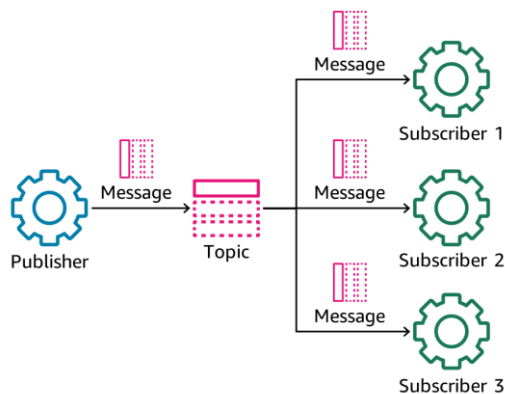
When you use Amazon SNS, you create a topic and set policies that restrict who can publish or subscribe to the topic. A publisher sends messages to topics that they have either created or that they have permission to publish to. Amazon SNS matches the topic to a list of subscribers who have subscribed to that topic and delivers the message to each of those subscribers. Each topic has a unique name that identifies the Amazon SNS endpoint for publishers to post messages and subscribers to register for notifications. Subscribers receive all messages that are published to the topics that they subscribe to, and all subscribers to a topic receive the same messages. In addition, both publishers and subscribers can use TLS to help secure the channel to send and receive messages.

Amazon SNS provides durable storage of a message while it processes the message. Upon receiving a publish request, Amazon SNS stores multiple copies of the message to disk across multiple Availability Zones before acknowledging receipt of the request to the publisher. Amazon SNS deletes the message after publishing it to the subscribers. Unlike Amazon SQS, Amazon SNS does not persist messages.

You can access Amazon SNS by using the AWS Management Console to create topics and subscriptions, and test message publication. You can also access Amazon SNS programmatically by using an API through the AWS SDKs.



## Types of subscribers



- Email destination
- Mobile text messaging destination
- Mobile push notifications endpoint
- HTTP or HTTPS endpoint
- AWS Lambda function
- SQS queue
- Amazon Kinesis Data Firehose delivery stream

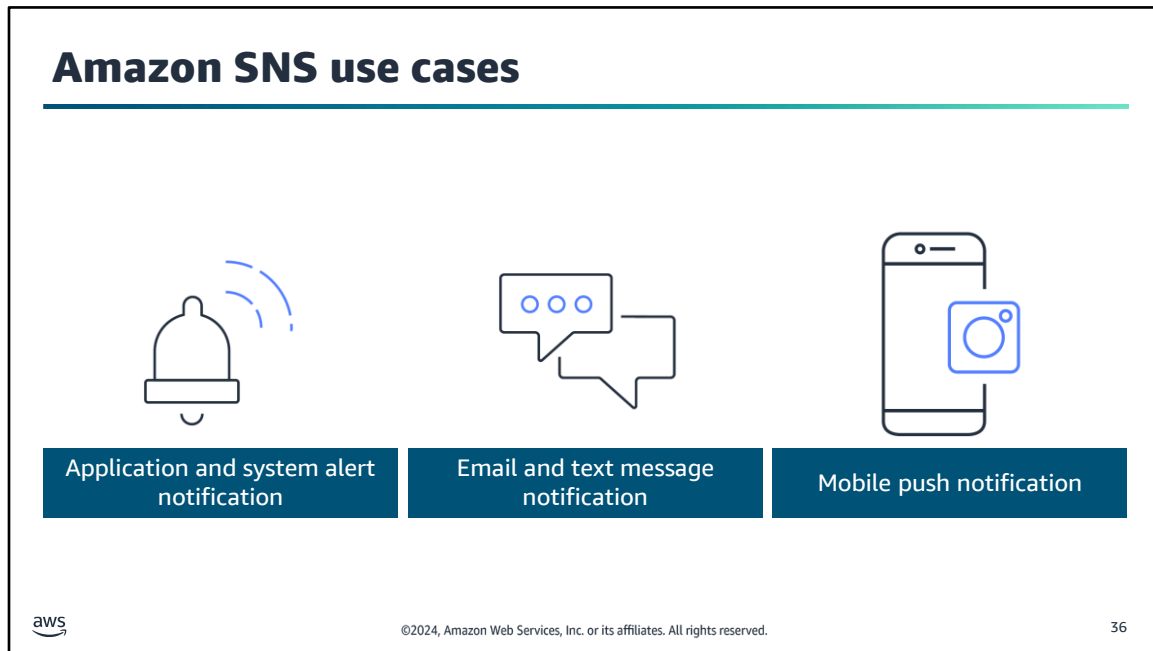


©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

35

Amazon SNS supports multiple destinations for delivering messages, including the following:

