SWINBURNE UNIVERSITY OF TECHNOLOGY

Cloud Computing Architecture

Lecture 9 Decoupled Architectures

includes material from
ACA Module 3.06 – Decoupling Your Infrastructure

Reminders

- Assignment 2 - was due 8 October
- Assignment 3

Last week

- HA – more on managing demand
  - More on Load Balancing
  - Elastic IP addresses
  - More on AutoScaling
  - Scaling Data Stores
  - AWS Lambda and Event-Driven Scaling
- Automating Infrastructure
  - Why automated Infrastructure?
  - CloudFormation
  - CloudFormation Template anatomy

Quizzes:
ACA Mod 4 Designing for High Availability
ACA Mod 5 Automating your Infrastructure

SWINBURNE UNIVERSITY OF TECHNOLOGY

This week – Decoupled Architectures

- ☐Loose coupling
- ☐Message Driven Architectures
  - ☐Message Queues
  - ☐MoM middleware, JMS, ActiveMQ
  - ☐AWS SQS (Simple Message Queue)
- ☐Event Driven Architectures
  - ☐Events and Complex Event Processing
  - ☐Publish Subscribe (pub-sub) Systems
  - ☐AWS SNS (Simple Notification Service)

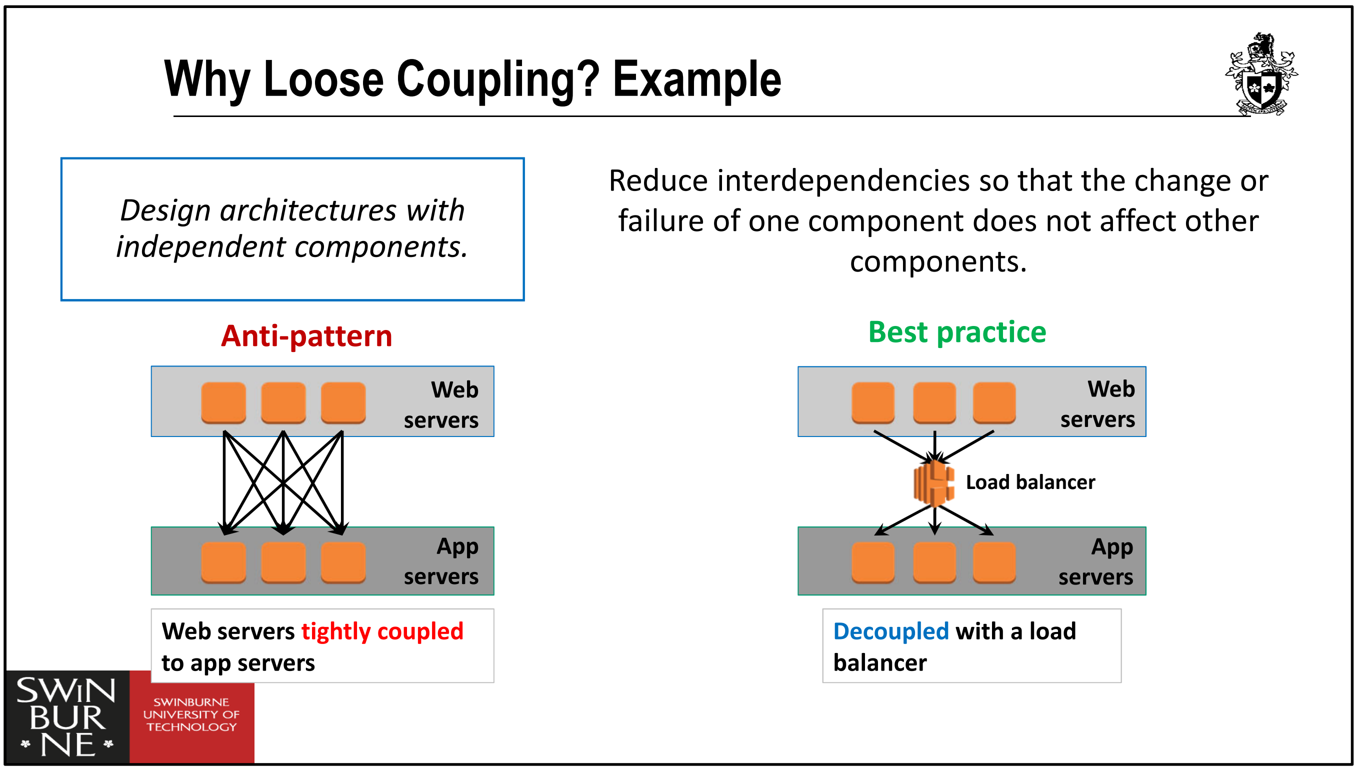
Quizzes:
ACA Mod 6 Decoupling you Infrastructure

SWINBURNE UNIVERSITY OF TECHNOLOGY

This week – Decoupled Architectures

- ☐**Loose coupling**
- ☐Message Driven Architectures
  - ☐Message Queues
  - ☐MoM middleware, JMS, ActiveMQ
  - ☐AWS SQS (Simple Message Queue)
- ☐Event Driven Architectures
  - ☐Events and Complex Event Processing
  - ☐Publish Subscribe (pub-sub) Systems
  - ☐AWS SNS (Simple Notification Service)

SWINBURNE UNIVERSITY OF TECHNOLOGY



Why Loose Coupling? Example

Design architectures with independent components.

Reduce interdependencies so that the change or failure of one component does not affect other components.

Anti-pattern
Web servers
App servers
Best practice
Web servers
Load balancer
App servers

Web servers tightly coupled to app servers

Decoupled with a load balancer

Traditional infrastructures revolve around chains of tightly integrated servers, each with a specific purpose. When one component or layer goes down, the disruption to the system can be fatal. Additionally, traditional infrastructures impede scaling; if you add or remove servers at one layer, every server on every connecting layer must also be connected appropriately.

A best practice is to make sure components are loosely coupled. Loose coupling refers to designing architectures with interdependent components. This reduces interdependencies, so a change or failure of one component does not affect other components. With loose coupling, you leverage managed solutions as intermediaries between layers of your system. This way, failures and scaling of a component or a layer are automatically handled by the intermediary.

Two primary solutions for decoupling components are load balancers and message queues.

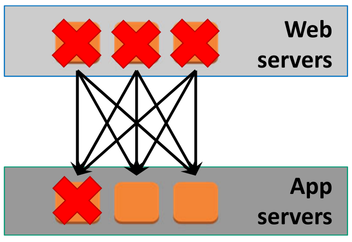
In the diagram, the solution on the left illustrates a collection of tightly coupled web and app servers. If one of the app servers goes down, each connection attempt between a web server and the unhealthy app server will cause an access error.

The solution on the left can be decoupled using load balancing. The solution on the right shows a load balancer—in this case, an Elastic Load Balancing instance—that routes requests between the web servers and the app servers. If one of the app servers goes down, the load balancer automatically directs all traffic to the healthy servers, and the infrastructure is not impacted.

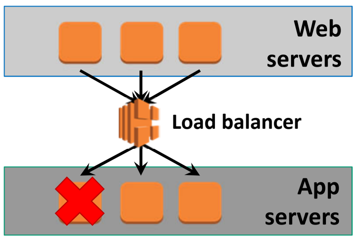
Why Loose Coupling? Example

The more loosely your system is coupled, the more easily it scales and the more fault-tolerant it can be.

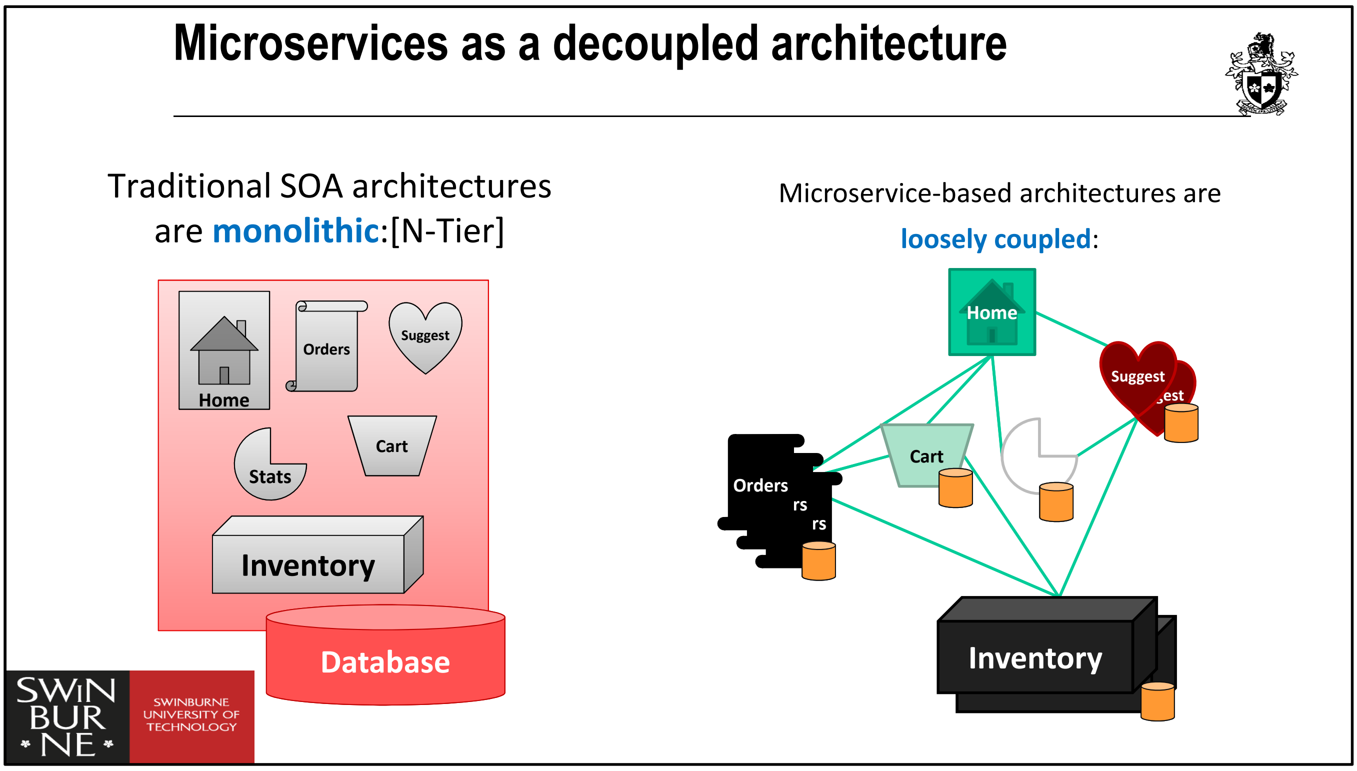
Tightly coupled



Loosely coupled



Loose coupling also enables your environment to be more fault tolerant, which is a key to high availability. With a loosely coupled distributed system, the failure of one component can be managed between tiers, so the fault does not spread beyond a single instance.



Microservices as a decoupled architecture

Traditional SOA architectures are **monolithic**:[N-Tier]

Microservice-based architectures are **loosely coupled**:

So, how does a traditional architecture compare to a microservices-based architecture?

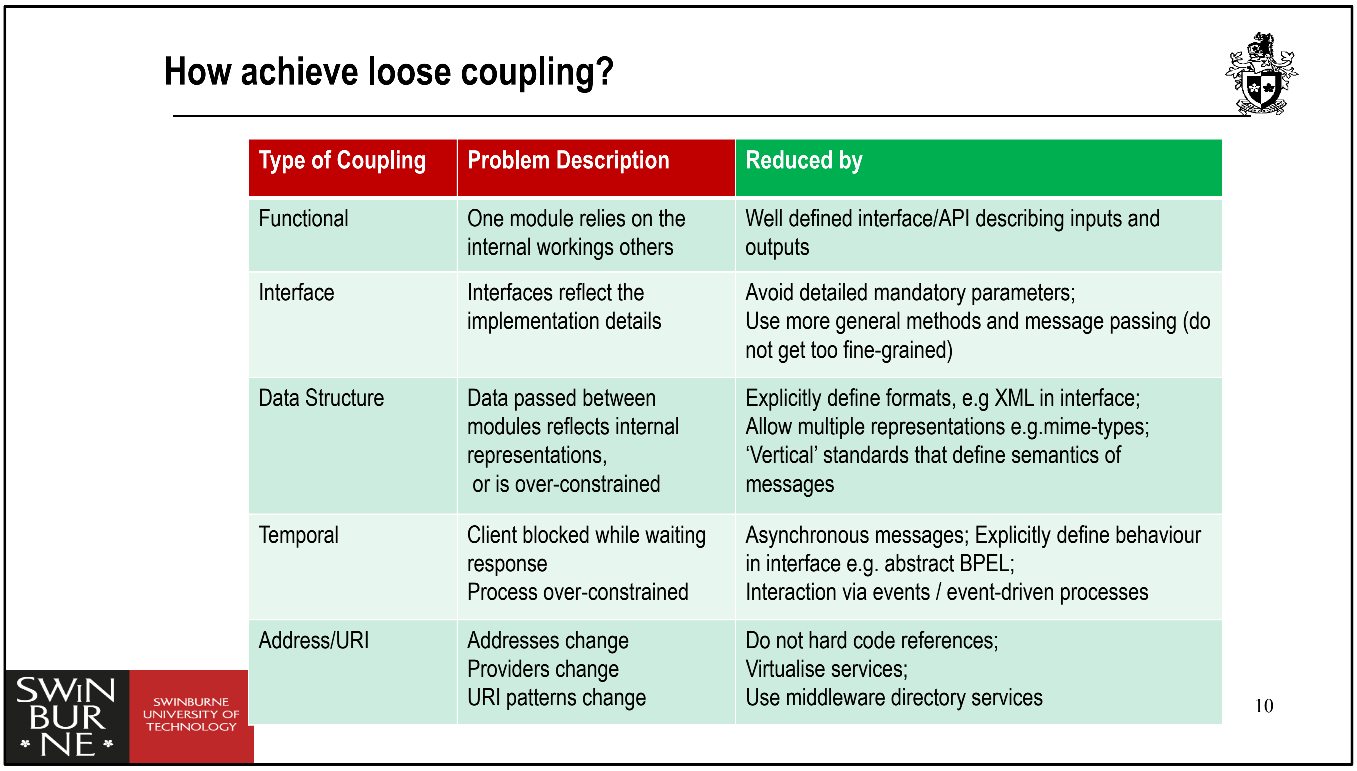
Traditional application architectures are monolithic—all components of the system are tightly coupled together. For example, the architecture on the left illustrates a traditional monolithic architecture, with a database containing inventory, statistics, orders, shopping cart contents, and customer suggestions.

In contrast, microservices-based architectures treat components as independent services connected together in a loosely coupled manner, as illustrated in the architecture shown on the right. This architecture implements more than one database for orders, uses multiple tables, and includes multiple backups of the inventory database. The cart, statistics, and suggestions are separate, so they can be easily scaled. With this microservices structure, if any part of the site goes down, the site can survive and continue to serve customers.

