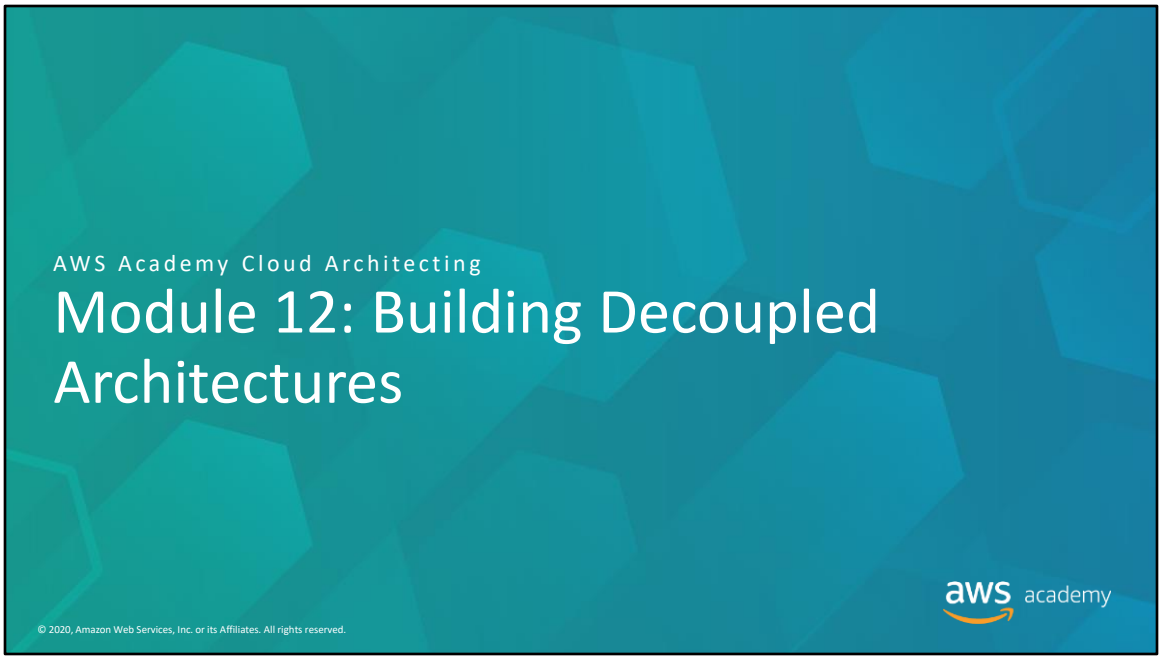


Welcome to Module 12: Building Decoupled Architectures.

Module overview

aws academy

Sections

1. Architectural need
2. Decoupling your architecture
3. Decoupling with Amazon Simple Queue Service (Amazon SQS)
4. Decoupling with Amazon Simple Notification Service (Amazon SNS)
5. Sending messages between cloud applications and on-premises with Amazon MQ

Knowledge check

This module includes the following sections:

1. Architectural need
2. Decoupling your architecture
3. Decoupling with Amazon Simple Queue Service (Amazon SQS)
4. Decoupling with Amazon Simple Notification Service (Amazon SNS)
5. Sending messages between cloud applications and on-premises with Amazon MQ

At the end of this module, you will be asked to complete a knowledge check that will test your understanding of key concepts covered in this module.

Module objectives

aws academy

At the end of this module, you should be able to:

- 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

At the end of this module, you should be able to:

- 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

Module 12: Building Decoupled Architectures

Section 1: Architectural need

aws academy

Introducing Section 1: Architectural need.

Café business requirement

aws academy

The café's architecture now supports hundreds of thousands of users. However, it's difficult to make changes to one layer of the application without affecting the other layers.

The café's architecture now supports hundreds of thousands of users. However, the café's systems are too tightly coupled. It's difficult to make changes to one layer of the application without affecting the other layers. For example, daily ordering reports are generated from the same web server that also serves the café's website to customers.

In addition, Frank mentioned that he's not getting the regular 17:00 report on Fridays. After some investigation, Sofía and Nikhil observe that the scheduled maintenance window coincides with the time that the reporting system attempts to generate the report.

They speak with Olivia, who recommends that they decouple the architecture. By moving the reporting process to another system, the reporting data will not be lost, even if the web server becomes temporarily unavailable. Also, the need to generate a report will be queued and handled.

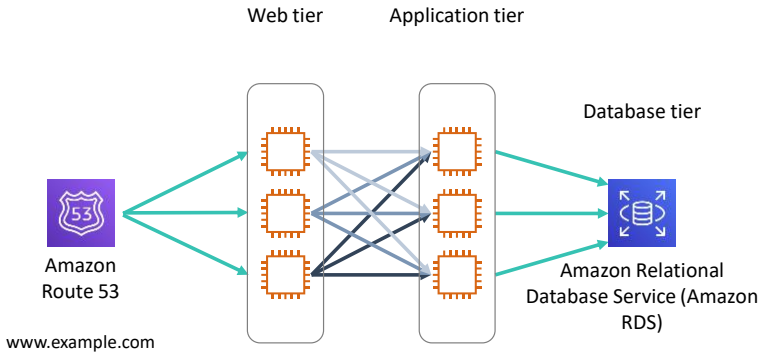
Module 12: Building Decoupled Architectures

Section 2: Decoupling your architecture

aws academy

Introducing Section 2: Decoupling your architecture.

Tightly coupled architectures

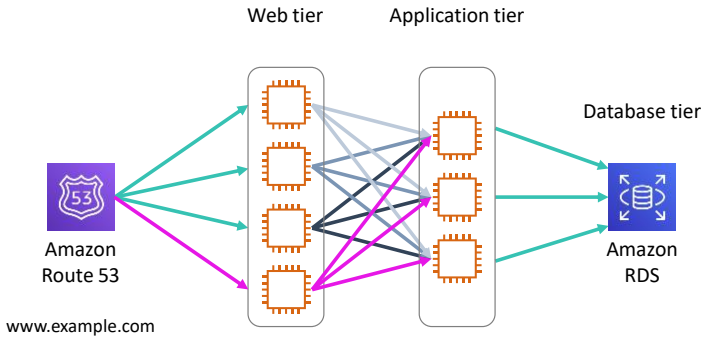


Components are strongly connected to each other.

Traditional infrastructures have chains of tightly integrated components. Each component has a specific purpose. When one component goes down, the disruption to the system can be fatal.

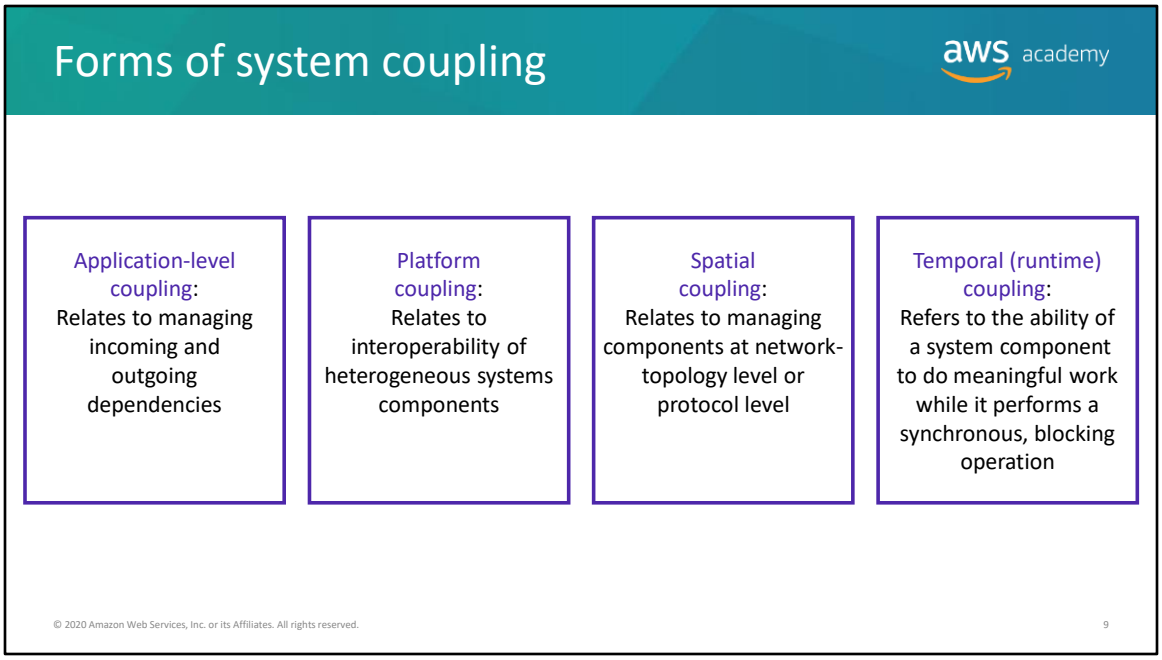
Consider this example of a three-tier architecture for a web application that processes customer orders. Each instance in the web tier communicates with each instance in the application tier. Each instance in the application tier persists data to a backend database. An instance failure in the web or application tiers would also cause failure in the persistence of some customer order data.