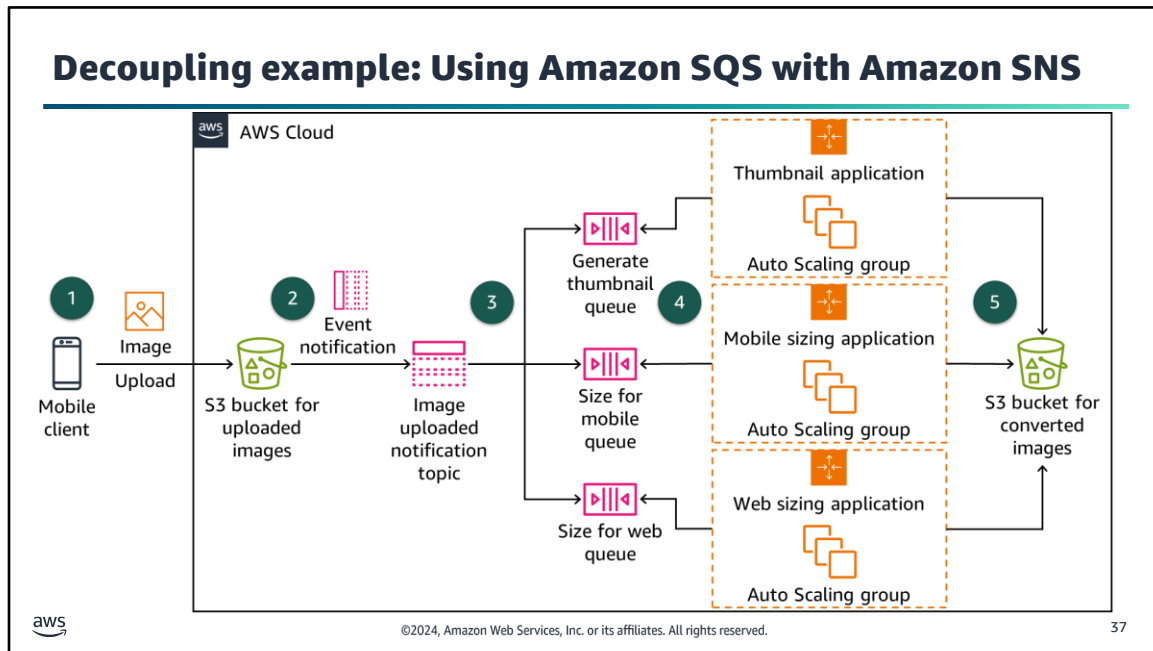
- Email destination: Amazon SNS sends the message to a registered email address. The message can be sent in a text format or as a JSON object.
- Mobile text messaging destination: Amazon SNS sends the message to a registered phone number as a Short Message Service (SMS) text message.
- Mobile push notifications endpoint: Amazon SNS sends the message to a mobile device endpoint as a native push notification.
- HTTP or HTTPS endpoint: Amazon SNS sends the message to a URL address by using an HTTP POST request.
- AWS Lambda function: Amazon SNS sends the message to a Lambda function to invoke the custom business logic to run.
- SQS queue: Amazon SNS sends the message to an SQS queue for a receiver application to process at a later time.
- Amazon Kinesis Data Firehose delivery stream: Amazon SNS sends the message to a Firehose delivery stream for delivery to storage and analytics endpoints.



You can use Amazon SNS to support a wide variety of needs, including event notification, workflow systems, mobile applications, and any other application that generates or consumes notifications.

Some specific uses cases for Amazon SNS include the following:

- **Application and system alerts:** You can use Amazon SNS to receive immediate notification when an event occurs, such as a change to an Auto Scaling group.
- **Push email and text messaging:** You can use Amazon SNS to push targeted news headlines to subscribers by email or SMS.
- **Mobile push notifications:** You can use Amazon SNS to send notifications to an application (for example, to indicate that an update is available). The notification message can include a link to download and install the update.