Loose Coupling

- A well architected system has *Loose Coupling*
- Tightly coupled systems tend to be hard to modify/ maintain. They exhibit the following disadvantages:
  - A change in one module usually forces a ripple effect of changes in other modules.
  - Assembly of modules might require more effort and/or time due to the increased inter-module dependency.
  - A particular module might be harder to reuse and/or test because dependent modules must be included.
- Software engineering can be viewed as successive attempts to create loosely coupled systems of encapsulated entities at different level of abstraction:
  - Code - Late-binding, dependency injection
  - Stateless interactions – e.g. Web
  - Service-oriented computing, micro-services, API registries, etc
  - Architectural interaction style: RPC → Message driven → Event driven

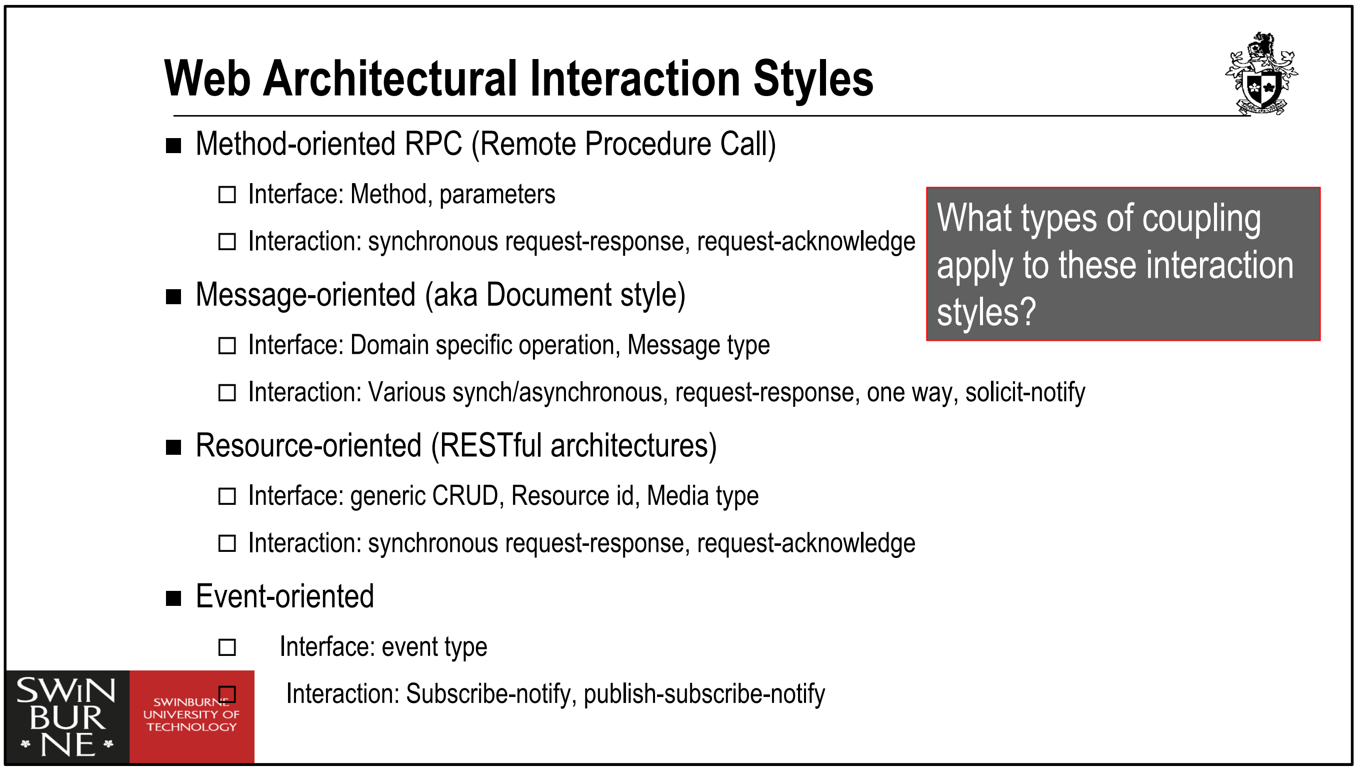
SWINBURNE UNIVERSITY OF TECHNOLOGY



How achieve loose coupling?

Type of Coupling	Problem Description	Reduced by
Functional	One module relies on the internal workings others	Well defined interface/API describing inputs and outputs
Interface	Interfaces reflect the implementation details	Avoid detailed mandatory parameters; Use more general methods and message passing (do not get too fine-grained)
Data Structure	Data passed between modules reflects internal representations, or is over-constrained	Explicitly define formats, e.g XML in interface; Allow multiple representations e.g.mime-types; 'Vertical' standards that define semantics of messages
Temporal	Client blocked while waiting response Process over-constrained	Asynchronous messages; Explicitly define behaviour in interface e.g. abstract BPEL; Interaction via events / event-driven processes
Address/URI	Addresses change Providers change URI patterns change	Do not hard code references; Virtualise services; Use middleware directory services

10



Web Architectural Interaction Styles

- Method-oriented RPC (Remote Procedure Call)
  - Interface: Method, parameters
  - Interaction: synchronous request-response, request-acknowledge
- Message-oriented (aka Document style)
  - Interface: Domain specific operation, Message type
  - Interaction: Various synch/asynchronous, request-response, one way, solicit-notify
- Resource-oriented (RESTful architectures)
  - Interface: generic CRUD, Resource id, Media type
  - Interaction: synchronous request-response, request-acknowledge
- Event-oriented
  - Interface: event type
  - Interaction: Subscribe-notify, publish-subscribe-notify

What types of coupling apply to these interaction styles?

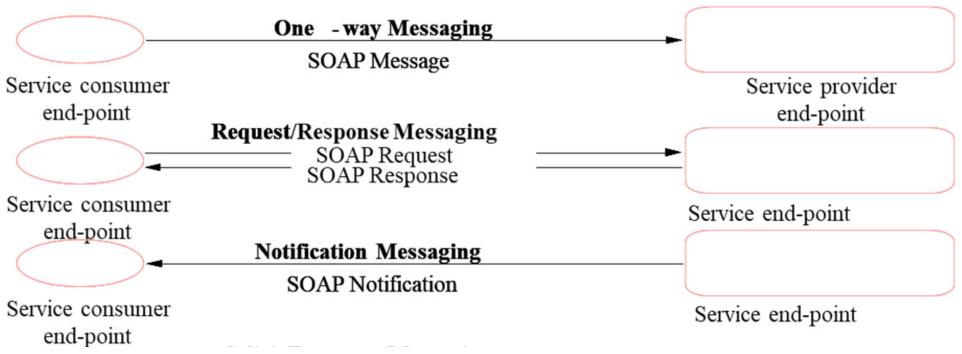
SWINBURNE UNIVERSITY OF TECHNOLOGY

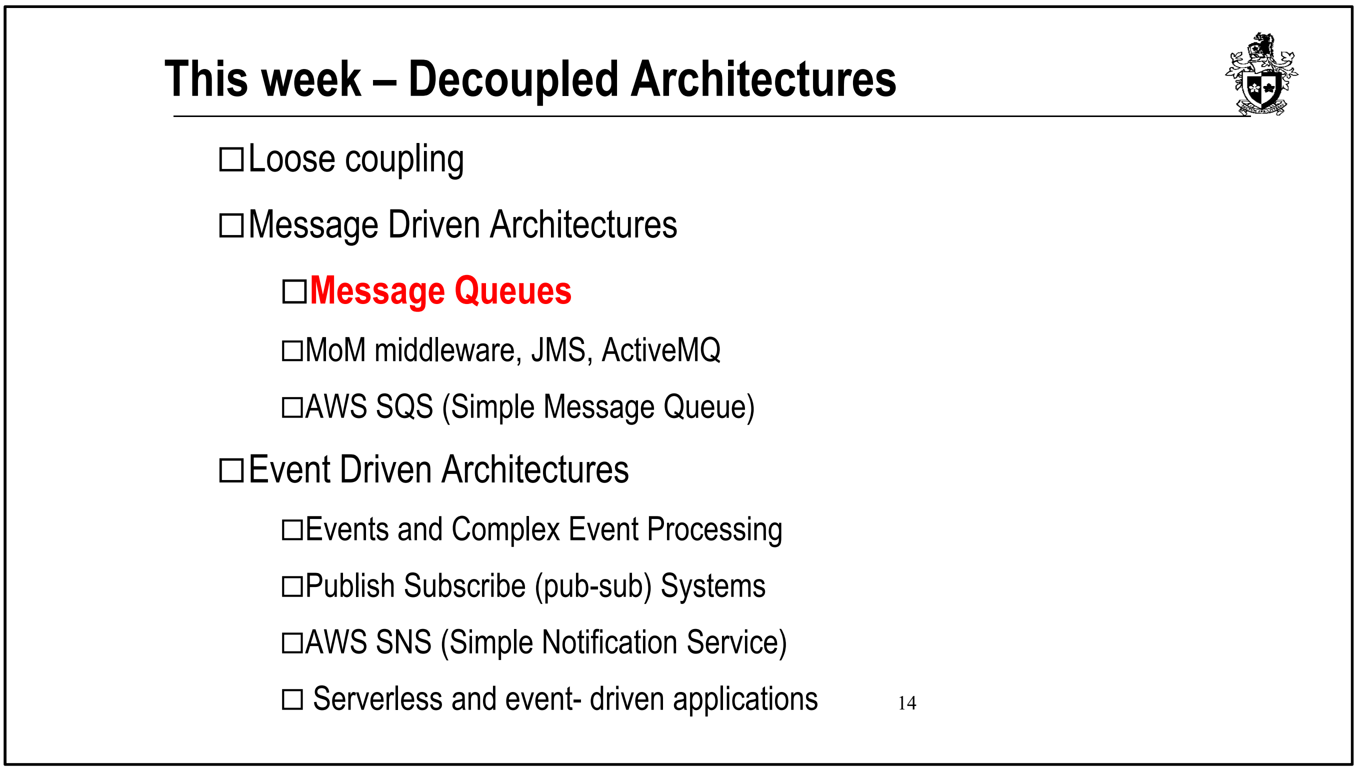
This week – Decoupled Architectures

- ☐Loose coupling
- ☐**Message Driven Architectures**
  - ☐Message Queues
  - ☐MoM middleware, JMS, ActiveMQ
  - ☐AWS SQS (Simple Message Queue)
- ☐Event Driven Architectures
  - ☐Events and Complex Event Processing
  - ☐Publish Subscribe (pub-sub) Systems
  - ☐AWS SNS (Simple Notification Service)
  - ☐ Serverless and event- driven applications

Message Exchange Patterns

- Different types of message exchange patterns

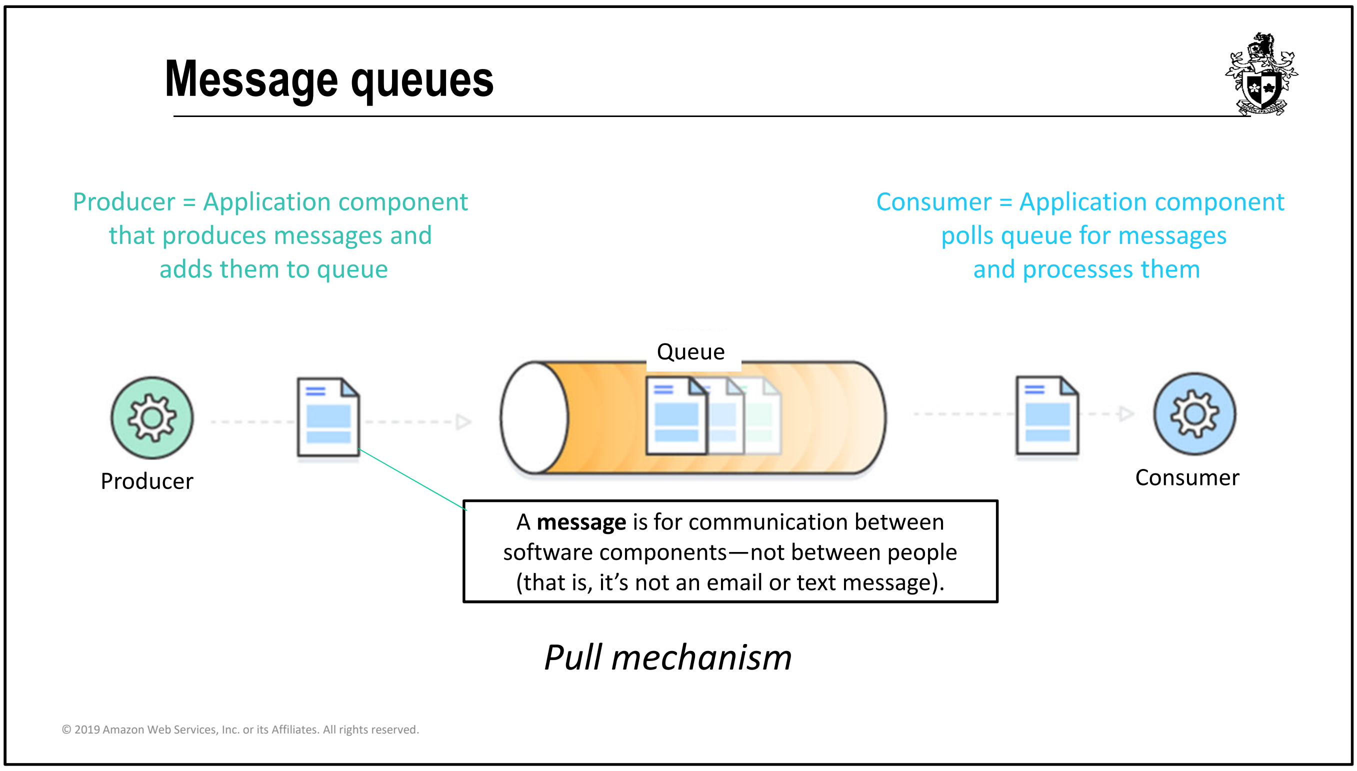

One - way Messaging
SOAP Message
Service consumer end-point
Service provider end-point
Request/Response Messaging
SOAP Request
SOAP Response
Service consumer end-point
Service end-point
Notification Messaging
SOAP Notification
Service consumer end-point
Service end-point



This week – Decoupled Architectures

- ☐Loose coupling
- ☐Message Driven Architectures
  - ☐**Message Queues**
  - ☐MoM middleware, JMS, ActiveMQ
  - ☐AWS SQS (Simple Message Queue)
- ☐Event Driven Architectures
  - ☐Events and Complex Event Processing
  - ☐Publish Subscribe (pub-sub) Systems
  - ☐AWS SNS (Simple Notification Service)
  - ☐ Serverless and event- driven applications