Tightly coupled architectures impede scaling



Adding resources increases complexity and impedes scaling.

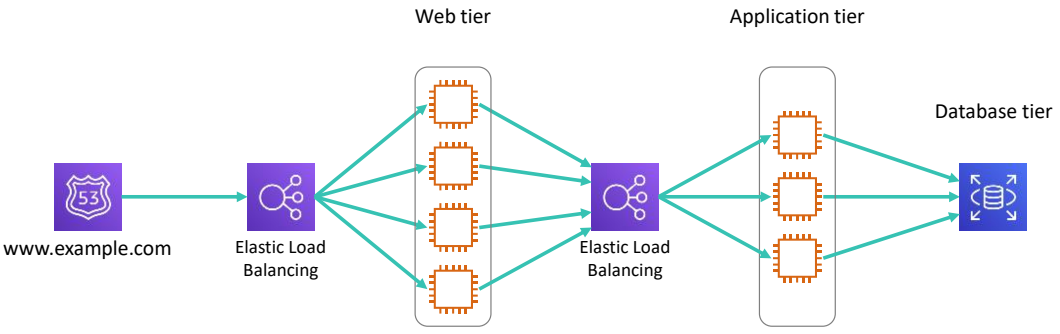
In a tightly coupled system, scaling is also impeded: if you add servers at one layer, you must also connect them to servers in every connecting layer. Continuing with the three-tier architecture example, an instance that's added to the web tier must be connected to every instance in the application tier.



A system can be coupled in many ways, and you should consider them when you build distributed applications in the cloud:

- *Application-level* coupling relates to managing incoming and outgoing dependencies
- *Platform* coupling relates to the interoperability of heterogeneous systems components
- *Spatial* coupling relates to managing components at a network-topology level or protocol level
- *Temporal* (or runtime) coupling refers to the ability of a system component to do meaningful work while it performs a synchronous, blocking operation

Loosely coupled architectures

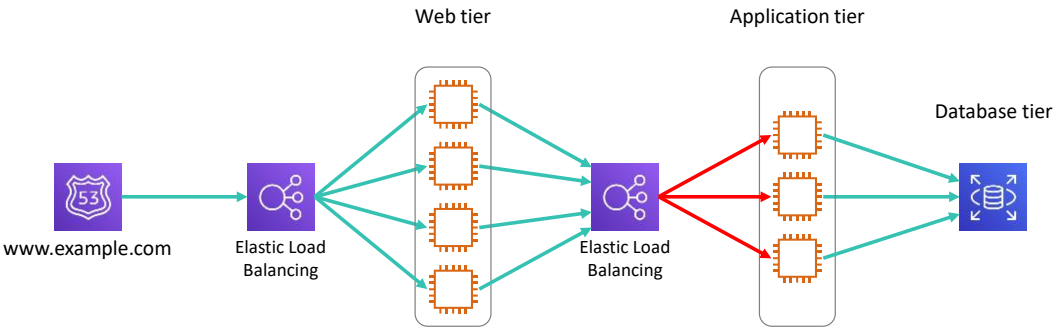


Use managed solutions as intermediaries between layers.

To help ensure that your application scales with increasing load and that there are no bottlenecks or single points of failure in your system, implement loose coupling. With loose coupling, you reduce dependencies in your system by using managed solutions as intermediaries between the layers of your system. This way, failures and the scaling of components or layers are automatically handled by the intermediary.

Consider the three-tier web application architecture again. You can implement loose coupling by adding load balancers in front of the web and application tiers to distribute traffic at each layer. If one server goes down, the load balancer will automatically direct traffic to the healthy instances.

Considerations in loosely coupled architectures



In the business use case of the order processing workflow, one potential point of vulnerability is saving order data to the database. If the business requires that order data must persist in the database, then various scenarios—such as a potential deadlock, race condition, or network issue—could cause the persistence of the order to fail. In this case, the order would be lost and you could not restore it.

Section 2 key takeaways

- Tightly coupled systems have chains of tightly integrated components and impede scaling
- You can implement loose coupling in your system by using managed solutions (such as Elastic Load Balancing) as intermediaries between layers

Some key takeaways from this section of the module include:

- Tightly coupled systems have chains of tightly integrated components and impede scaling
- You can implement loose coupling in your system by using managed solutions (such as Elastic Load Balancing) as intermediaries between layers

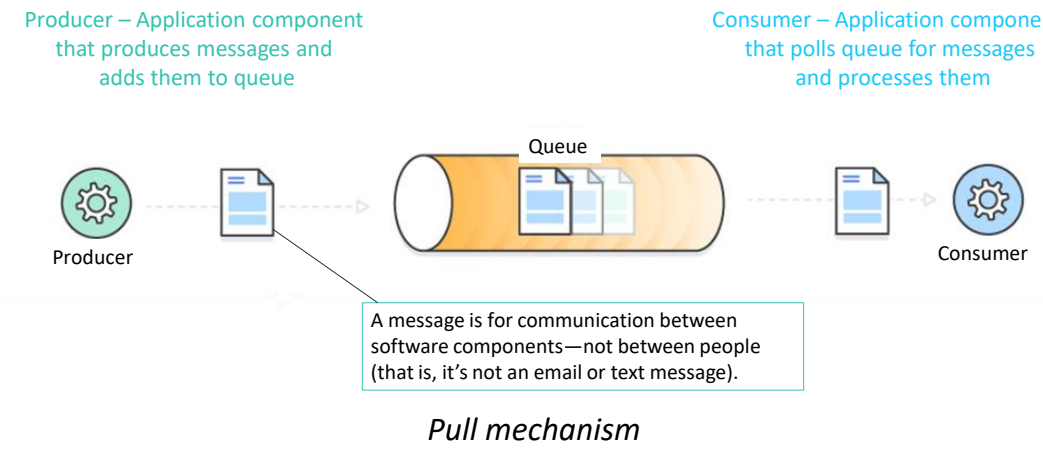
Module 12: Building Decoupled Architectures

Section 3: Decoupling with Amazon SQS

aws academy

Introducing Section 3: Decoupling with Amazon SQS.

Message queues for decoupling architectures



Message queues are another component that help you implement a decoupled architecture. Message queues provide communication and coordination for these distributed applications.

A message queue is a temporary repository for messages that are waiting to be processed. Messages are usually small, and can be things like requests, replies, error messages, or plain information. Examples of messages include customer records, product orders, invoices, patient records, among others.

To send a message, a component that is called a *producer* adds a message to the queue. The message is stored in the queue until another component that is called a *consumer* retrieves and processes it.

Amazon SQS

- Fully managed message queueing service
- Uses a pull mechanism
- Messages are encrypted and stored until they are processed and deleted
- Acts as a buffer between producers and consumers

Amazon Simple Queue Service (Amazon SQS) is a fully managed, message queuing service that enables you to decouple application components so they run independently. It lets web service applications queue messages that are generated by one application component to be consumed by another component.