Using highly available services such as Amazon SNS and Amazon SQS to perform message routing is an effective way to distribute messages to decoupled application functions.

This example illustrates a fanout scenario that uses Amazon SNS and Amazon SQS. In a fanout scenario, a message is sent to an SNS topic and is then replicated and pushed to multiple SQS queues, HTTP endpoints, or email addresses. This can be used for parallel asynchronous processing.

In this example, a mobile client uploads an image to an S3 bucket to convert it into three different sizes that are suitable for a thumbnail, a mobile application, and a web application. The flow of the use case is as follows:

1. The mobile application uploads the image to an S3 bucket.
2. Once the image is uploaded to the bucket, the event notification feature of Amazon S3 invokes a notification that publishes a message to an SNS topic. The message contains the object URL of the image in Amazon S3.
3. The topic simultaneously delivers the message (fans out) to three different queues: generate thumbnail queue, size for mobile queue, and size for web queue. Each queue is observed by different applications running in their own Auto Scaling group.
4. The thumbnail application polls the generate thumbnail queue, the mobile sizing application polls the size for mobile queue, and the web sizing application polls the size for web queue.
5. Each application performs the intended resizing function independently of each other and stores the result in a converted images S3 bucket.

## Amazon SNS considerations

### Message publishing

- Single published message
- No recall options

### Message delivery

- Use a *standard* topic if the message delivery order does not matter.
- Use a *FIFO* topic if an exact message delivery order is required.
- Customize the delivery policy of an HTTP or HTTPS endpoint to control the retry behavior.



©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

38

The following are important considerations when using Amazon SNS:

- Each notification message contains a single published message.
- When a message is delivered successfully, there is no way to recall it.
- Amazon SNS offers two types of topics: standard and FIFO. With a standard topic, Amazon SNS will attempt to deliver messages from the publisher in the order that the messages were published to the topic. However, network issues could potentially result in out-of-order messages at the subscriber end. If you require strict message ordering, use a FIFO topic. A FIFO topic delivers messages in the order that the messages were published.
- Amazon SNS defines a *delivery policy* to control the retry pattern for each delivery protocol that it supports. This policy controls how Amazon SNS retries the delivery of messages when the system that hosts the subscribed endpoint becomes unavailable. When the delivery policy is exhausted, Amazon SNS stops retrying the delivery and discards the message unless a dead-letter queue is attached to the subscription. You cannot modify the delivery policies defined by Amazon SNS with the exception of the delivery policy for an HTTP or HTTPS endpoint. For such an endpoint, you could, for example, customize the policy based on your HTTP server's capacity.

## Amazon SNS and Amazon SQS comparison

	Amazon SNS	Amazon SQS
<b>Messaging Model</b>	Publisher-Subscriber	Producer-Consumer
<b>Distribution Model</b>	One to many	One to one
<b>Delivery Mechanism</b>	Push (passive)	Pull (active)
<b>Message Persistence</b>	No	Yes



©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

39

This table summarizes the key differences between Amazon SNS and Amazon SQS:

- Amazon SNS uses a publisher-subscriber messaging paradigm where the publisher does not necessarily know anything about the subscriber.
- Amazon SQS uses a producer-consumer messaging paradigm where the producer knows the consumer of the message.
- An Amazon SNS publisher can send a message to one or more subscribers. The subscribers must be available at the time of publication to receive the message.
- An Amazon SQS producer sends a message to only one consumer. The consumer does not need to be available at the time the message is sent.
- Amazon SNS sends messages through a push mechanism.
- Amazon SQS sends messages through a pull mechanism.
- Amazon SNS messages are not persistent and are deleted after they are published.
- Amazon SQS messages persist in the queue until the consumer deletes them or the message retention limit is reached.

## Key takeaways: Decoupling applications with Amazon SNS



- Amazon SNS is a web service that you can use to set up, operate, and send notifications from the cloud.
- Amazon SNS follows the pub/sub messaging paradigm.
- To use Amazon SNS, you create a topic, create subscribers for the topic, and publish messages to the topic.
- You can use topics to decouple message publishers from subscribers and fanout messages to multiple recipients at one time.
- AWS services can publish messages to an SNS topic to invoke event-driven computing and workflows.