14



Message queues

Producer = Application component that produces messages and adds them to queue

Consumer = Application component polls queue for messages and processes them

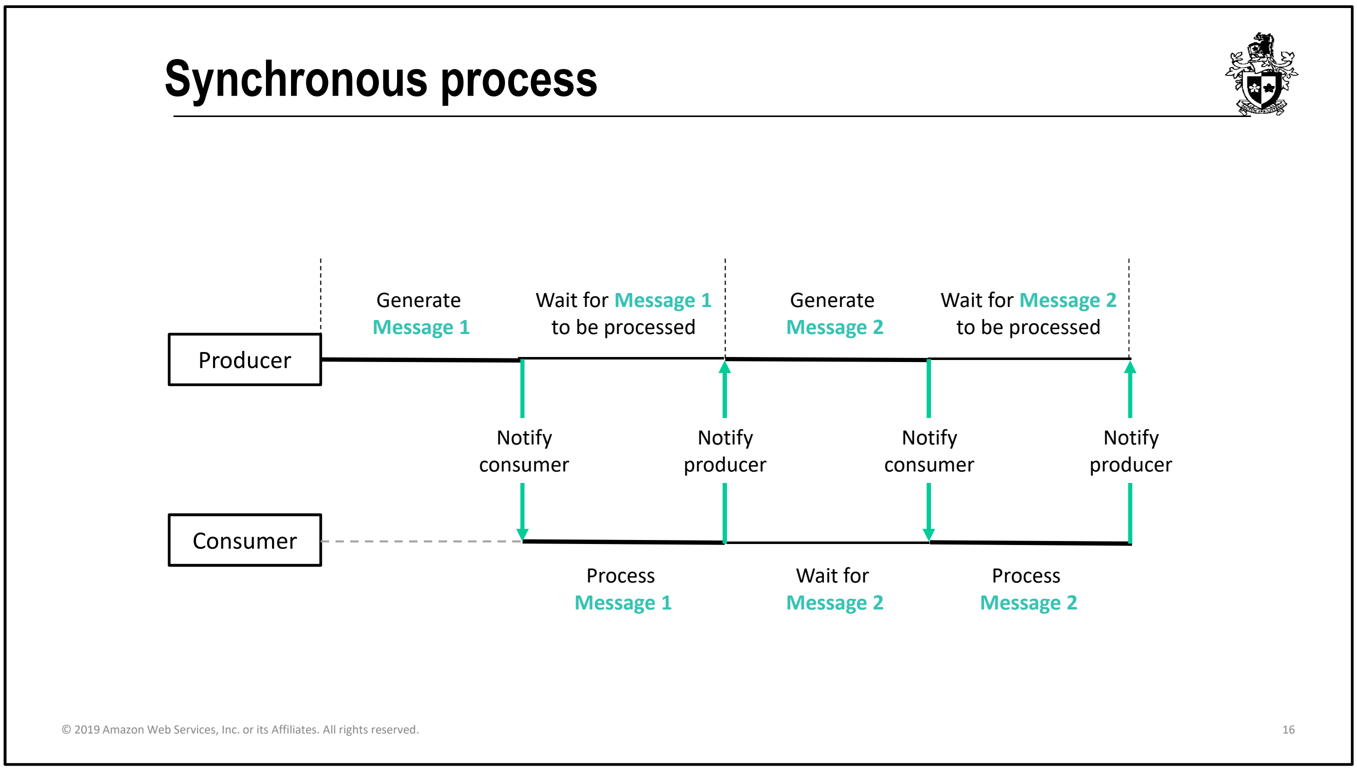
A **message** is for communication between software components—not between people (that is, it's not an email or text message).

Pull mechanism

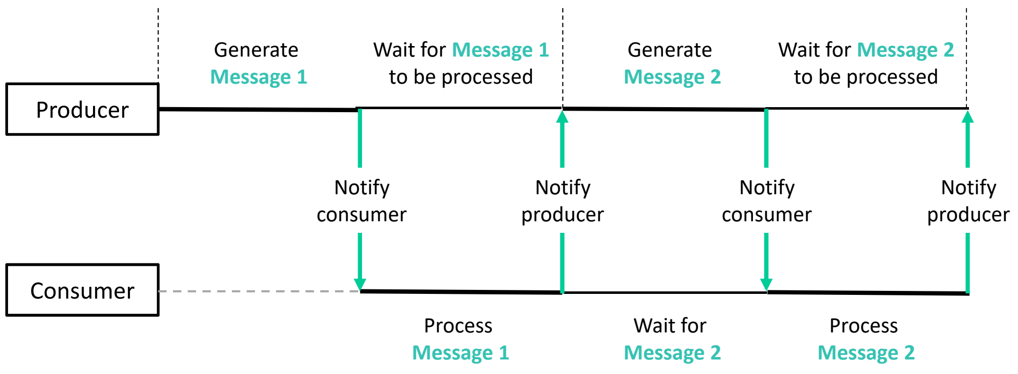
In modern cloud architecture, applications are decoupled into smaller, independent building blocks that are easier to develop, deploy, and maintain. Message queues provide communication and coordination for these distributed applications.

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, etc.

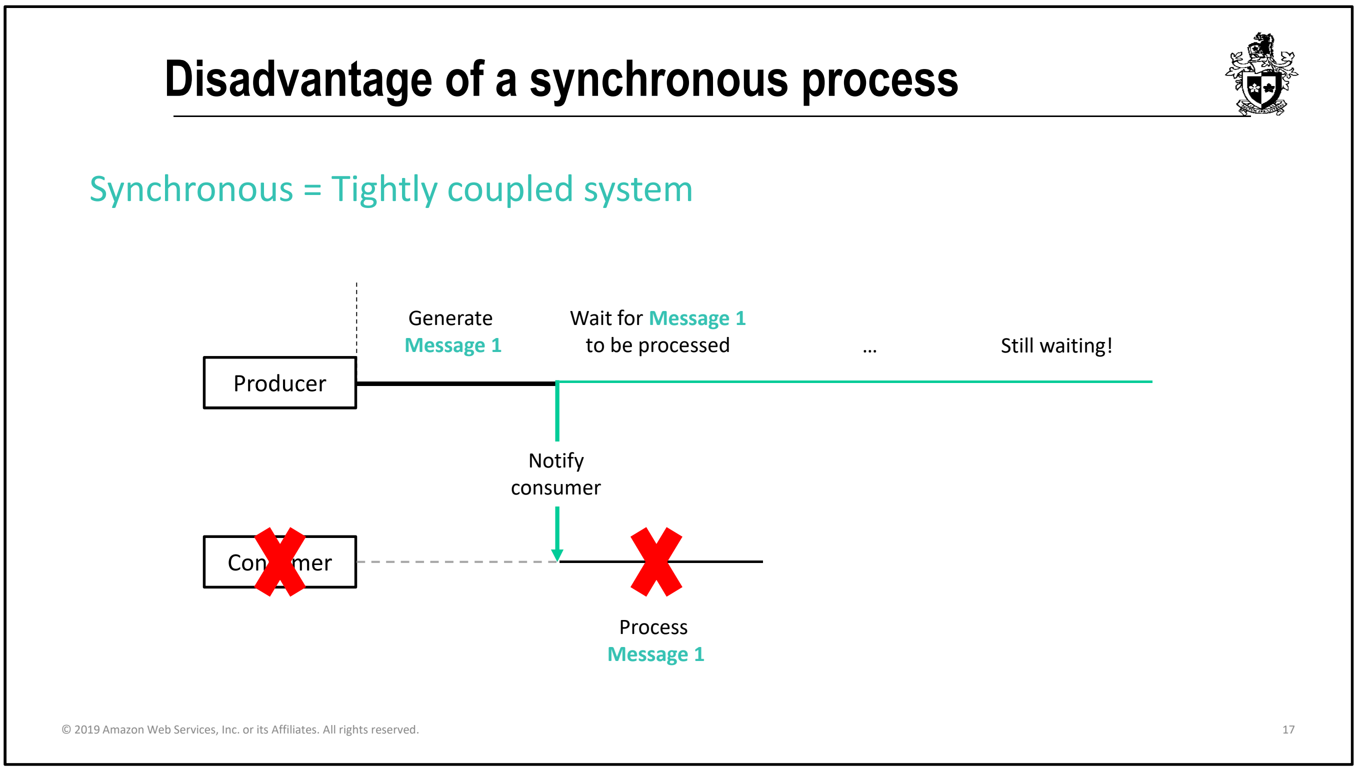
To send a message, a component that is called a *producer* adds a message to the queue. The message is stored in the queue until another component that is called a *consumer* retrieves and processes it.



Synchronous process

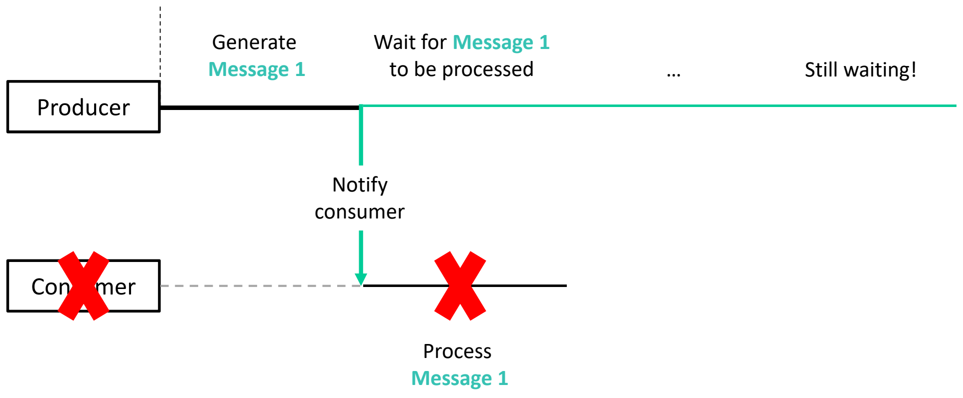


Message queues enable you to create an asynchronous process. To understand what this means, first consider how a synchronous process works. Say that you have a producer and a consumer. The producer generates a message, notifies the consumer, and waits for the message to be processed. The consumer processes the message and notifies the consumer when it's done. The consumer then waits for the next message. As you can see, this is a very dependent process, where there is strong interdependency between the producer and the consumer. This interdependency creates a *tightly coupled system*.

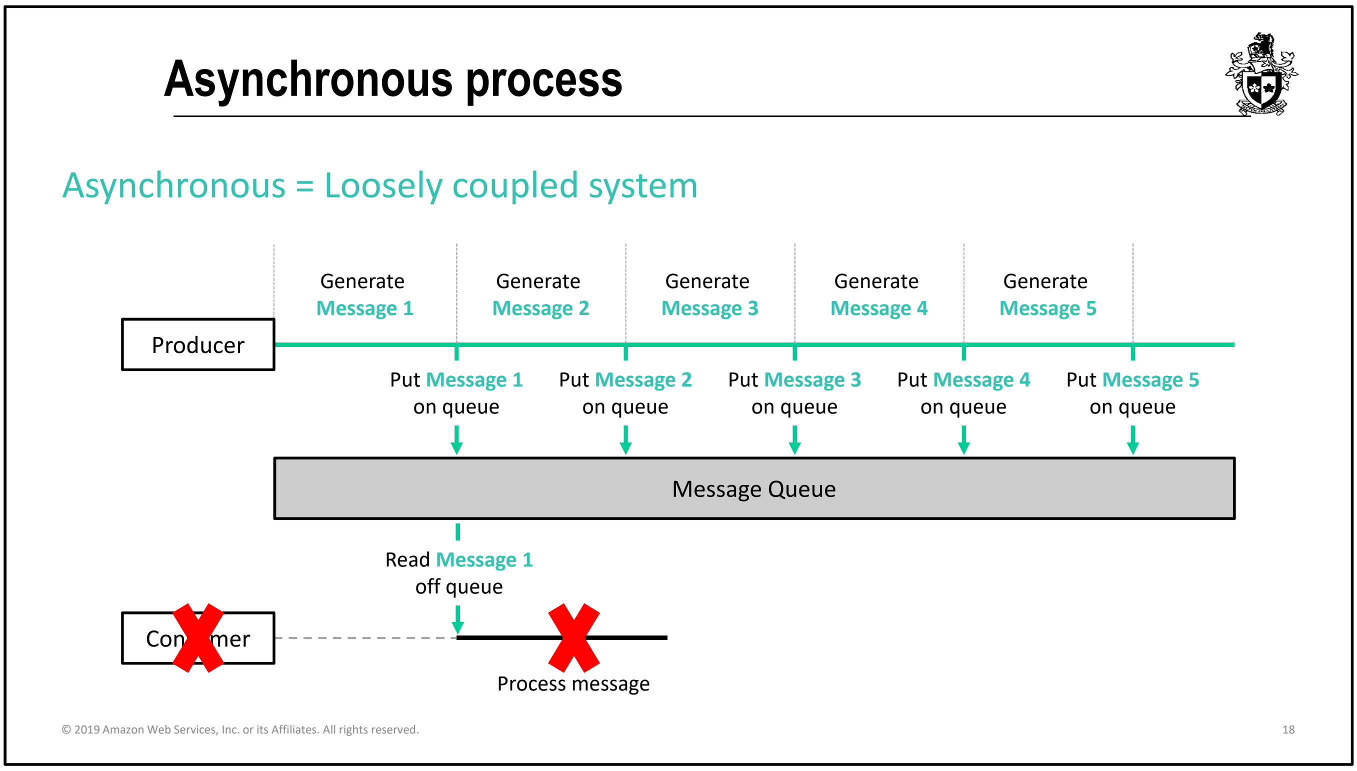


Disadvantage of a synchronous process

Synchronous = Tightly coupled system

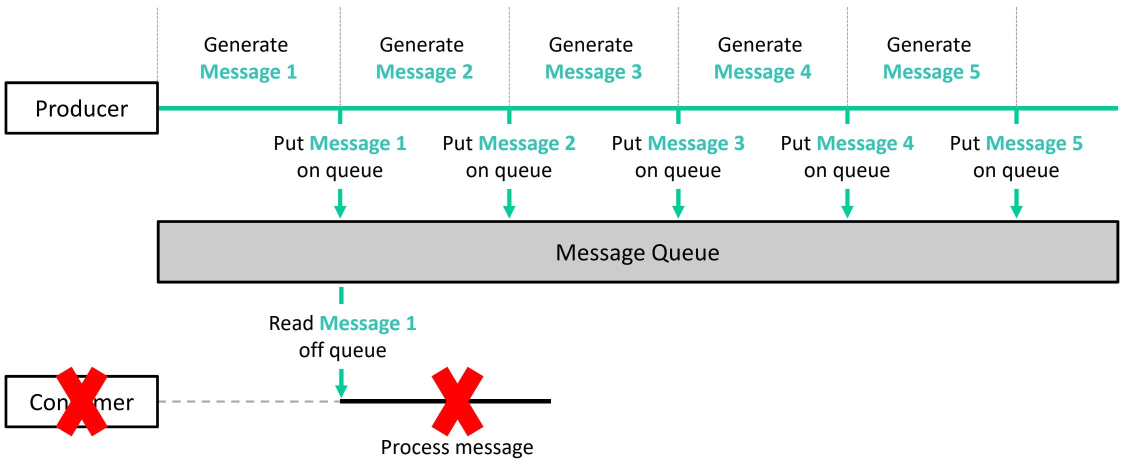


The disadvantage of a tightly coupled system is that *it is not fault-tolerant*. This means that if any component of the system fails, then the entire system will fail. If the consumer fails while processing a message, then the producer will be forced to wait until that message gets processed (assuming that the message is not lost). Additionally, if new consumer instances are launched to recover from a failure or to keep up with an increased workload, the producer must be explicitly made aware of the new consumer instances. In this scenario, the producer is tightly coupled with the consumer, and the coupling is prone to brittleness.