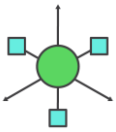
A *queue* is a temporary repository for messages that are waiting to be processed. Messages are stored until they are processed and deleted (from 1 through 14 days; the default is 4 days). Messages can contain up to 256 KB of text in any format. Amazon SQS works on a massive scale and processes billions of messages per day. It stores all message queues and messages within a single, highly available AWS Region with multiple redundant Availability Zones. No single computer, network, or Availability Zone failure can make messages inaccessible. Messages can be sent and read simultaneously.

You can securely share Amazon SQS queues anonymously or with specific AWS accounts. You can also restrict queue sharing by IP address and the time of day. Messages in SQS queues are encrypted with server-side encryption (SSE) by using keys that are managed in the AWS Key Management Service (AWS KMS). Amazon SQS decrypts messages only when they are sent to an authorized consumer.

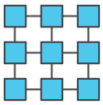
Amazon SQS supports multiple producers and consumers that interact with the same queue. Amazon SQS can be used with several AWS services, including: Amazon Elastic Compute Cloud (Amazon EC2), Amazon Simple Storage Service (Amazon S3), Amazon Elastic Container Service (Amazon ECS), AWS Lambda, and Amazon DynamoDB.

Achieve loose coupling with Amazon SQS

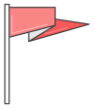
With Amazon SQS, you can:



Use asynchronous processing to get your responses from each step quickly



Handle performance and service requirements by increasing the number of job instances



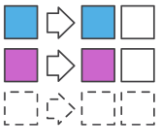
Easily recover from failed steps because messages will remain in the queue

Amazon SQS enables you to achieve loose coupling in your architecture.

- It performs asynchronous processing so that you can get responses from each step quickly
- It can handle performance and service requirements by increasing the number of job instances
- Your application can easily recover from failed steps because messages will remain in the queue

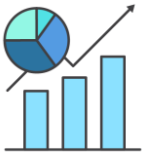
Amazon SQS general use cases

Work queues



Buffering batch operations

Request offloading



Trigger Amazon EC2 Auto Scaling

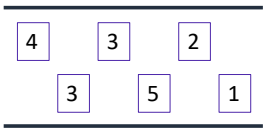
Amazon SQS can be used in many different ways:

- *Work queues* – Decouple components of a distributed application that might not all process the same amount of work simultaneously.
- *Buffering batch operations* – Add scalability and reliability to your architecture, and smooth out temporary volume spikes without losing messages or increasing latency.
- *Request offloading* – Queue requests to move slow operations off of interactive request paths
- *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.

Queue types

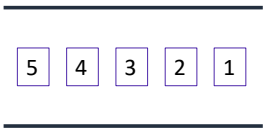
aws academy

Standard queues



- At-least-once delivery
- Best-effort ordering
- Nearly unlimited throughput

First in, first out (FIFO) queues



- First-in-first-out delivery
- Exactly once processing
- High throughput

There are two types of SQS queues:

Standard queues offer:

- 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 that they were sent in.
- Nearly unlimited throughput – Standard queues support a nearly unlimited number of transactions per section (TPS) per API action.

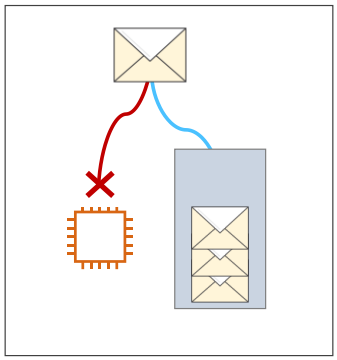
First in, first out (FIFO) queues:

- Are designed to guarantee that messages are processed exactly once, in the exact order that they are sent and received.
- Provide high throughput – FIFO queues support up to 300 messages per second (300 send, receive, or delete operations per second). When you batch 10 messages per operation (maximum), FIFO queues can support up to 3,000 messages per second.

Amazon SQS features (1 of 3)

aws academy

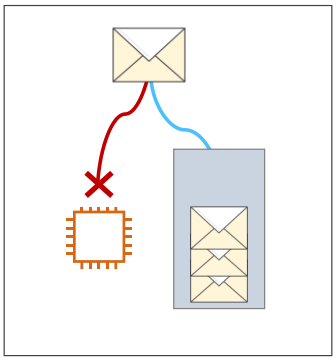
Dead-letter queue support



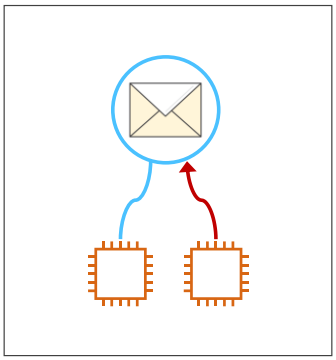
Amazon SQS features dead-letter queue support. A *dead-letter queue* (DLQ) is a queue of messages that could not be processed. It receives messages after the maximum number of processing attempts has been reached. A DLQ is like any other SQS queue: messages can be sent to it and received from it. You can create a DLQ from the Amazon SQS API and the console.

Amazon SQS features (2 of 3)

Dead-letter queue support



Visibility timeout



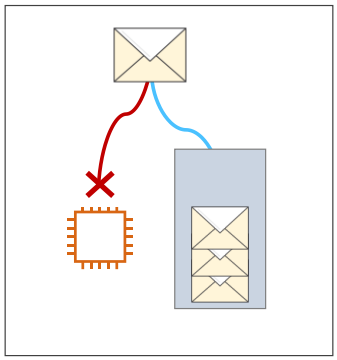
Another feature of Amazon SQS is the visibility timeout. *Visibility timeout* is the period of time when Amazon SQS prevents other consumers from receiving and processing the same message. The timeout helps ensure that a job does not get processed multiple times and cause duplication. During the visibility timeout, the component that received the message processes it and then deletes it from the queue. The default visibility timeout for a message is 30 seconds, and the maximum is 12 hours.

If the consumer fails to process and delete the message before the visibility timeout expires, the message becomes visible to other consumers and it 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.

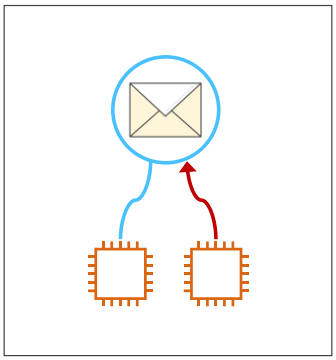
Amazon SQS features (3 of 3)

aws academy

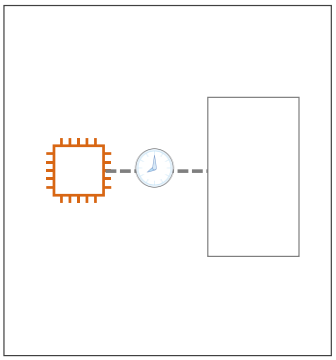
Dead-letter queue support



Visibility timeout



Long polling



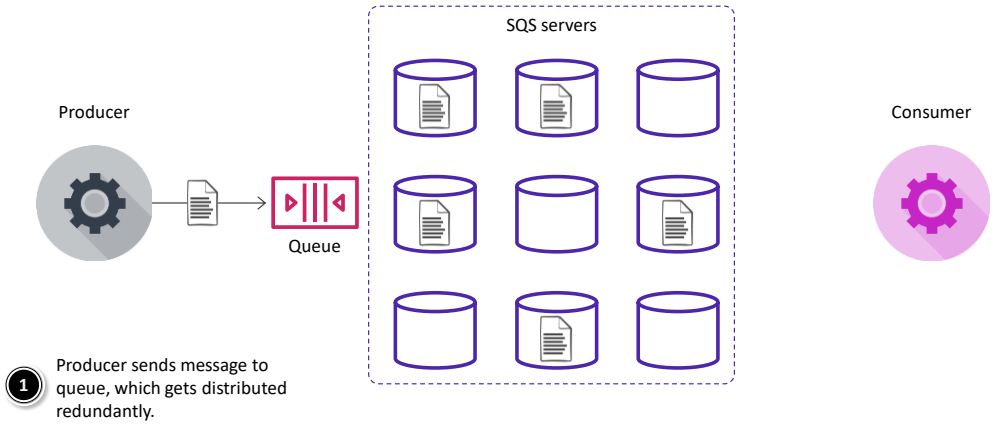
Finally, Amazon SQS supports both support short polling and long polling to retrieve messages from your SQS queues. By default, queues use short polling.