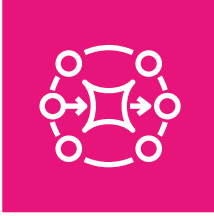
©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

40



This section provides an introductory description of the Amazon MQ service.

## Amazon MQ



Amazon MQ

- Is a fully managed message broker service
- Facilitates the setup, operation, and management of an Apache ActiveMQ or RabbitMQ message broker in the AWS Cloud
- Provides a queue- and topic-based solution for loosely coupling applications
- Enables software applications and components to communicate by using various programming languages, operating systems, and formal messaging protocols

aws

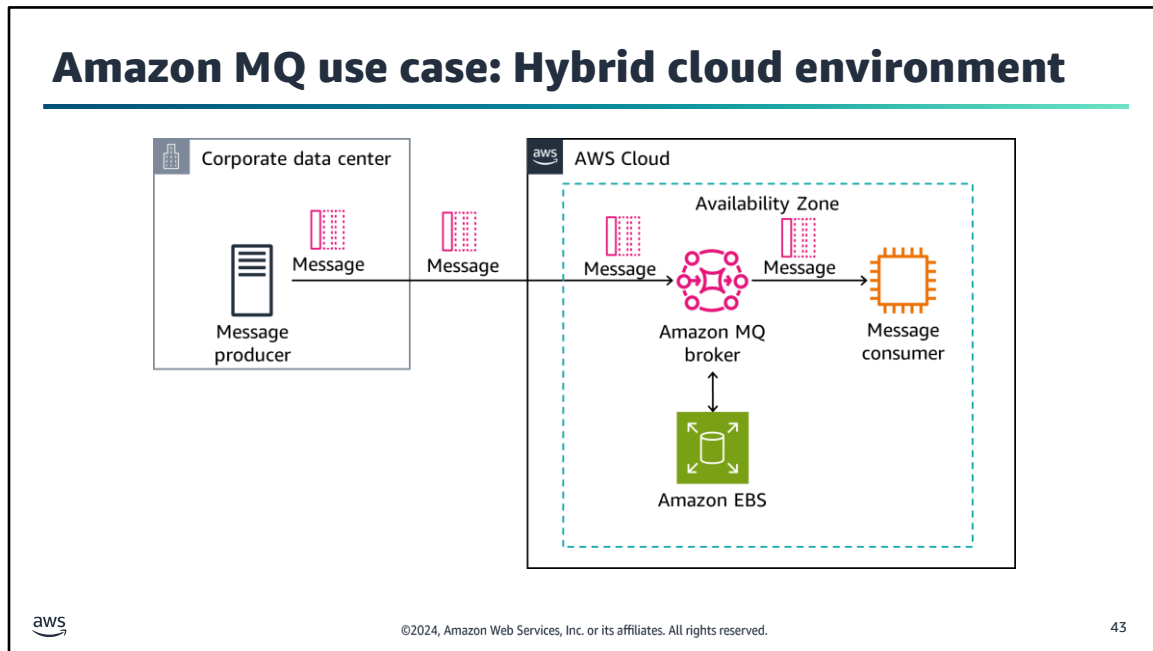
©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

42

Amazon MQ is a managed message broker service for Apache ActiveMQ and RabbitMQ that you can use to set up and operate message brokers in the cloud. ActiveMQ and RabbitMQ are popular open source message brokers. A message broker lets different software systems, which often use different programming languages on different platforms, communicate and exchange information. As a fully managed service, Amazon MQ reduces your operational responsibilities by managing the provisioning, setup, and maintenance of a message broker.

You can use Amazon MQ to integrate heterogeneous systems in a loosely coupled manner by using queues and topics. Amazon MQ connects to your existing applications with industry-standard messaging APIs and protocols that include Java Message Service (JMS), .NET Message Service (NMS), Advanced Message Queuing Protocol (AMQP), Streaming Text Oriented Messaging Protocol (STOMP), Message Queuing Telemetry Transport (MQTT), and WebSocket. You can migrate from any message broker that uses these standards to Amazon MQ, usually without the need to rewrite any messaging code. In most cases, you can update the endpoints of your applications to connect to Amazon MQ and start sending messages.





Many organizations, particularly enterprises, rely on message brokers to connect and coordinate different systems. Message brokers let distributed applications communicate with each other. Brokers serve as the technological backbone for an enterprise's IT environment and, ultimately, business services.