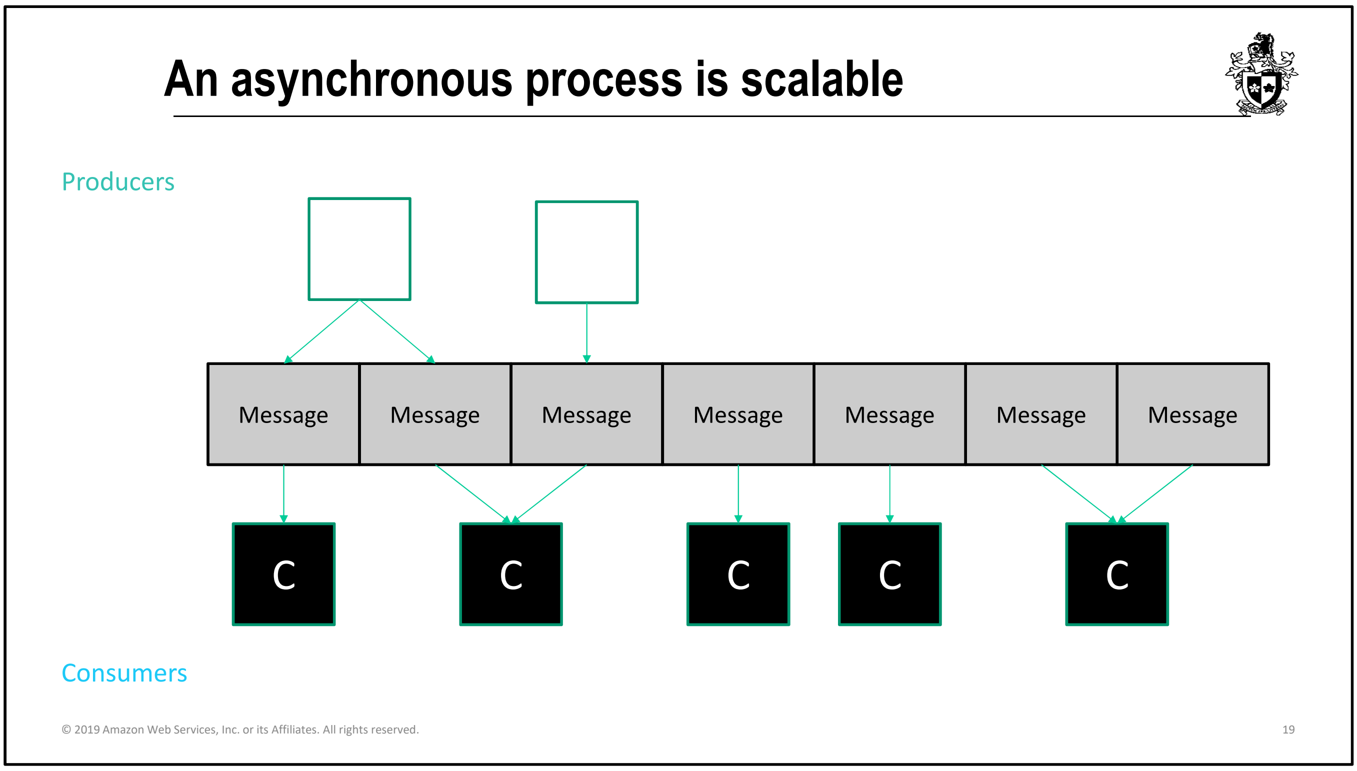
Asynchronous process

Asynchronous = Loosely coupled system



A message queue is a component of an asynchronous process. In an asynchronous system, the producer generates a message and puts in on the queue. The producer then generates another message and puts it on the queue. The producer does not have to wait for the first message to be read and processed before it puts the second message on the queue. Instead, the producer can continue to put messages on the queue whether they are being read by the consumer or not.

When the consumer reads a message off the queue and starts to process it, the producer is not impacted if this consumer fails. The producer can continue to put messages on the queue. There is no interdependency between the producer and the consumer, which makes for a more loosely coupled system. This solution has far greater fault tolerance.



An asynchronous process is scalable

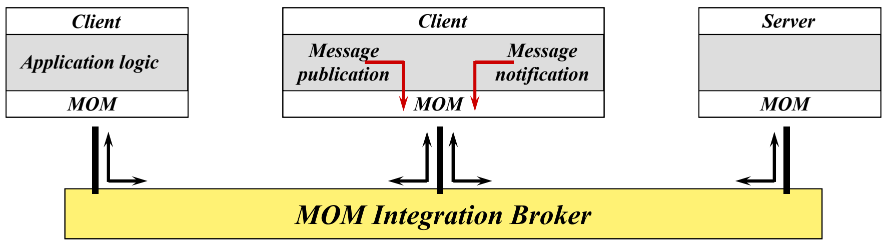
An asynchronous solution is also more scalable. You can add and remove producers and consumers as your application requires. A queue can support multiple producers and consumers. A single queue can be used simultaneously by many distributed application components, with no need for those components to coordinate with each other to share the queue.

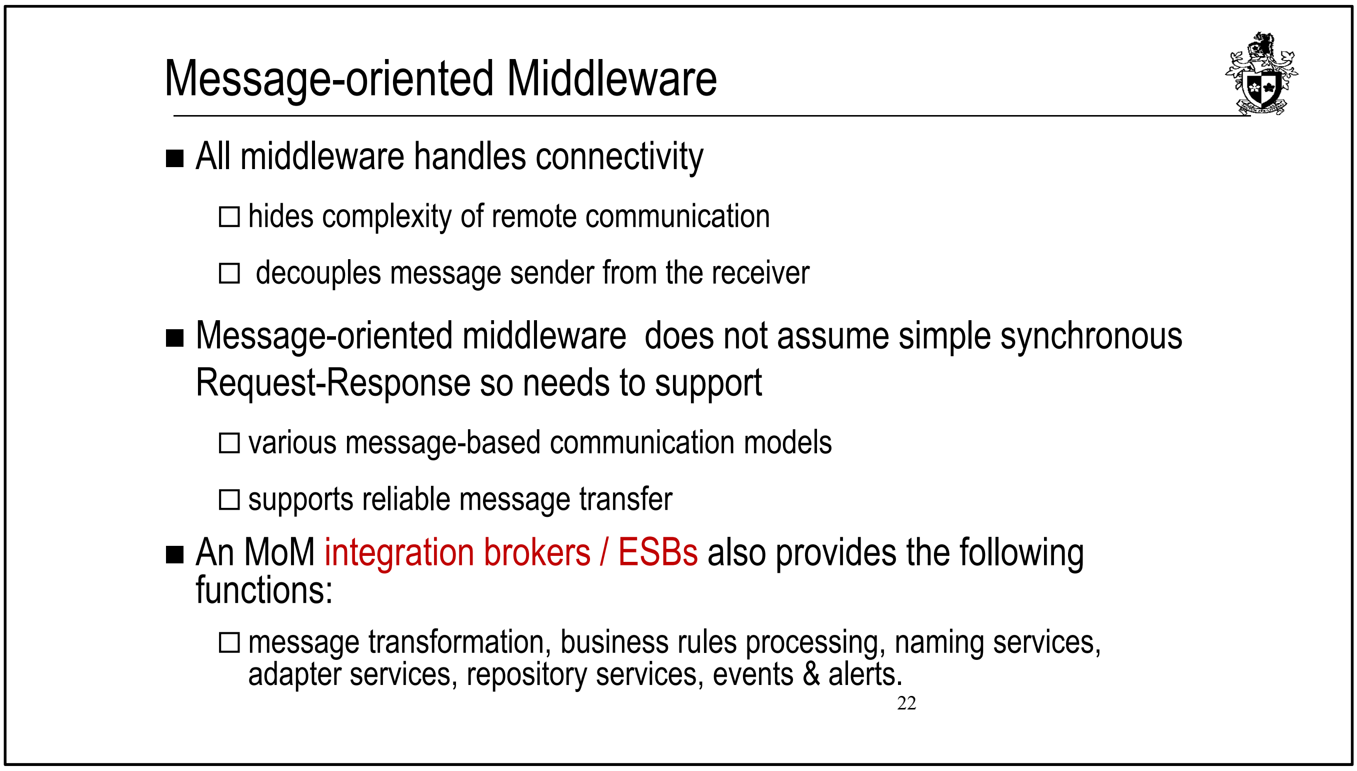
This week – Scalable and Decoupled Architectures

- Automated Scaling and Monitoring
- Decoupled Architectures
  - Types of decoupling
  - Message Driven Architectures
    - Message Queues
    - **MoM middleware, JMS, ActiveMQ**
    - AWS SQS (Simple Message Queue)
  - Event Driven Architectures
    - Events and Complex Event Processing
    - Publish Subscribe (pub-sub) Systems
    - AWS SNS (Simple Notification Service)

Message-oriented Middleware

- MOM is an infrastructure that involves the passing of data between applications using a common communication channel that carries self-contained messages.
- Messages are sent and received asynchronously.
- The messaging system (integration broker) is responsible for managing the connection points between clients & for managing multiple channels of communication between the connection points.





Message-oriented Middleware

- All middleware handles connectivity
  - ☐ hides complexity of remote communication
  - ☐ decouples message sender from the receiver
- Message-oriented middleware does not assume simple synchronous Request-Response so needs to support
  - ☐ various message-based communication models
  - ☐ supports reliable message transfer
- An MoM integration brokers / ESBs also provides the following functions:
  - ☐ message transformation, business rules processing, naming services, adapter services, repository services, events & alerts.

22

JMS

- The Java Message Service (JMS) API is a MOM API for sending messages between two or more clients.
- Allows application components based on Java EE to create, send, receive, and read messages.
  - loosely coupled
  - asynchronous
  - reliable

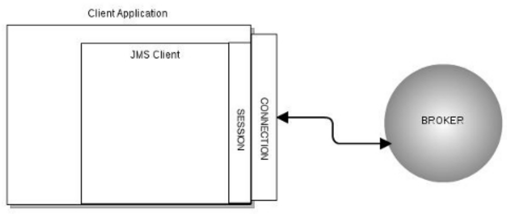
JMS interfaces are also provided on many non-Java MOMs.

- Version history
  - JMS 1.0.2b (June 25, 2001)
  - JMS 1.1 (March 18, 2002)
  - JMS 2.0 (May 21, 2013)

Ref: http://en.wikipedia.org/wiki/Java_Message_Service

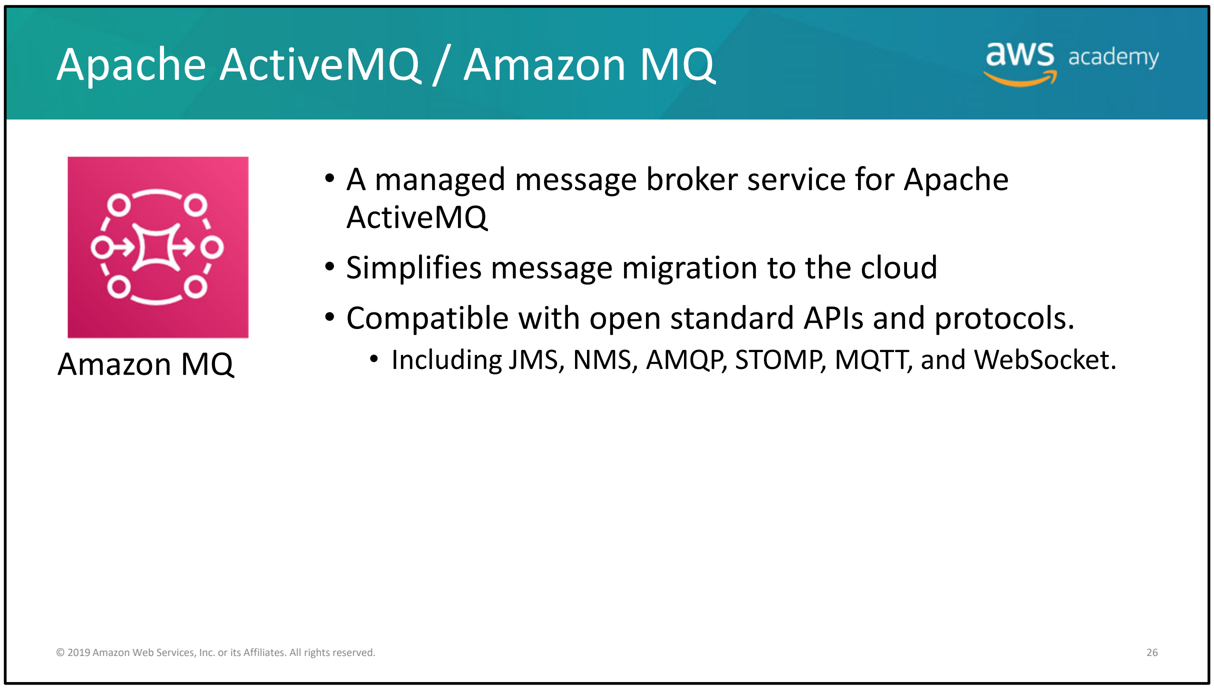
JMS Elements

- JMS provider / broker
  - An implementation of the JMS interface for a Message Oriented Middleware (MOM).
- JMS client
  - An application or process that produces and/or receives messages.
- JMS producer/publisher
  - A JMS client that creates and sends message.
- JMS consumer/subscriber
  - A JMS client that receives message.