Short polling queries only a subset of the servers (based on a weighted random distribution) to find messages can be included in the response. Amazon SQS sends the response immediately, even if the query found no messages.

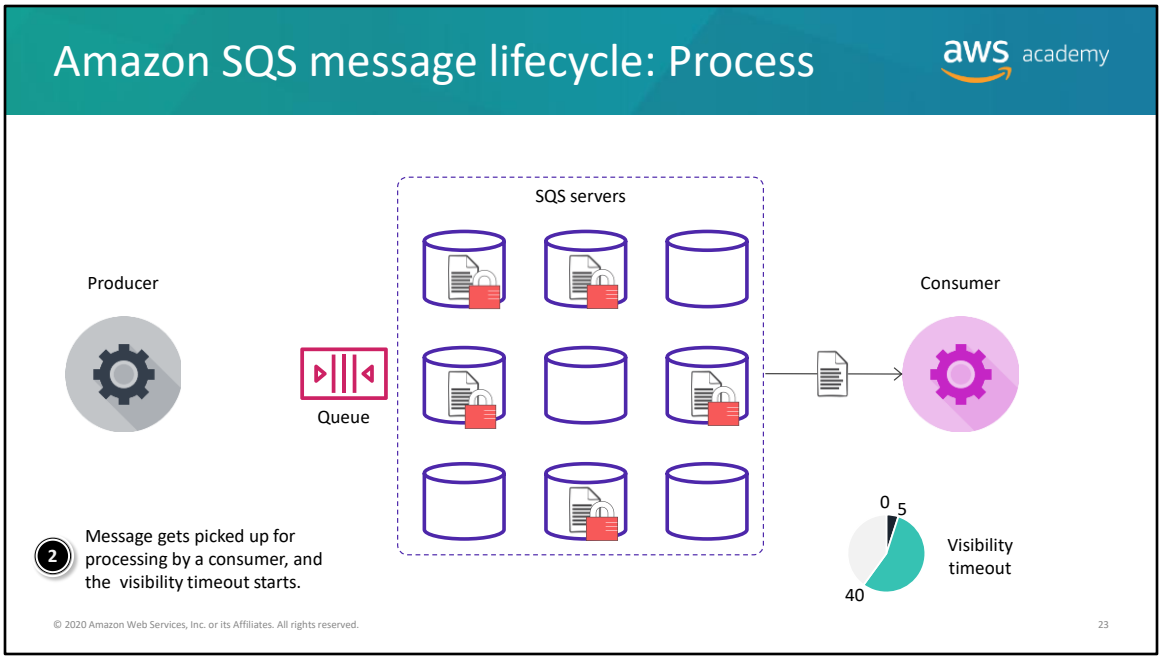
By contrast, long polling queries all the servers for messages. Amazon SQS sends the response after it collects the maximum number of messages for the response, or the polling wait time expires.

Long polling makes it inexpensive to retrieve messages from your SQS queue as soon as the messages are available. Long polling might reduce the cost of using Amazon SQS because you can reduce the number of empty receives.

Amazon SQS message lifecycle: Create

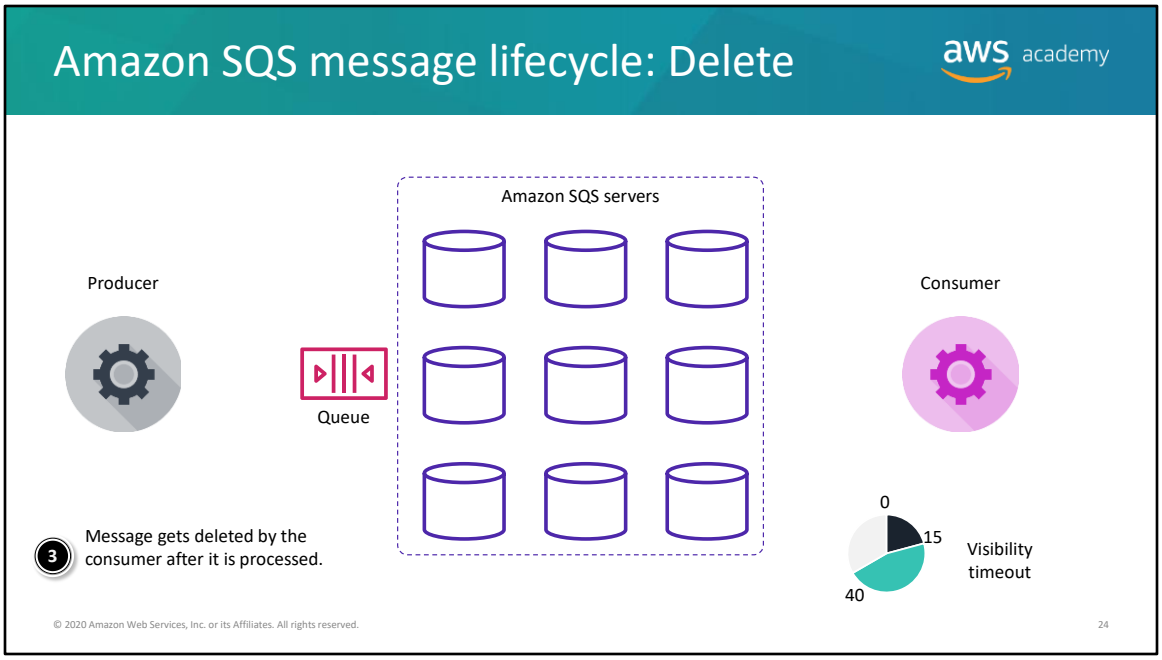


The lifecycle of a message in an SQS queue can be illustrated with the following scenario. First, a producer sends a message to a queue, and the message is distributed across the SQS servers redundantly.



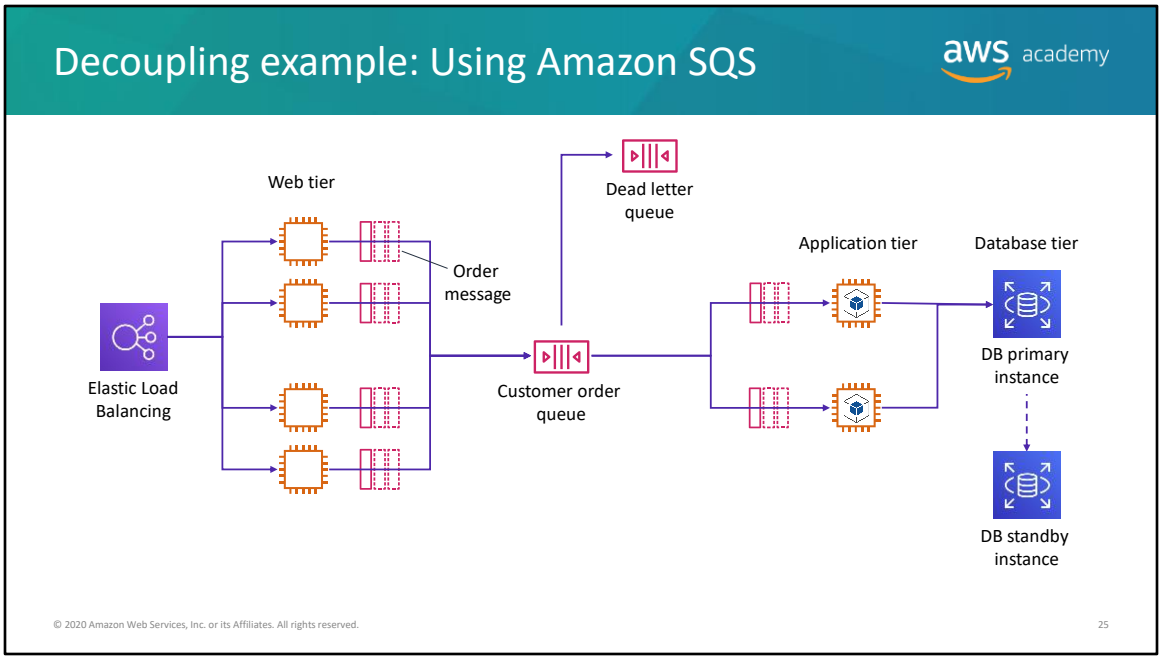
When a consumer is ready to process the message, it retrieves the message from the queue. While the message is being processed, it remains in the queue.