In many cases, these organizations have started to build new cloud-centered applications or to lift and shift applications to AWS. There are some applications, such as mainframe systems, that are too costly to migrate. In these cases, the on-premises applications must still interact with cloud-based components.

Amazon MQ gives organizations the ability to send messages between applications in the cloud and applications that are on premises to enable hybrid environments and application modernization. For example, you can invoke a Lambda function from queues and topics that are managed by Amazon MQ brokers to integrate legacy systems with serverless architectures.

In this example, an on-premises producer application wants to send messages to a consumer application running in the AWS Cloud. To support this requirement, the systems administrator creates a single Amazon MQ for ActiveMQ broker in an Availability Zone. The administrator also configures the broker to store messages in an EBS volume optimized for low latency and high throughput. The producer can then send messages to the ActiveMQ broker, which propagates them to the message consumer.

For more information on the Amazon MQ ActiveMQ and RabbitMQ engines and their different configurations, see the *Amazon MQ Developer Guide*. A link is provided in your course resources.

## Choosing the right solution for decoupling

	Amazon SQS	Amazon SNS	Amazon MQ
Applicability	Cloud-centered applications	Cloud-native applications	<ul style="list-style-type: none"><li>Hybrid applications</li><li>Message broker migration</li></ul>
Messaging Model	Producer-Consumer	Publisher-Subscriber	<ul style="list-style-type: none"><li>Producer-Consumer</li><li>Publisher-Subscriber</li></ul>
Programming API	Amazon SQS API	Amazon SNS API	<ul style="list-style-type: none"><li>Industry standard message broker APIs</li></ul>
Pricing Model	Pay per request	Pay per request	<ul style="list-style-type: none"><li>Pay per hour</li><li>Pay per GB</li></ul>



©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

44

If you need to integrate on-premises applications with cloud applications, AWS recommends using Amazon MQ. It supports open standard APIs and protocols, such as JMS and AMQP, and gives your on-premises applications the ability to communicate with cloud applications without the need to rewrite the messaging code. Similarly, if you use messaging in existing applications and want to move your message brokers to the cloud, Amazon MQ is the recommended solution. You can switch from any standards-based message broker to Amazon MQ to reduce broker maintenance, reduce licensing costs, and improve broker stability. With Amazon MQ, you pay for the time that your message broker instance runs, the storage that you use monthly, and data transfer fees.

If you are building new applications in the cloud, AWS recommends using Amazon SQS and Amazon SNS. They are queue and topic services that use APIs and don't require you to set up message brokers. Amazon SQS pricing is based on the number of monthly API requests made and data transfer fees. Amazon SNS pricing is based on the number of monthly API requests made, the number of deliveries to various endpoints, and data transfer fees.

## Key takeaways: Decoupling a hybrid application with Amazon MQ



- Amazon MQ is a managed service that you can use to set up and operate Apache ActiveMQ and RabbitMQ message brokers in the cloud.
- Amazon MQ is compatible with open standard messaging APIs and protocols.
- You can use Amazon MQ to integrate on-premises and cloud environments in a loosely coupled manner.
- You can also use Amazon MQ to migrate your on-premises open source message brokers to the AWS Cloud.

©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

45



This section summarizes what you have learned and brings the module to a close.

## Module summary

---

This module prepared you to do the following:

- Differentiate between tightly and loosely coupled architectures.
- Identify how Amazon SQS works and when to use it.
- Identify how Amazon SNS works and when to use it.
- Describe Amazon MQ.
- Decouple workloads by using Amazon SQS.



©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

47

## Considerations for the café

---



- Discuss how you as a cloud architect might advise the café based on the key cloud architect concerns presented at the start of this module.



©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

48

## Module knowledge check



- The knowledge check is delivered online within your course.
- The knowledge check includes 10 questions based on material presented on the slides and in the slide notes.
- You can retake the knowledge check as many times as you like.

©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

49

Use your online course to access the knowledge check for this module.

## Sample exam question

A company must perform asynchronous processing and implement Amazon Simple Queue Service (Amazon SQS) as part of a decoupled architecture. The company wants to ensure that the number of empty responses from polling requests is kept to a minimum.

What should a solutions architect do to ensure that empty responses are reduced?