MOM providers

- JMS (Java Messaging Service) defines a widely accepted API for MOM implementations
- Open Source
  - Apache Active MQ
  - JBoss Messaging
  - OpenJMS
  - WSO2 Message Broker
- Proprietory
  - BEA WebLogic
  - Oracle AQ
  - WebSphere MQ
  - Sun Java System Message Queue

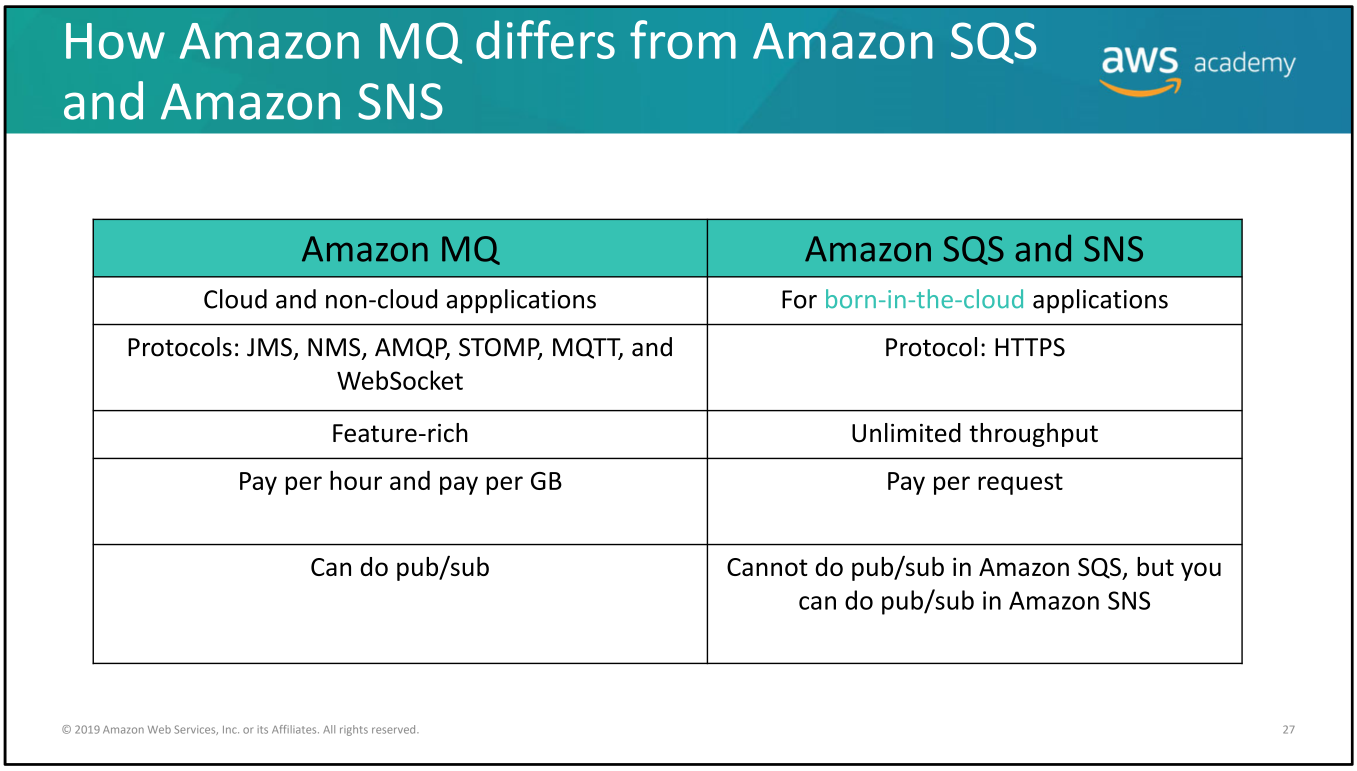


Apache ActiveMQ / Amazon MQ

Amazon MQ

- A managed message broker service for Apache ActiveMQ
- Simplifies message migration to the cloud
- Compatible with open standard APIs and protocols.
  - Including JMS, NMS, AMQP, STOMP, MQTT, and WebSocket.

Amazon MQ is a managed message broker service for Apache ActiveMQ that makes it easy to set up and operate message brokers in the cloud. Message brokers allow different software systems—which often use different programming languages and are on different platforms—to communicate and exchange information. Amazon MQ makes it easy to 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 WebSocket. You can move from any message broker that uses these standards to Amazon MQ, usually without needing to rewrite any messaging code. In most cases, you can simply update the endpoints of your applications to connect to Amazon MQ and start sending messages.



How Amazon MQ differs from Amazon SQS and Amazon SNS

Amazon MQ	Amazon SQS and SNS
Cloud and non-cloud appplications	For born-in-the-cloud applications
Protocols: JMS, NMS, AMQP, STOMP, MQTT, and WebSocket	Protocol: HTTPS
Feature-rich	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

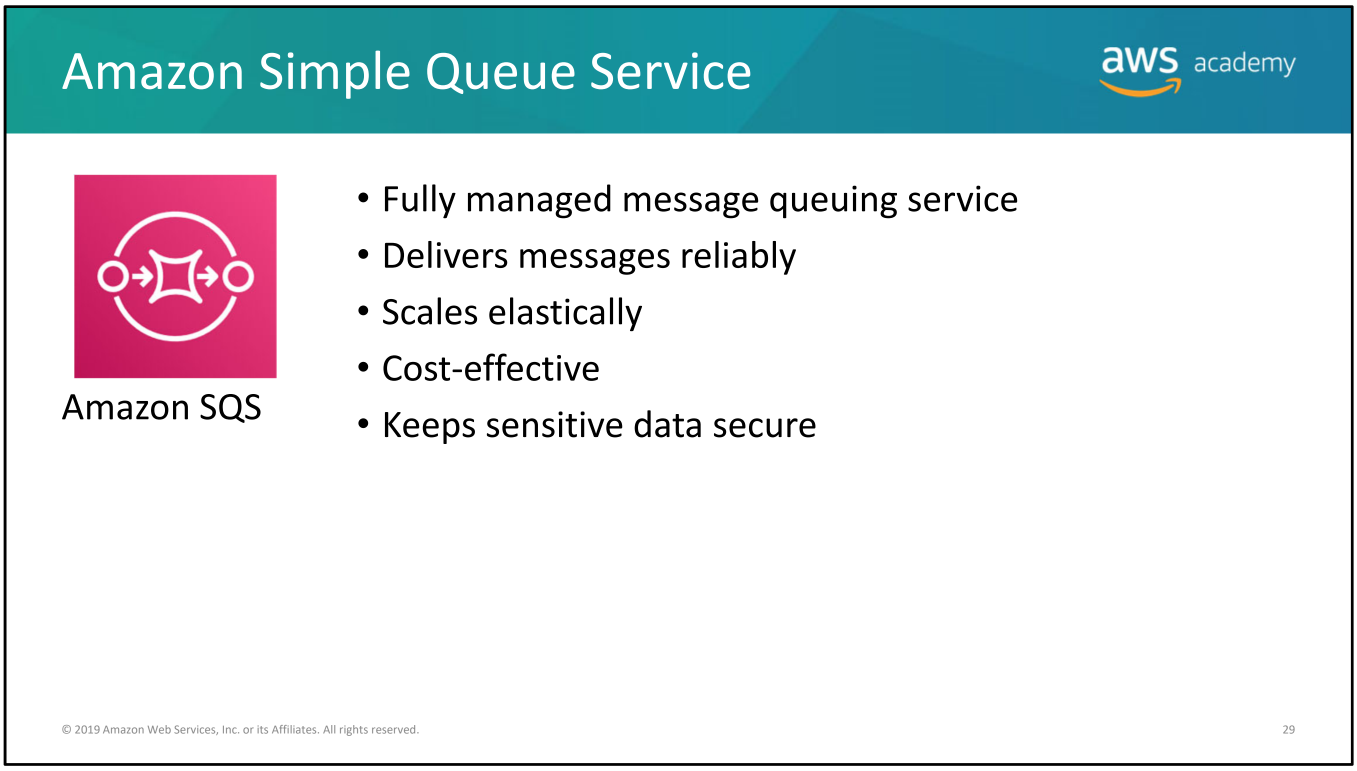
Amazon MQ is a managed message broker service that provides compatibility with many popular message brokers. Amazon SQS and Amazon SNS are queue and topic services, respectively, that are highly scalable, simple to use, and don't require you to set up message brokers.

- If you use messaging with existing applications, and want to move your messaging to the cloud quickly and easily, we recommend using Amazon MQ. It supports open standard APIs and protocols so you can switch from any standards-based message broker to Amazon MQ without rewriting the messaging code in your applications.
- If you are building new applications in the cloud, we recommend using Amazon SQS and Amazon SNS. Amazon SQS and Amazon SNS are lightweight, fully managed message queue and topic services, respectively, that scale almost infinitely and provide simple, easy-to-use APIs. You can use Amazon SQS and Amazon SNS to improve reliability and to decouple and scale microservices, distributed systems, and serverless applications.

This week – Decoupled Architectures

- ☐Loose coupling
- ☐Message Driven Architectures
  - ☐Message Queues
  - ☐MoM middleware, JMS, ActiveMQ
  - ☐**AWS SQS (Simple Message Queue)**
- ☐Event Driven Architectures
  - ☐Events and Complex Event Processing
  - ☐Publish Subscribe (pub-sub) Systems
  - ☐AWS SNS (Simple Notification Service)
  - ☐ Serverless and event- driven applications

SWINBURNE UNIVERSITY OF TECHNOLOGY



Amazon Simple Queue Service

Amazon SQS

- Fully managed message queuing service
- Delivers messages reliably
- Scales elastically
- Cost-effective
- Keeps sensitive data secure

Amazon Simple Queue Service (Amazon SQS) is a fully managed message queuing service that enables you to decouple and scale microservices, distributed systems, and serverless applications. You can use Amazon SQS to send, store, and receive messages between software components at any volume, without losing messages or requiring other services to be available.

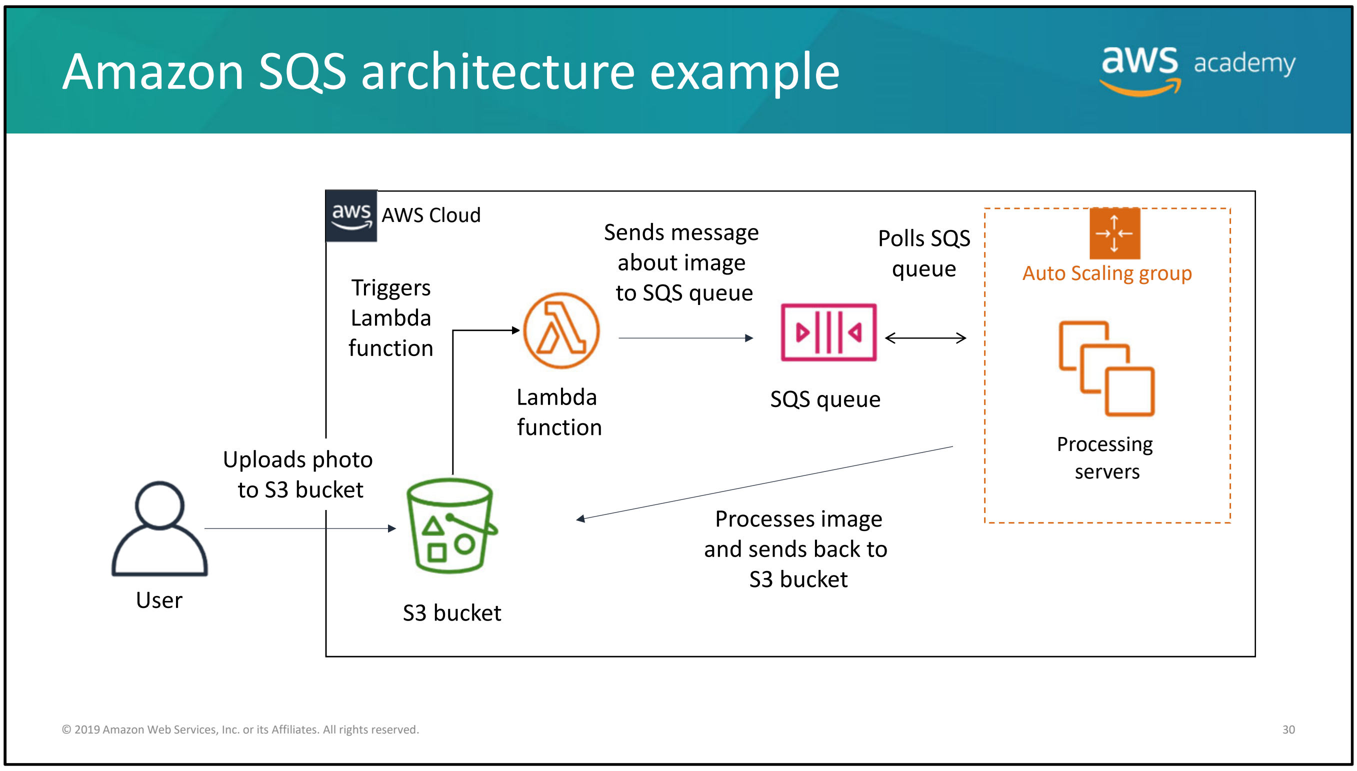
Amazon SQS offers the following benefits:

- Fully managed – AWS manages all ongoing operations and underlying infrastructure that are required to provide a highly available and scalable message queuing service. With Amazon SQS, there is no upfront cost; no need to acquire, install, and configure messaging software; and no time-consuming build-out and maintenance of supporting infrastructure.
- Delivers messages reliably – You can use Amazon SQS to transmit any volume of data at any level of throughput, without losing messages or requiring other services to be available. Amazon SQS enables you to decouple application components so that they run and fail independently, which increases the overall fault tolerance of the system. Multiple copies of every message are stored redundantly across multiple Availability Zones so that they are available when they are needed.
- Scales elastically and cost-effectively – Amazon SQS uses the AWS Cloud to dynamically scale based on demand. Amazon SQS scales elastically with your application so you don't have to worry about capacity planning and pre-provisioning. There is no limit to the number of messages per queue, and standard queues provide nearly unlimited throughput.
- Cost-effective – Costs are based on usage, which provides significant cost savings compared to the always-on model of self-managed messaging middleware.
- Keeps sensitive data secure – Amazon SQS supports server-side encryption (SSE). SSE lets you transmit sensitive data in encrypted queues by using 256-bit Advanced Encryption Standard (AES) encryption. SSE protects the contents of messages in Amazon SQS queues

by using keys that are managed in AWS Key Management Service (AWS KMS). AWS KMS is integrated with AWS CloudTrail to provide logs of all encryption key usage.

See the following resources for more information on:

- Amazon SQS: https://aws.amazon.com/sqs/
- Protecting Amazon SQS data using SSE and AWS KMS: https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-server-side-encryption.html
- Configuring AWS KMS permissions: https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-server-side-encryption.html#sqs-what-permissions-for-sse.