During the visibility timeout, other consumers cannot process the message. In this example, the visibility timeout is *40 seconds*.



After processing the message, the consumer deletes the message from the queue. 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's 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.

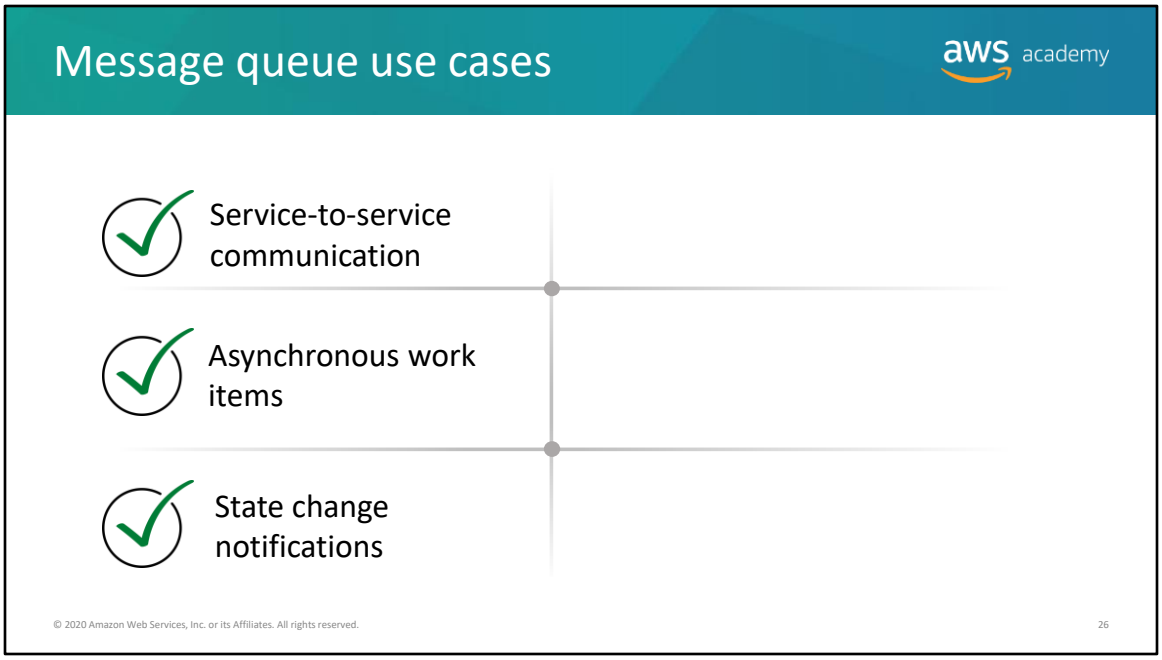


Again, consider the ordering processing application that you learned about in the previous section. You can decouple the architecture for this application by introducing an SQS queue. You can use the queue to isolate the processing logic into its own component so that it runs in a separate process from the web application.

This design enables the system to be more resilient to spikes in traffic. Further, it enables work to be performed only as fast as necessary to manage costs.

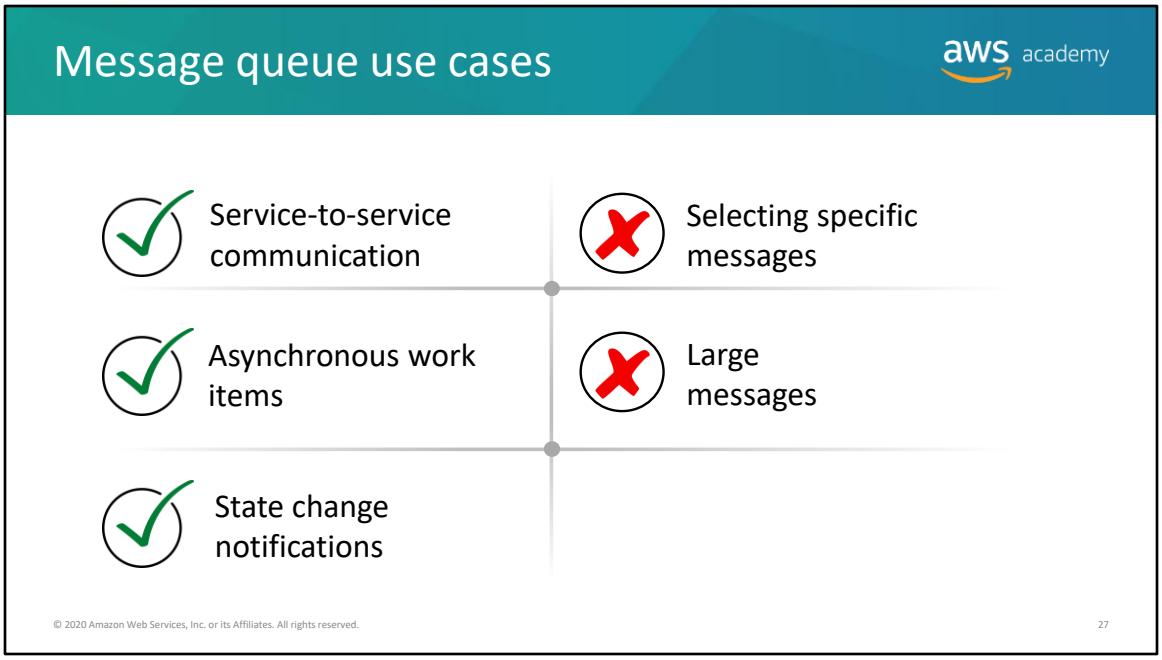
In addition, you now have a mechanism for persisting orders as messages (with the queue acting as a temporary database). You also moved the scope of your transaction with your database further down the stack. If an application exception or transaction fails, this design helps to ensure that the order processing can be retried or redirected to a dead letter queue for re-processing at a later stage.

For more information about this use case, see this AWS Compute Blog post.



These common use cases demonstrate where a message queue is a great fit:

- *Service-to-service communication* – For example, consider a frontend website that must update a customer's delivery address in a backend customer relationship management (CRM) service. You can have the code of the frontend website send messages to an SQS queue, and have the backend CRM service consume them.
- *Asynchronous work items* – For example, say that a hotel booking system must cancel a reservation, which is a process that takes a long time. You can send messages to an SQS queue and have the same hotel booking system consume those messages and perform asynchronous cancellations.
- *State change notifications* – For example, say that you have a service that manages some resource. You want other services to receive updates about changes to that resource. An inventory system might publish notifications when a certain item is low and needs to be ordered.



It's also important to know when a particular technology will *not* fit your use case. Messaging has its own set of commonly encountered anti-patterns.

- *Selecting specific messages* – You might want to selectively receive messages from a queue that match a particular set of attributes, or match an ad hoc 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.
- *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.

Section 3 key takeaways

- Amazon SQS is a fully managed, message-queuing service that enables you to decouple application components so they run independently.
- Amazon SQS supports standard and FIFO queues.
- A producer sends a message to a queue. A consumer processes and deletes the message during the visibility timeout.
- Messages that cannot be processed can be sent to a dead letter queue.
- Long polling is a way to retrieve a large number of messages from your SQS queues.