Identify the key words and phrases before continuing.

The following are the key words and phrases:

- SQS queue
- Empty responses kept to a minimum



©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

50

The key words identified on the slide pull out the elements of the question that are most important in selecting the correct answer.




## Sample exam question: Response choices

A company must perform asynchronous processing and implement Amazon Simple Queue Service (Amazon SQS) as part of a decoupled architecture. The company wants to ensure that the number of empty responses from polling requests is kept to a minimum.

What should a solutions architect do to ensure that empty responses are reduced?

Choice	Response
A	Increase the maximum message retention period for the queue.
B	Increase the maximum receives for the redrive policy for the queue.
C	Increase the default visibility timeout for the queue.
D	Increase the receive message wait time for the queue.


 ©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved. 51

Use the key words that you identified on the previous slide, and review each of the possible responses to determine which one best addresses the question.

**Sample exam question: Answer**

The answer is D.

Choice	Response
D	Increase the receive message wait time for the queue

©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.52

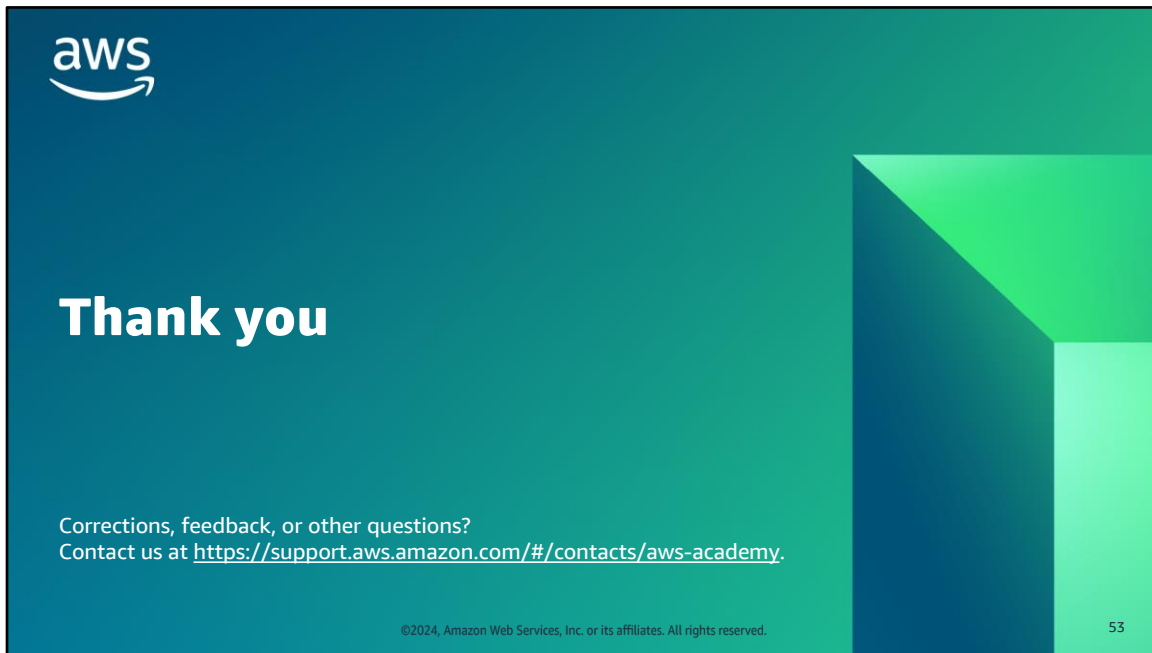
Empty polling means that there are no messages to be retrieved when the queue polls for new messages.

Choice A (Increase the maximum message retention period) would increase how long messages are retained on the queue but would not prevent empty responses if there are no new messages on the queue.

Choice B (Increase the maximum receives for the redrive policy for the queue) would increase the number of times Amazon SQS would try to deliver a message if it were failing, but this is unrelated to polling and finding no messages.

Choice C (Increase the default visibility timeout for the queue) might be used if your application weren't processing and deleting messages fast enough before they became visible to the queue again. This helps to prevent processing messages twice.

Choice D is correct. Increasing the receive message wait time increases the time between requests, which makes it more likely that additional messages would arrive in the interval between polls.



That concludes this module. The Content Resources page of your course includes links to additional resources that are related to this module.