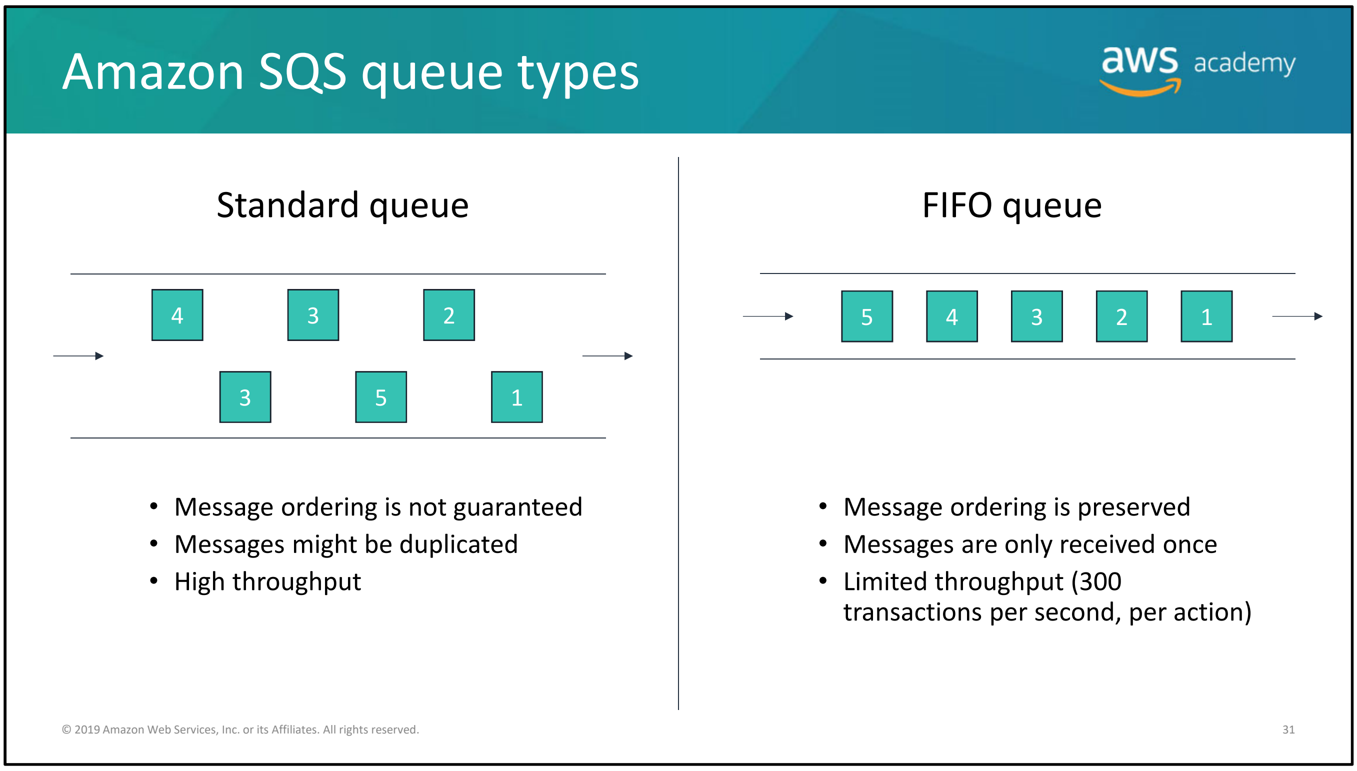


Amazon SQS architecture example

You can integrate Amazon SQS with other AWS services to make applications more reliable and scalable.

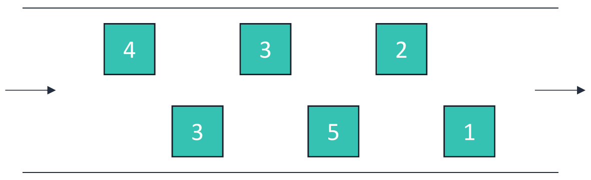
Say you have an application that processes images. A user uploads a photo to your Amazon Simple Storage Services (Amazon S3) bucket, which triggers an AWS Lambda function. The Lambda function sends a message containing information about the image to an SQS queue. A processing server sitting behind an Amazon EC2 Auto Scaling group polls the SQS queue for messages to process. After processing the photo, the processing server sends the processed photo back to the Amazon S3 bucket.

Note that the Auto Scaling group scales based on the size of the SQS queues. The processing servers keep working until the queue is empty, and then keep holding until messages appear. Further, if the backend processing servers went down, the Lambda function can keep generating messages and putting them on the queue until the Auto Scaling group was ready to start consuming messages again.



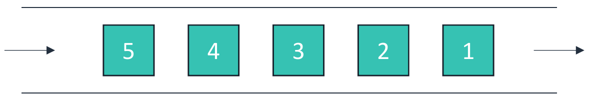
Amazon SQS queue types

Standard queue



- Message ordering is not guaranteed
- Messages might be duplicated
- High throughput

FIFO queue



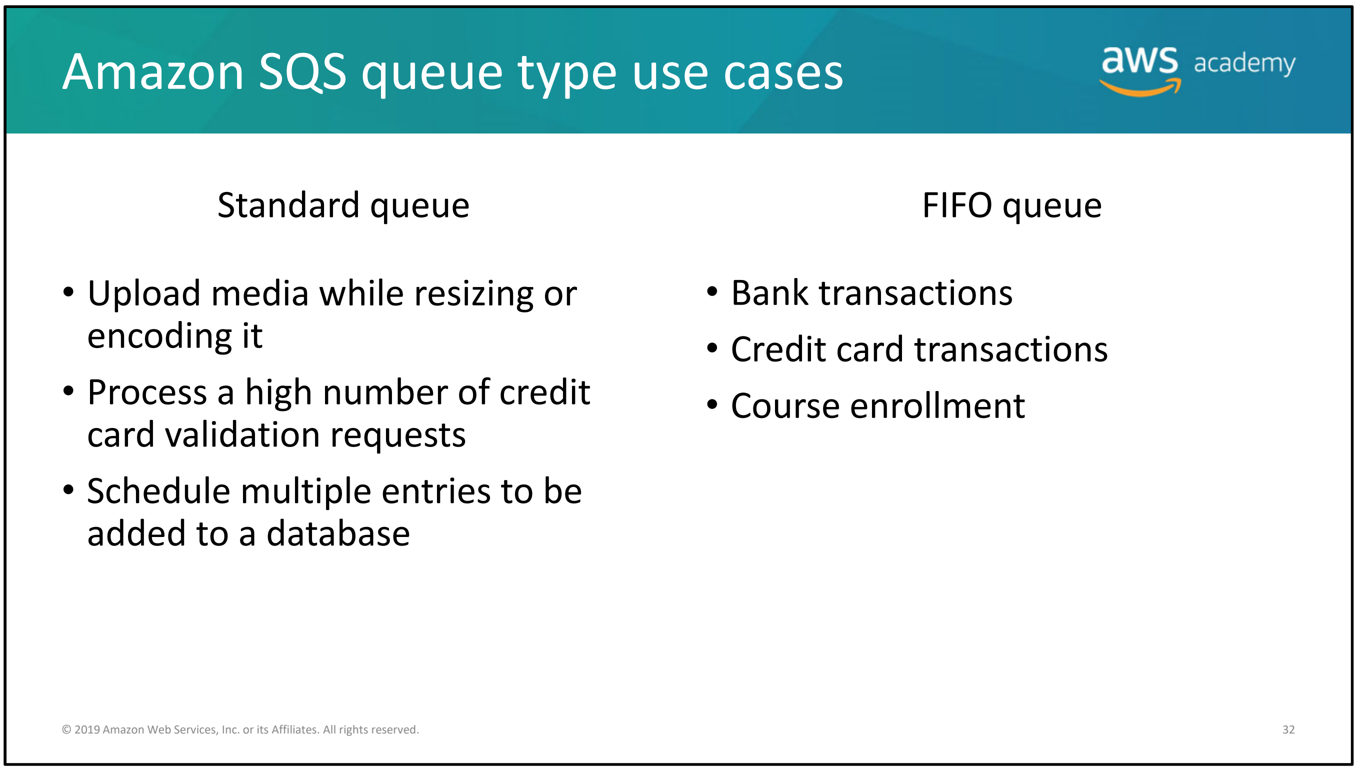
- Message ordering is preserved
- Messages are only received once
- Limited throughput (300 transactions per second, per action)

Amazon SQS supports two types of message queues:

- *Standard queues* are the default queue type. Standard queues provide best-effort ordering, which ensures that messages are generally delivered in the same order that they are sent. Standard queues support at-least-once message delivery. However, occasionally more than one copy of a message might be delivered out of order. Standard queues also support a nearly unlimited number of transactions per second (TPS) per action.
- *First-In-First-Out (FIFO) queues* are designed to enhance messaging between applications when the order of operations and events is critical, or where duplicates can't be tolerated. FIFO queues also provide exactly once processing, but they have a limited number of TPS. FIFO queues support up to 300 messages per second, per action without batching and up to 3,000 messages per second, per action with batching.

See the AWS Documentation for more information on:

- Amazon SQS standard queues: https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/standard-queues.html
- Amazon SQS FIFO queues: https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/FIFO-queues.html



Amazon SQS queue type use cases

aws academy

Standard queue

- Upload media while resizing or encoding it
- Process a high number of credit card validation requests
- Schedule multiple entries to be added to a database

FIFO queue

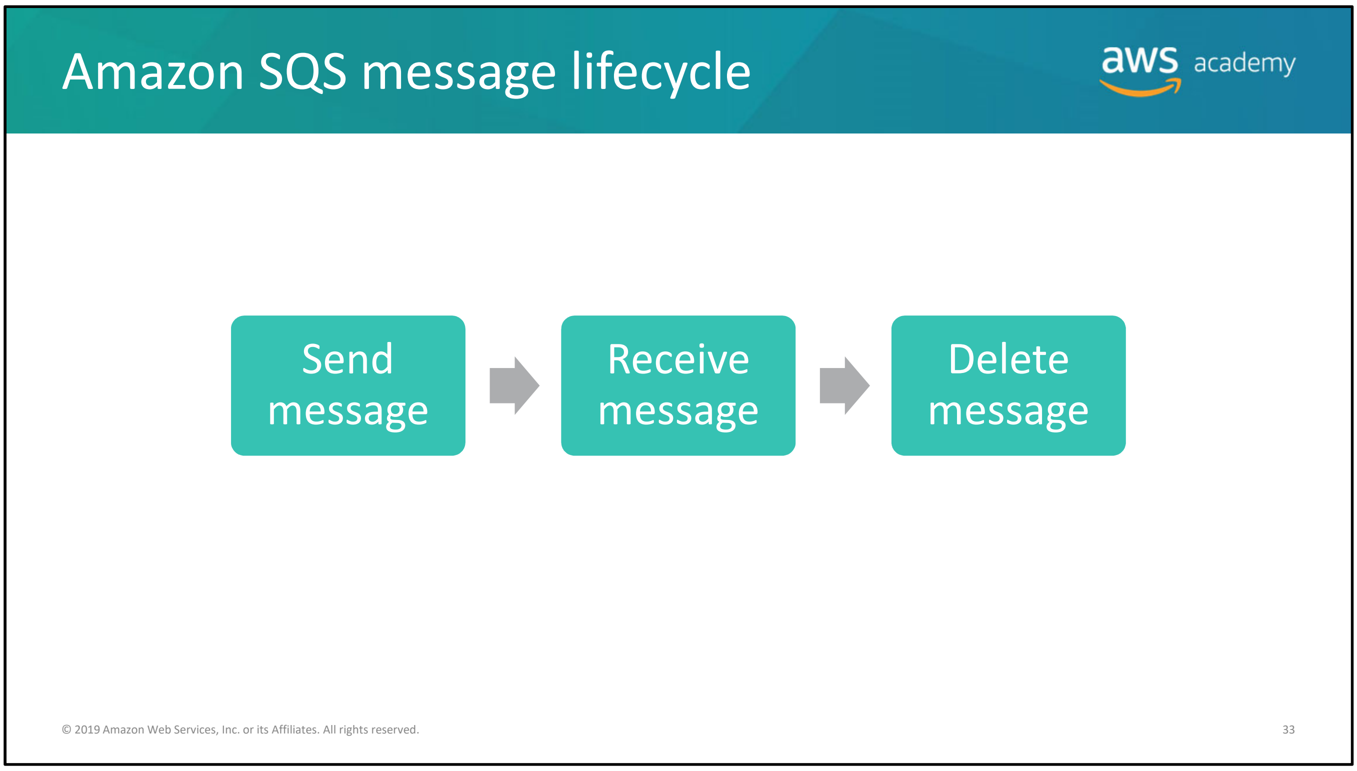
- Bank transactions
- Credit card transactions
- Course enrollment

You can use standard message queues in many scenarios, as long as your application can process messages that arrive more than once and out of order. For example, you can use a standard queue to:

- Decouple live user requests from intensive background work – Let users upload media while resizing or encoding it.
- Allocate tasks to multiple worker nodes – Process a high number of credit card validation requests.
- Batch messages for future processing – Schedule multiple entries to be added to a database.

You use FIFO queues for applications when the order of operations and events is critical, or in cases where duplicates can't be tolerated. For example, you can use a FIFO queue for:

- Bank transactions – Ensure that a deposit is recorded before a bank withdrawal happens.
- Credit card transactions – Ensure that a credit card transaction is not processed more than once.
- Course enrollment – Prevent a student from enrolling in a course before they register for an account.



Amazon SQS message lifecycle

$$\begin{tikzcd}\text{Send message} \arrow[r] & \text{Receive message} \arrow[r] & \text{Delete message}\end{tikzcd}$$

The lifecycle of an Amazon SQS message is as follows:

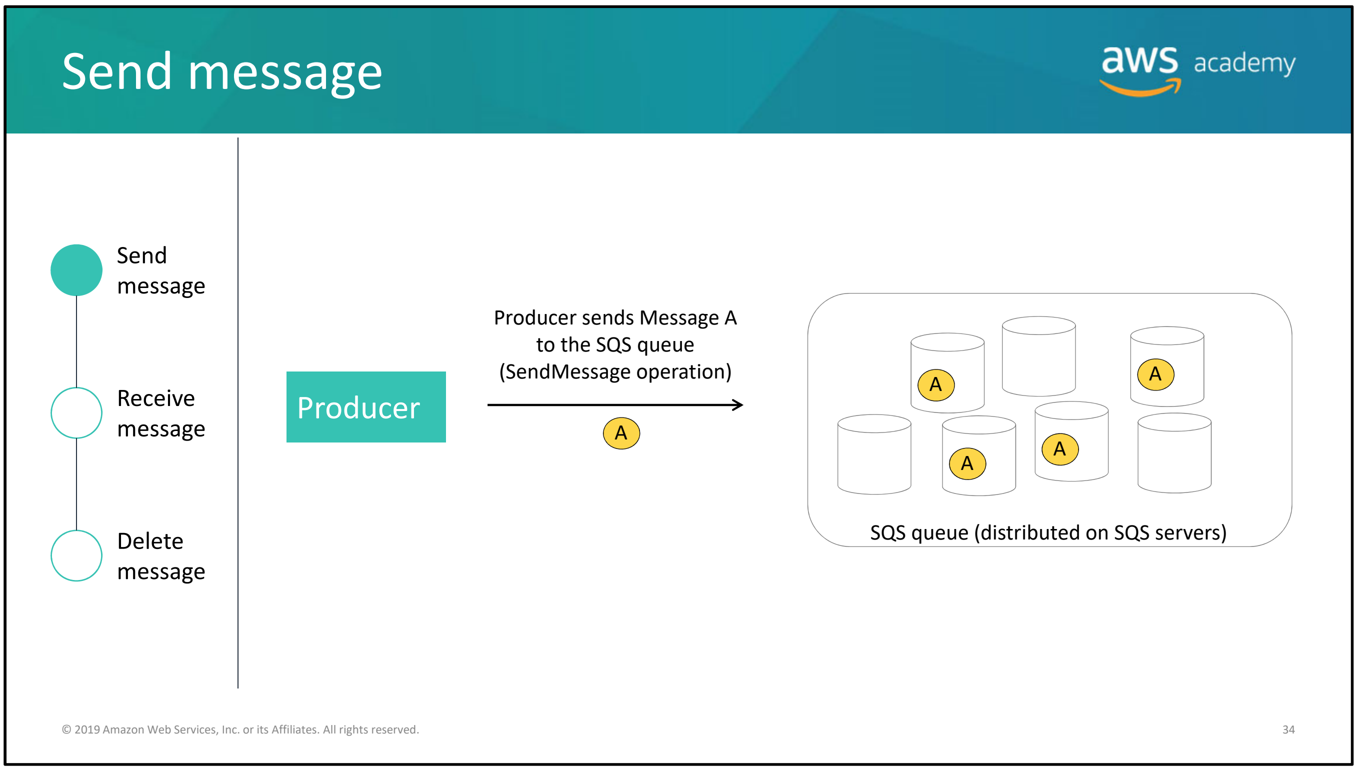
1. A producer component sends a message to the SQS queue.
2. A consumer component retrieves the message from the queue. The visibility timeout period starts.
3. The consumer component processes the message and then deletes it from the queue during the visibility timeout period.