28

Some key takeaways from this section of the module include:

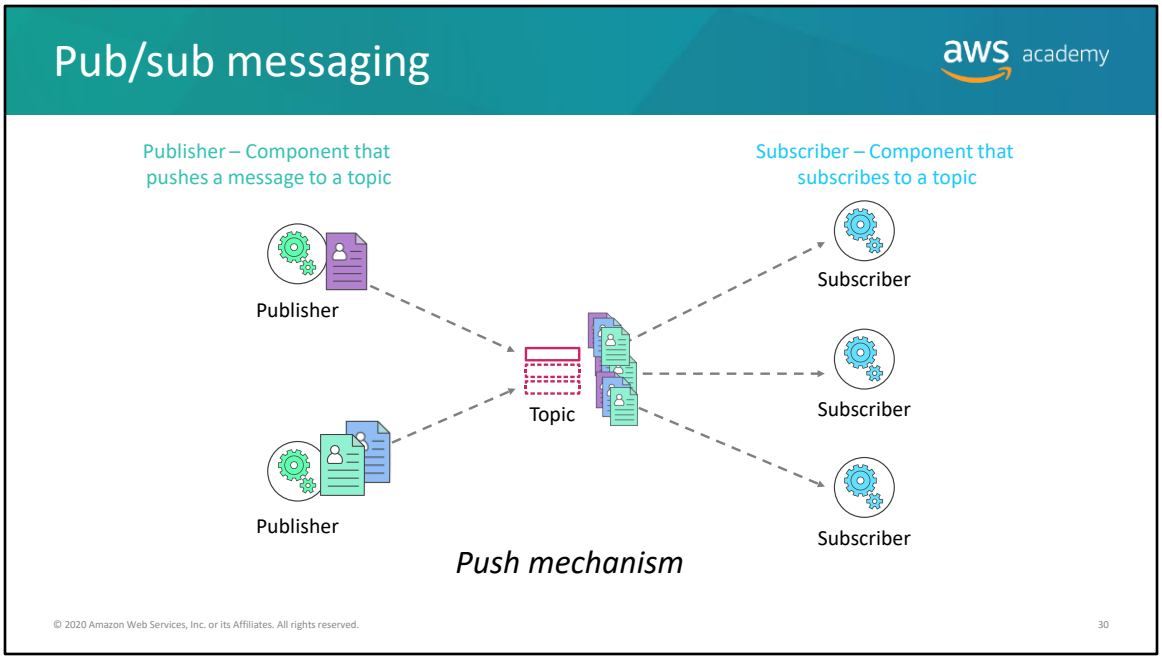
- Amazon SQS is a fully managed, message-queuing service that enables you to decouple application components so they run independently.
- Amazon SQS supports standard and FIFO queues.
- A producer sends a message to a queue. A consumer processes and deletes the message during the visibility timeout.
- Messages that cannot be processed can be sent to a dead letter queue.
- Long polling is a way to retrieve a large number of messages from your SQS queues.

Module 12: Building Decoupled Architectures

Section 4: Decoupling with Amazon SNS

aws academy

Introducing Section 4: Decoupling with Amazon SNS.



In modern cloud architecture, applications are decoupled into smaller, independent building blocks that are easier to develop, deploy, and maintain. Publish/subscribe (pub/sub) messaging provides instant event notifications for distributed applications.

The pub/sub model enables messages to be broadcast to different parts of a system asynchronously. A message *topic* provides a lightweight mechanism to broadcast asynchronous event notifications. It also provides endpoints that enable software components to connect to the topic so that they can send and receive those messages.

To broadcast a message, a component called a *publisher* pushes a message to the topic. Unlike message queues, which batch messages until they are retrieved, message topics transfer messages with no or little queuing, and push them out immediately to all *subscribers*. A subscriber will receive every message that is broadcast, unless it sets a message filtering policy. Examples of subscribers include web servers, email addresses, Amazon SQS queues, and AWS Lambda functions.

The subscribers to the message topic 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.

In the pub/sub messaging paradigm, notifications are delivered to clients by using a *push mechanism* that eliminates the need to periodically check or poll for new information and updates.

Amazon SNS

- Is a highly available, durable, secure, and fully managed pub/sub messaging service
- Uses a push mechanism
- Supports encrypted topics using customer master keys (CMKs)

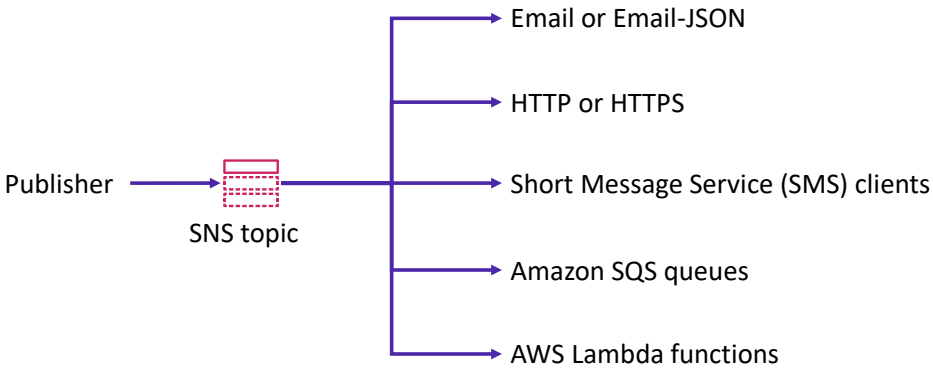
Amazon Simple Notification Service (Amazon SNS) is a web service that you can use to set up, operate, and send notifications from the cloud. The service follows the *pub/sub* messaging paradigm, where notifications are delivered to clients by using a push mechanism. Amazon SNS is designed to meet the needs of the largest and most demanding applications, and it enables applications to publish an unlimited number of messages at any time.

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.

Amazon SNS supports encrypted topics. After you publish messages to encrypted topics, Amazon SNS uses customer master keys (CMKs) to encrypt your messages. CMKs are the primary resources in AWS KMS. Amazon SNS supports both customer managed and AWS managed CMKs.

When Amazon SNS receives your messages, they are encrypted by using a 256-bit Advanced Encryption Standard-Galois/Counter Mode (AES-GCM) algorithm. The encrypted messages are stored redundantly across multiple servers and data centers, and across multiple Availability Zones for durability. Messages are decrypted just before they are delivered to subscribed endpoints. For more information about encrypting messages published to Amazon SNS, read this AWS Compute blog post.

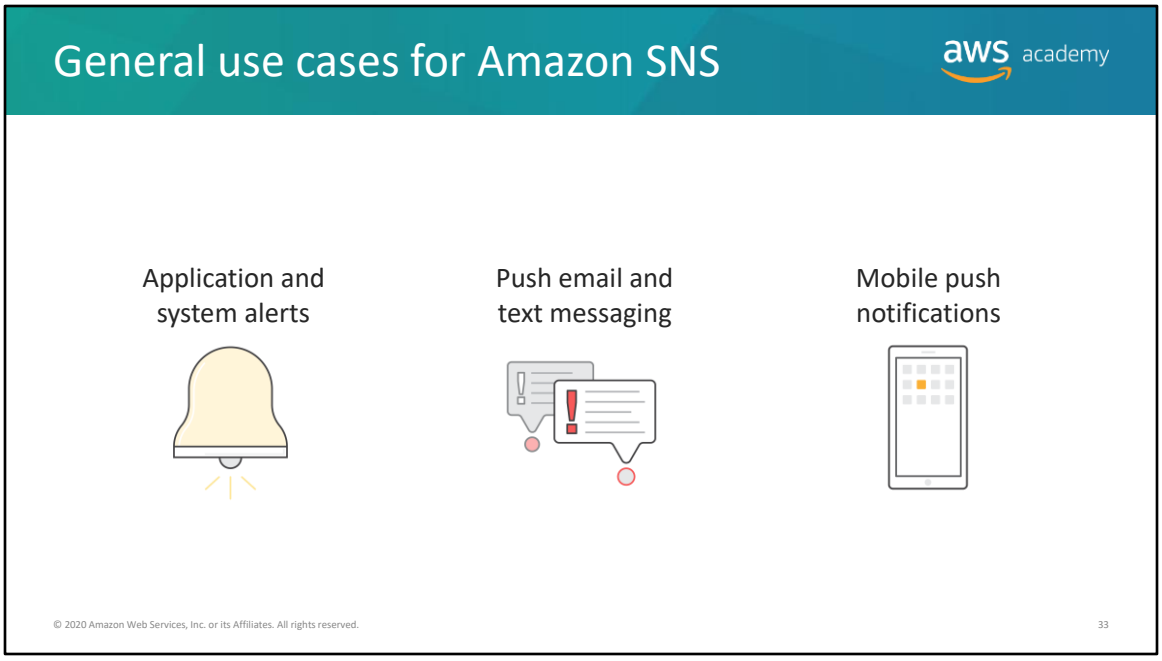
Supported transport protocols



Amazon SNS supports the following transport protocols for delivering messages:

- *Email or Email-JSON* – Messages are emailed to registered addresses. Email-JSON sends notifications as a JavaScript Object Notation (JSON) object, and Email sends text-based email message.
- *Hypertext Transfer Protocol (HTTP) or Secure HTTP (HTTPS)* – During subscription registration, subscribers specify a URL. Messages are delivered through an HTTP POST request to the specified URL.
- *Short Message Service (SMS)* – Messages are sent to registered phone numbers as SMS text messages.
- *Amazon SQS queues* – Users specify an SQS standard queue as the endpoint. Amazon SNS will enqueue a notification message to the specified queue. FIFO queues are not currently supported.
- *AWS Lambda functions* – Messages are delivered to AWS Lambda functions, which handle message customizations, enable message persistence, or communicate with other AWS services.