You can perform these actions with the SendMessage, ReceiveMessage, and DeleteMessage API calls, respectively, either programmatically or from the Amazon SQS console. There are a variety of configuration options that affect cost, message processing, and message retention.

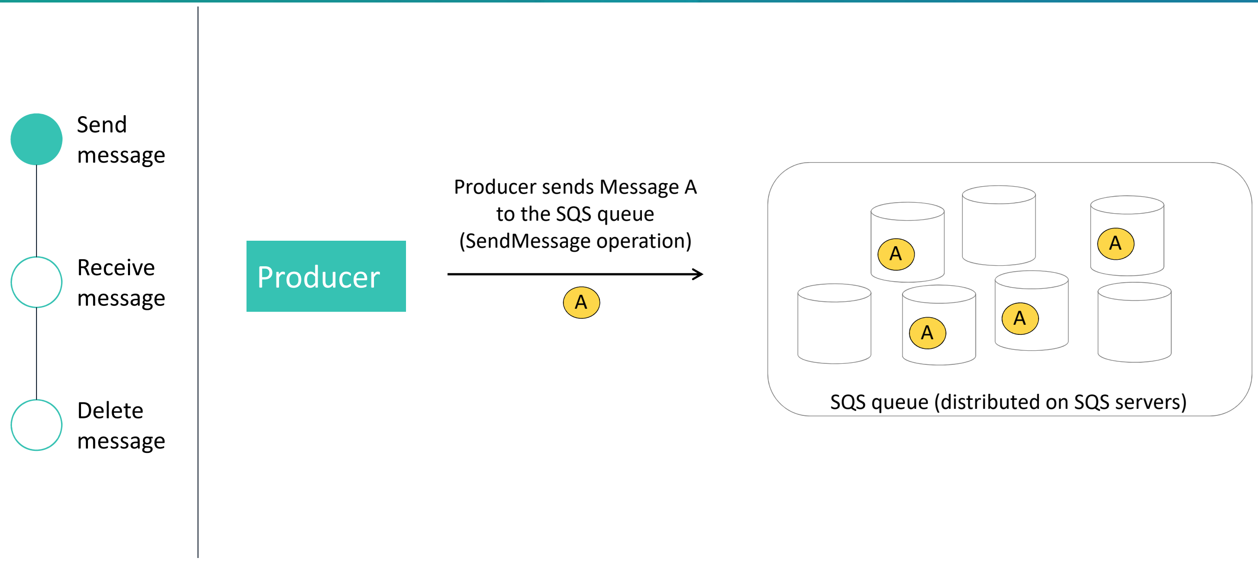
For information about pricing, see the Amazon SQS pricing page: https://aws.amazon.com/sqs/pricing/

See the AWS Documentation for more information about:

- Amazon SQS message lifecycle: https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-basic-architecture.html
- How to work with Amazon SQS using the AWS Management Console and using Java: https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-tutorials.html

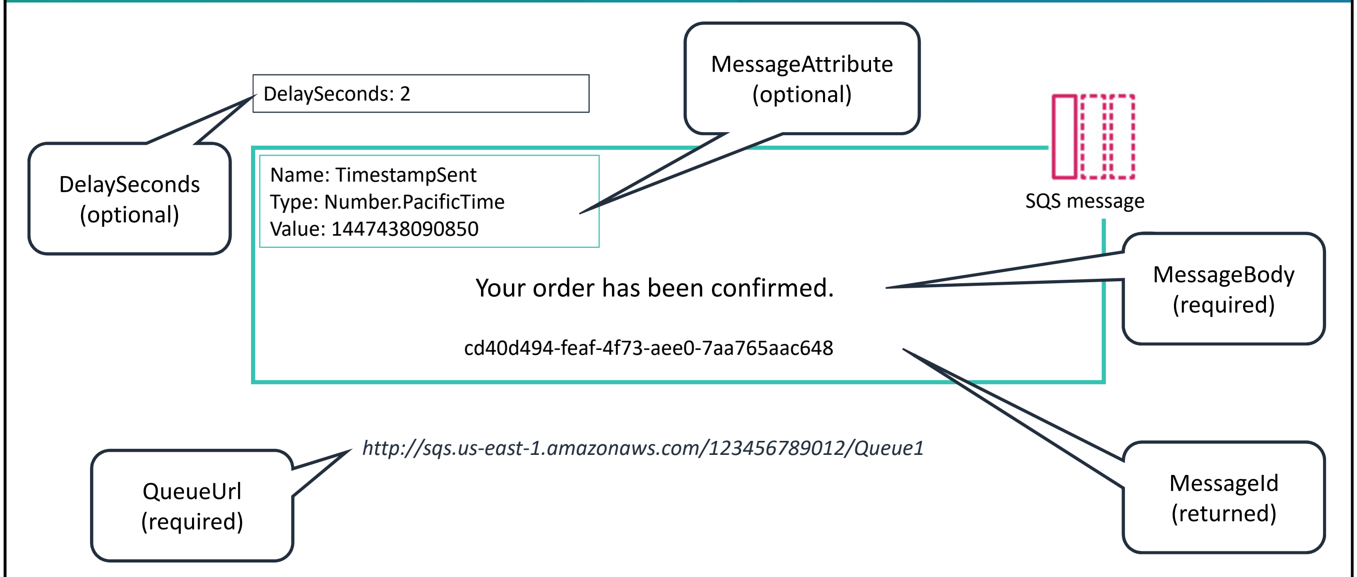


Send message


Send message
Receive message
Delete message
Producer
Producer sends Message A to the SQS queue (SendMessage operation)
A
A
A
A
A
SQS queue (distributed on SQS servers)

With Amazon SQS, you can use the SendMessage operation to send messages individually to a queue, or you can use the SendMessageBatch operation to send up to 10 messages to a queue. When a producer sends a message to an SQS queue, the queue redundantly stores the message across multiple Amazon SQS servers.

SendMessage operation



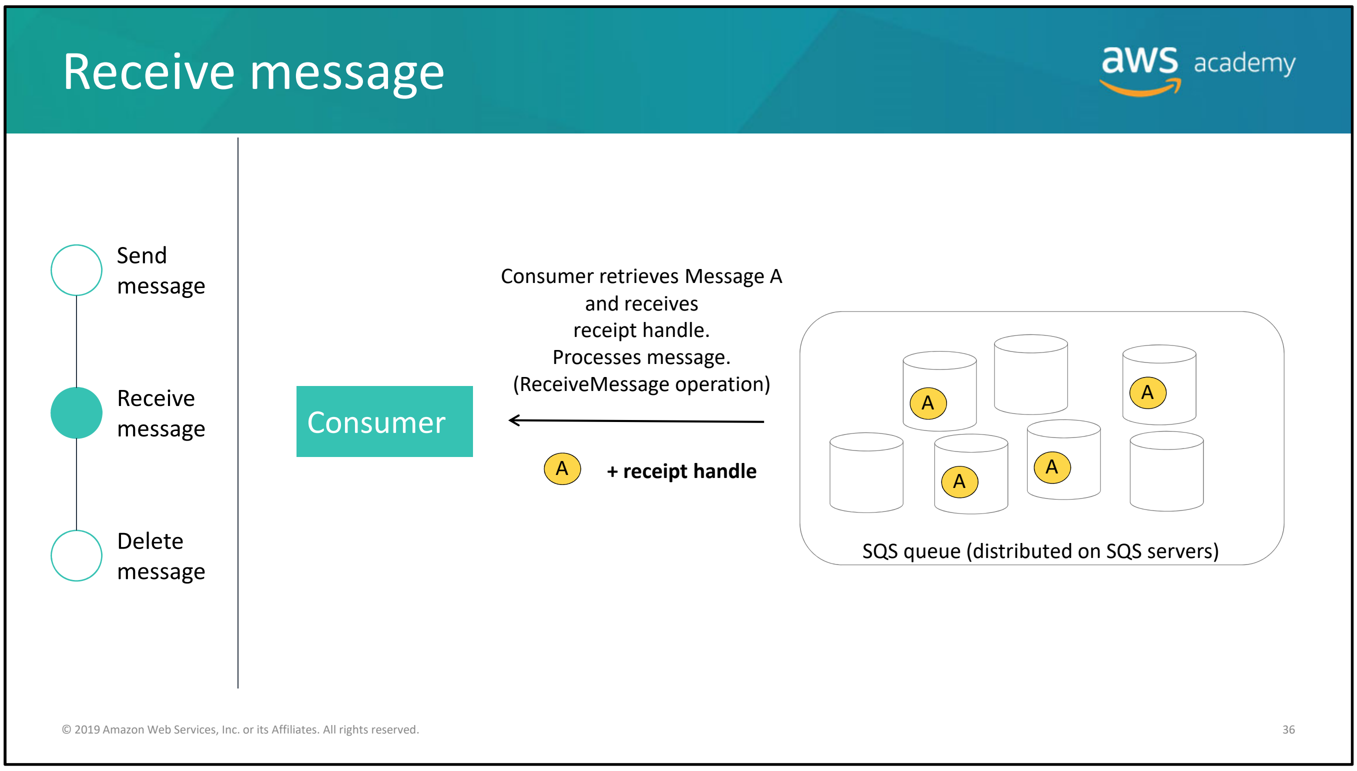
The *SendMessage* operation delivers a message to a specific queue. This operation takes the following request parameters:

- *MessageBody* (required) – The message to send. The maximum string size is 256 KB.
- *MessageAttribute* (optional) – Amazon SQS enables you to include structured metadata (such as timestamps, geospatial data, signatures, and identifiers) with messages by using *message attributes*. Each message can have up to 10 attributes. Message attributes are separate from the message body (however, they are sent alongside it). Each message attribute consists of a name, type, and value.
- *QueueUrl* (required) – The URL of the Amazon SNS queue where a message is sent.
- *DelaySeconds* (optional) – The length of time, in seconds, to delay a specific message.

After you send a message, Amazon SQS returns a *MessageId*, which is a systemassigned identifier that is useful for identifying messages.

See the AWS Documentation for more information about:

- SendMessage API call: https://docs.aws.amazon.com/AWSSimpleQueueService/latest/APIReference/API_SendMessage.html
- Queue name and URL and message IDs: https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-general-identifiers.html



Receive message

aws academy
Send message
Receive message
Delete message
Consumer
Consumer retrieves Message A and receives receipt handle. Processes message. (ReceiveMessage operation)
A
+ receipt handle
A
A
A
A
SQS queue (distributed on SQS servers)

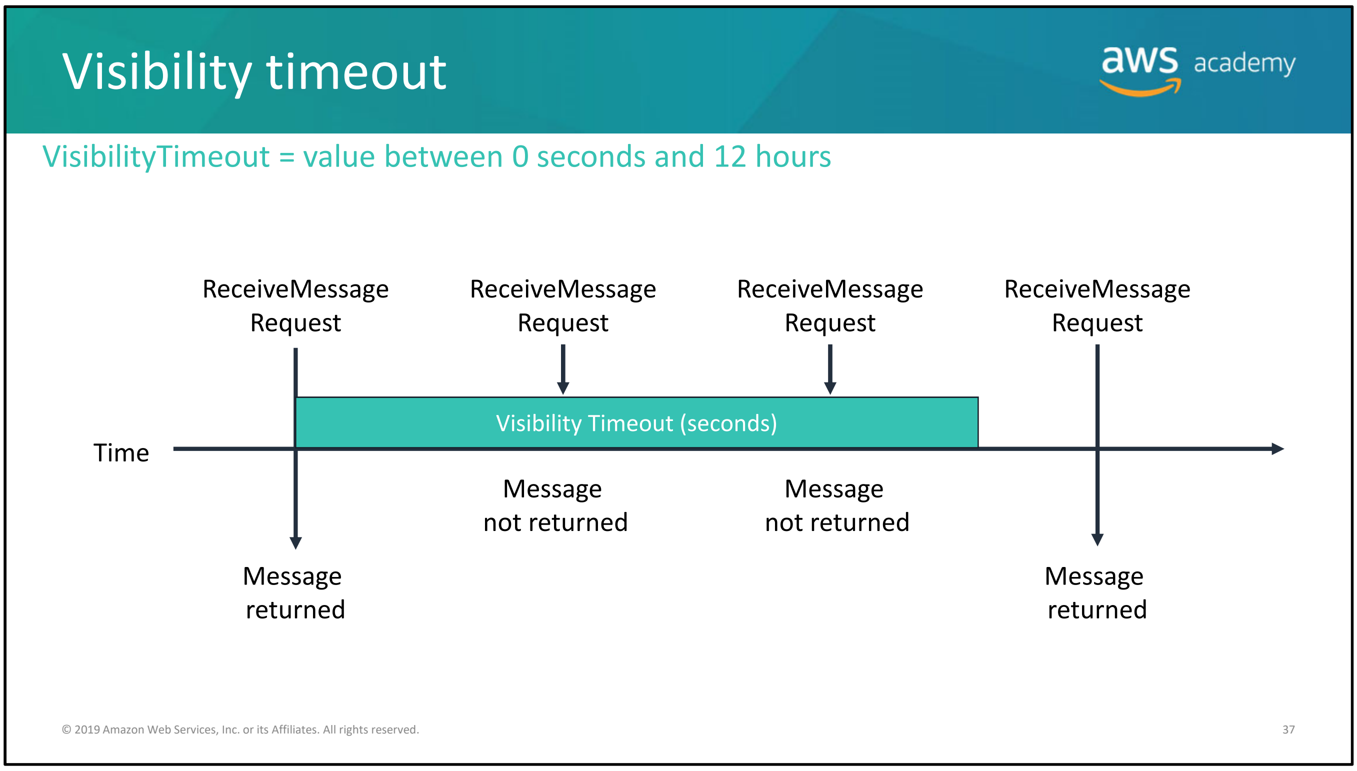
You can retrieve one or more messages (up to 10) from the queue by using the *ReceiveMessage operation*. You set the number of messages that you want to retrieve by using the MaxNumberOfMessages parameter.