General use cases for Amazon SNS



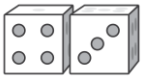
There are many ways to use Amazon SNS:

- *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, indicating that an update is available. The notification message can include a link to download and install the update.

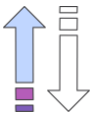
Amazon SNS considerations

Single published message

No recall options



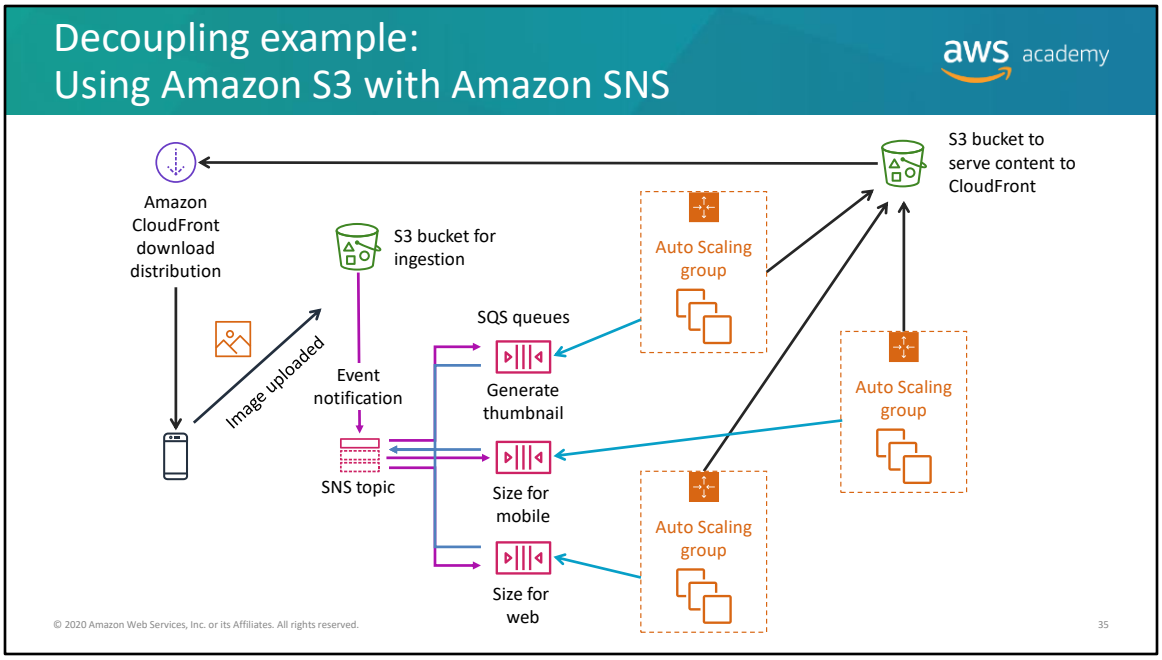
Order and delivery not guaranteed



Retry policy for each delivery protocol

If you are planning to use Amazon SNS, consider the following points:

- Each notification message contains a single published message.
- When a message is delivered successfully, there is no way to recall it.
- Amazon SNS will attempt to deliver messages from the publisher in the order that they were published into the topic. However, network issues could potentially result in out-of-order messages at the subscriber end.
- Amazon SNS defines a delivery policy for each delivery protocol. The delivery policy defines how Amazon SNS retries the delivery of messages when server-side errors occur (that is, when the system that hosts the subscribed endpoint becomes unavailable). If a message cannot be successfully delivered on the first attempt, Amazon SNS uses a four-phase retry policy:
    1. Retries with no delay in between attempts;
    2. Retries with minimum delay between attempts;
    3. Retries according to a back-off model;
    4. Retries with maximum delay between attempts. When the message delivery retry policy is exhausted, Amazon SNS can move the message to a DLQ.



With Amazon SNS, you can use topics to decouple message publishers from subscribers, fan-out messages to multiple recipients at one time, and eliminate polling in your applications.

You can use Amazon SNS to send messages in a single account or to resources in different accounts.

AWS services (such as Amazon EC2, Amazon S3, and Amazon CloudWatch) can publish messages to your SNS topics to trigger event-driven computing and workflows. In this example, after an image is uploaded to an S3 bucket, Amazon S3 triggers an event notification, which automatically sends the message to the SNS topic. Amazon SNS then delivers the S3 event notification to SQS queue subscribers. Auto Scaling groups of EC2 instances process the messages in the SQS queues and publish the processed images to an S3 bucket that serves the content to Amazon CloudFront.

Amazon SNS versus Amazon SQS

Feature	Amazon SNS (Publisher/Subscriber)	Amazon SQS (Producer/Consumer)
Producer/consumer	Publish/subscribe	Send/receive
Delivery mechanism	Push (passive)	Poll (active)
Distribution model	Many to many	One to one
Message persistence	No	Yes

- Amazon SNS uses a pub/sub messaging paradigm and enables applications to send time-critical messages to multiple subscribers through a push mechanism.
- Amazon SQS uses a send/receive messaging paradigm and exchanges messages through a polling model—sending and receiving components are decoupled.
- Amazon SQS provides flexibility for distributed components of applications—they can send and receive messages without requiring that each component must be concurrently available.

Section 4 key takeaways

- 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
- When you using Amazon SNS, you create a topic and set policies that restrict who can publish or subscribe to the topic
- You can use topics to decouple message publishers from subscribers, fan-out messages to multiple recipients at one time, and eliminate polling in your applications
- AWS services can publish messages to your SNS topics to trigger event-driven computing and workflows

Some key takeaways from this section of the module include:

- 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
- When you use Amazon SNS, you create a topic and set policies that restrict who can publish or subscribe to the topic
- You can use topics to decouple message publishers from subscribers, fan-out messages to multiple recipients at one time, and eliminate polling in your applications
- AWS services can publish messages to your SNS topics to trigger event-driven computing and workflows

Module 12: Building Decoupled Architectures

Section 5: Sending messages between cloud applications and on-premises with Amazon MQ

aws academy

Introducing Section 5: Sending messages between cloud applications and on-premises with Amazon MQ.

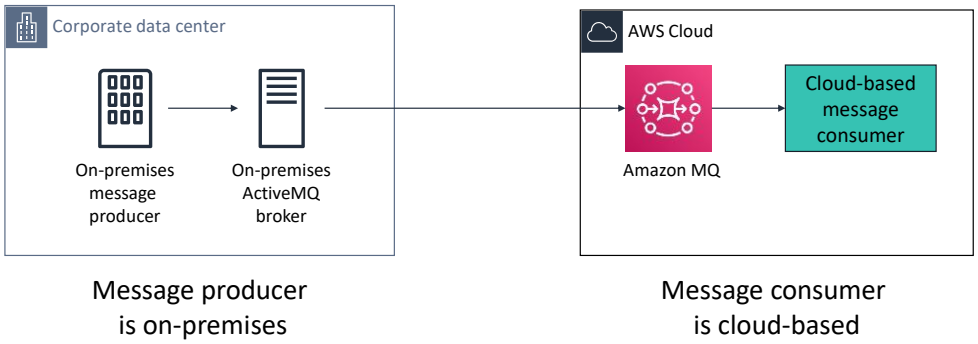
Amazon MQ

- Is a managed message broker service for Apache ActiveMQ
- Manages the provisioning, setup, and maintenance of ActiveMQ
- Simplifies message migration to the cloud
- Is compatible with open-standard APIs and protocols
  - JMS, NMS, AMQP, STOMP, MQTT, and WebSockets

Amazon MQ is a managed message broker service for Apache ActiveMQ that enables you to set up and operate message brokers in the cloud. Message brokers enable different software systems—which often use different programming languages on different platforms—to communicate and exchange information. Amazon MQ reduces your operational load by managing the provisioning, setup, and maintenance of ActiveMQ, a popular open-source message broker.

With Amazon MQ, you can migrate messaging to the cloud while preserving the existing connections between your applications. It supports open standard APIs and protocols for messaging, including Java Message Service (JMS), .NET Message Service (NMS), Advanced Message Queuing Protocol (AMQP), Streaming Text Oriented Messaging Protocol (STOMP), MQ Telemetry Transport (MQTT), and WebSockets. You can move 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.

Amazon MQ use case: Hybrid cloud environment



Many organizations, particularly enterprises, rely on message brokers to connect and coordinate different systems. Message brokers enable distributed applications to communicate with each other. They serve as the technological backbone for their IT environment and, ultimately, their business services. Applications depend on messaging to work.

In many cases, these organizations have started to build new cloud-native 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 enables organizations 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 AWS Lambda from queues and topics that are managed by Amazon MQ brokers to integrate legacy systems with serverless architectures.

The example shows that you can use Amazon MQ to integrate on-premises and cloud environments by using the network of brokers feature of ActiveMQ. The diagram shows the message lifecycle from the on-premises producer to the on-premises broker, which traverses the hybrid connection between the on-premises broker and Amazon MQ. Finally, the message moves to consumption within the AWS Cloud.

For more information about how to use Amazon MQ to integrate on-premises and cloud environments, read this AWS Compute blog post.

Amazon MQ versus Amazon SQS and Amazon SNS

Amazon MQ	Amazon SQS and SNS
For application migration	For born-in-the-cloud applications
Protocols: JMS, NMS, AMQP, STOMP, MQTT, and WebSockets	Protocol: HTTPS
Feature-rich	Nearly unlimited throughput
Pay per hour and pay per GB	Pay per request
Can do pub/sub	Cannot do pub/sub in Amazon SQS, but you can do pub/sub in Amazon SNS

Amazon MQ is a managed message broker service that provides compatibility with many popular message brokers. Amazon SQS and Amazon SNS are queue and topic services, respectively, that are highly scalable, simple to use, and don't require you to set up message brokers.

- If you use messaging with existing applications, and want to move your messaging to the cloud, AWS recommends using Amazon MQ. It supports open standard APIs and protocols. You can switch from any standards-based message broker to Amazon MQ without the need to rewrite the messaging code in your applications.
- If you are building new applications in the cloud, AWS recommends using Amazon SQS and Amazon SNS.

Section 5 key takeaways

42

aws academy

- Amazon MQ is a managed message-broker service for Apache ActiveMQ that enables you to set up and operate message brokers in the cloud
- Amazon MQ manages the provisioning, setup, and maintenance of ActiveMQ, which is a popular open-source message broker
- Amazon MQ is compatible with open standard APIs and protocols (that is, JMS, NMS, AMQP, STOMP, MQTT, and WebSockets)
- You can use Amazon MQ to integrate on-premises and cloud environments by using the network of brokers feature of ActiveMQ

Some key takeaways from this section of the module include:

- Amazon MQ is a managed message broker service for Apache ActiveMQ that enables you set up and operate message brokers in the cloud
- Amazon MQ manages the provisioning, setup, and maintenance of ActiveMQ, which is a popular open-source message broker
- Amazon MQ is compatible with open standard APIs and protocols (that is, JMS, NMS, AMQP, STOMP, MQTT, and WebSockets)
- You can use Amazon MQ to integrate on-premises and cloud environments by using the network of brokers feature of ActiveMQ

Module 12: Building Decoupled Architectures

Module wrap-up

aws academy

It’s now time to review the module and wrap up with a knowledge check and discussion of a practice certification exam question.

Module summary

In summary, in this module, you learned how to:

- 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

In summary, in this module, you learned how to:

- 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

Complete the knowledge check

It is now time to complete the knowledge check for this module.

Sample exam question

A company must perform asynchronous processing, and implemented 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 are kept to a minimum.

What should a Solutions Architect do to ensure that empty responses are reduced?

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 long polling wait time for the queue

Look at the answer choices and rule them out based on the keywords that were previously highlighted.

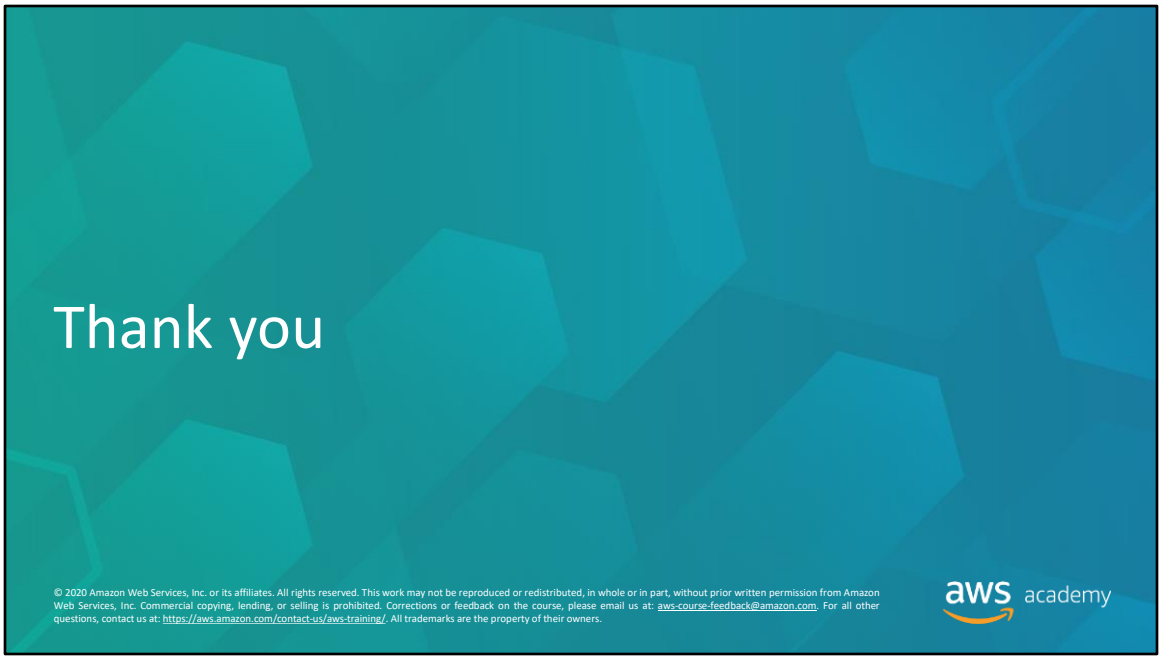
***The correct answer is D**: "Increase the long polling wait time for the queue."* When the ReceiveMessageWaitTimeSeconds property of a queue is set to a value greater than zero, long polling is in effect. Long polling reduces the number of empty responses by enabling Amazon SQS to wait until a message is available before it sends a response to a ReceiveMessage request.

Additional resources

- Building Loosely Coupled, Scalable, C# Applications with Amazon SQS and Amazon SNS
- Amazon SQS resources
- Amazon SNS resources
- Amazon MQ resources

If you want to learn more about the topics covered in this module, you might find the following additional resources helpful:

- Building Loosely Coupled, Scalable, C# Applications with Amazon SQS and Amazon SNS
- Amazon SQS resources
- Amazon SNS resources
- Amazon MQ resources



Thank you for completing this module.