Each time you receive a message from a queue, you receive a *receipt handle* for that message. This handle is associated with the action of receiving the message, not with the message itself. If you receive a message more than once, you get a different receipt handle each time you receive it.

To delete the message or to change the visibility of the message, you must provide the receipt handle (not the message ID). Thus, you must always receive a message before you can delete it. (You can't put a message into the queue and then recall it.) The maximum length of a receipt handle is 1,024 characters.

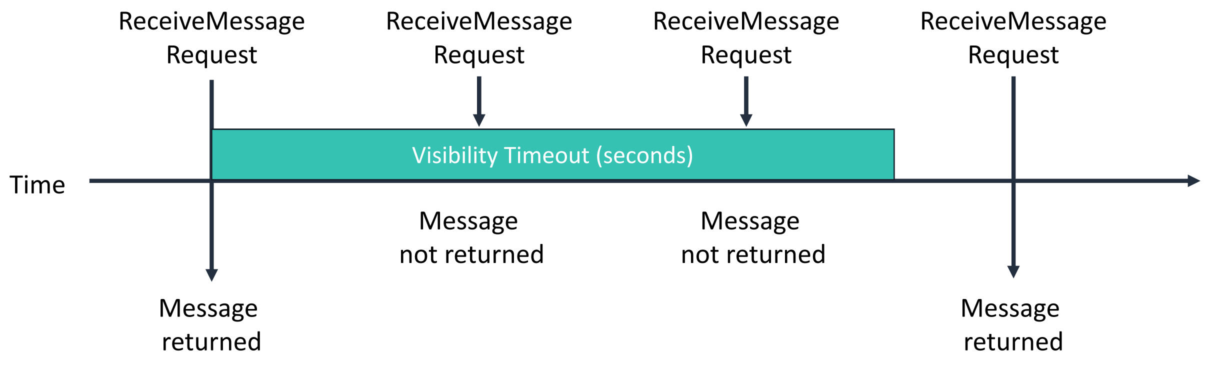
See the AWS Documentation for more information about

- ReceiveMessage API call: https://docs.aws.amazon.com/AWSSimpleQueueService/latest/APIReference/API_ReceiveMessage.html
- Receipt handles: https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-general-identifiers.html



Visibility timeout

VisibilityTimeout = value between 0 seconds and 12 hours



When a consumer receives and processes a message from a queue, the message remains in the queue. 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 because of an issue in the consumer application). Thus, the consumer must delete the message from the queue after it receives and processes it.

Immediately after a message is received, it remains in the queue. To prevent other consumers from processing the message again, Amazon SQS sets a *visibility timeout*, which is a period of time where Amazon SQS prevents other consumers from receiving and processing the message. The default visibility timeout for a message is 30 seconds. The minimum is 0 seconds. The maximum is 12 hours. You set this value by using the VisibilityTimeout parameter.

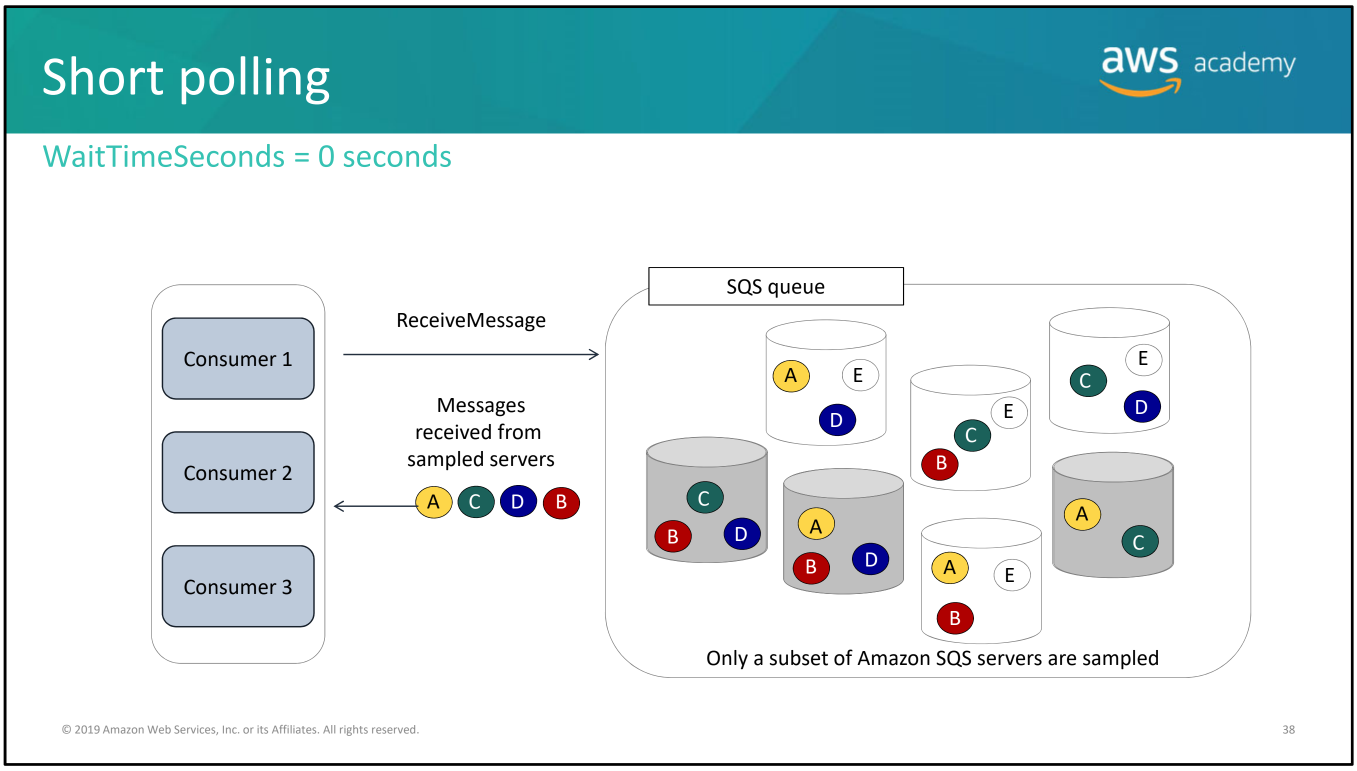
The visibility timeout begins when Amazon SQS returns a message. During this time, the consumer processes and deletes the message. However, if the consumer fails before it deletes the message and your system doesn't delete the message before the visibility timeout expires, the message becomes visible to other consumers and the message is received again.

FIFO queues allow the producer or consumer to attempt multiple retries:

- If the producer detects a failed SendMessage action, it can retry sending as many times as necessary.
- If the consumer detects a failed ReceiveMessage action, it can retry as many times as necessary.

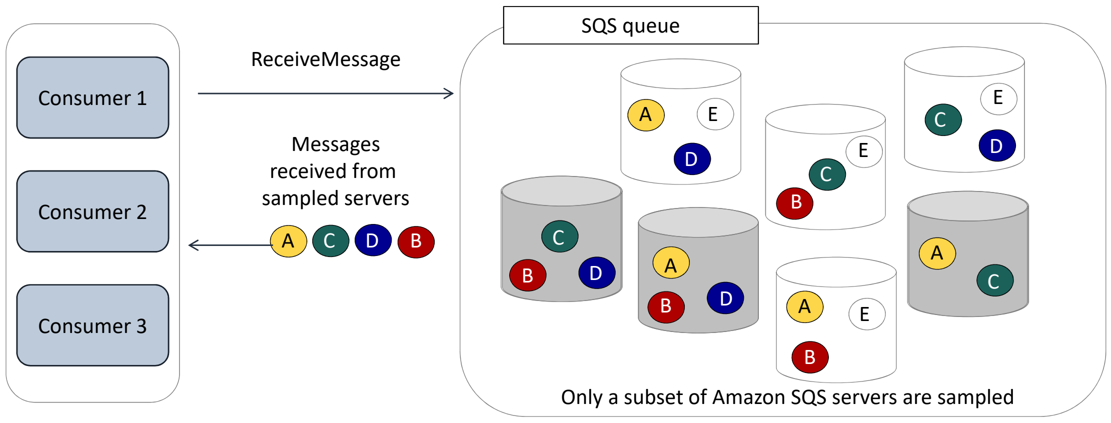
For more information about visibility timeout, see the AWS Documentation:
https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-

visibility-timeout.html



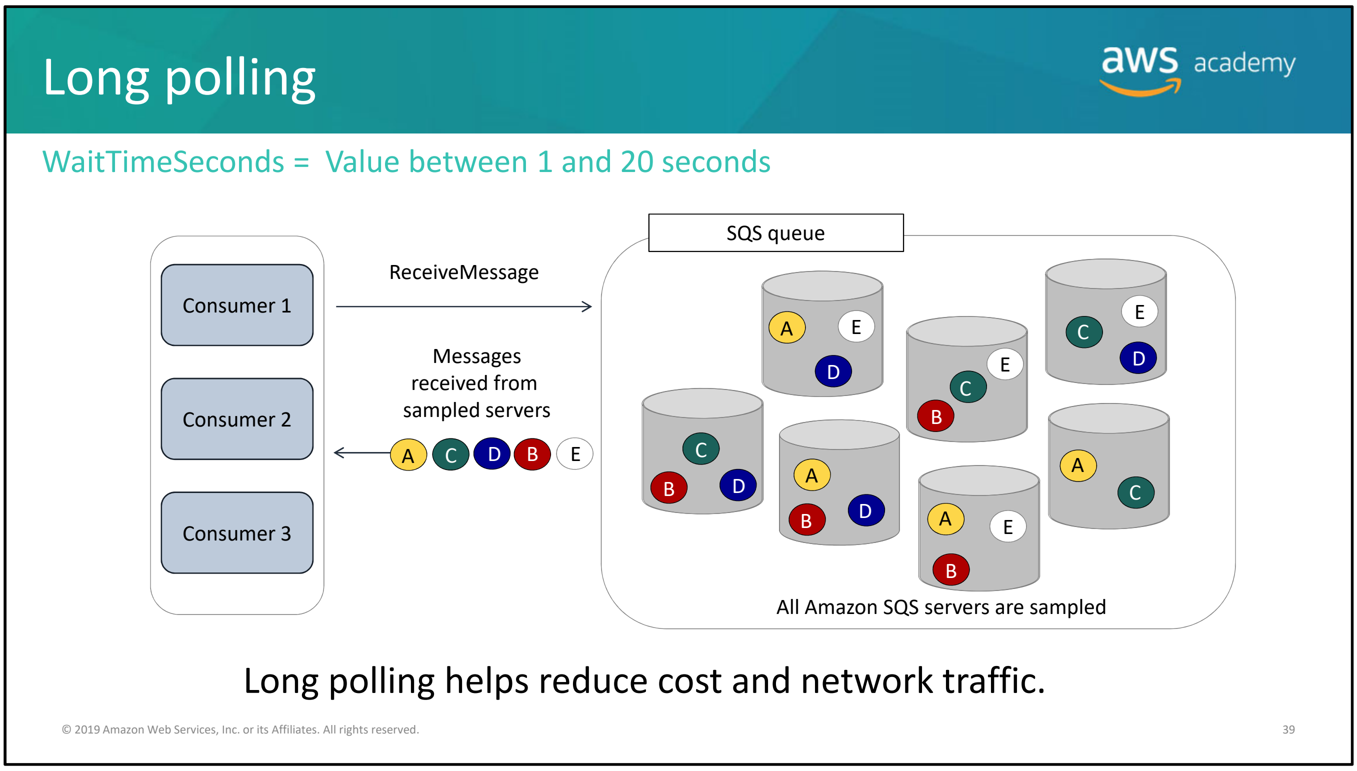
Short polling

WaitTimeSeconds = 0 seconds



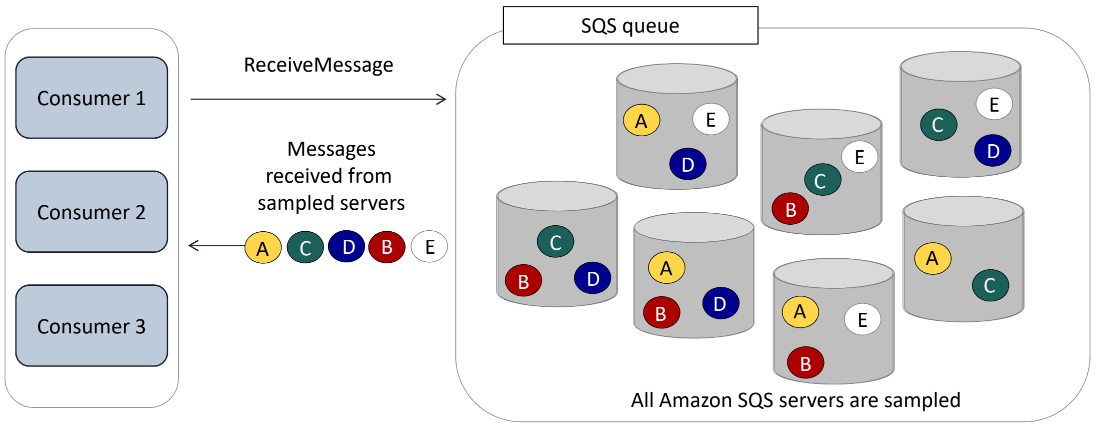
Amazon SQS uses a polling model to retrieve messages from a queue. By default, when you make a ReceiveMessage API call, Amazon SQS performs *short polling*, in which it samples a *subset* of servers (based on a weighted random distribution). It immediately returns only the messages that are on the sampled servers. In the example, short polling returns messages A, B, C, and D, but it does not return message E. Short polling occurs when the `WaitTimeSeconds` parameter of a `ReceiveMessage` call is set to 0.

If the number of messages in the queue is small (fewer than 1,000), you will most likely get fewer messages than you requested per ReceiveMessage call. If the number of messages in the queue is extremely small, you might not receive any messages from a particular ReceiveMessage call. If you keep requesting ReceiveMessage, Amazon SQS will sample all the servers and you will eventually receive all of your messages.



Long polling

WaitTimeSeconds = Value between 1 and 20 seconds

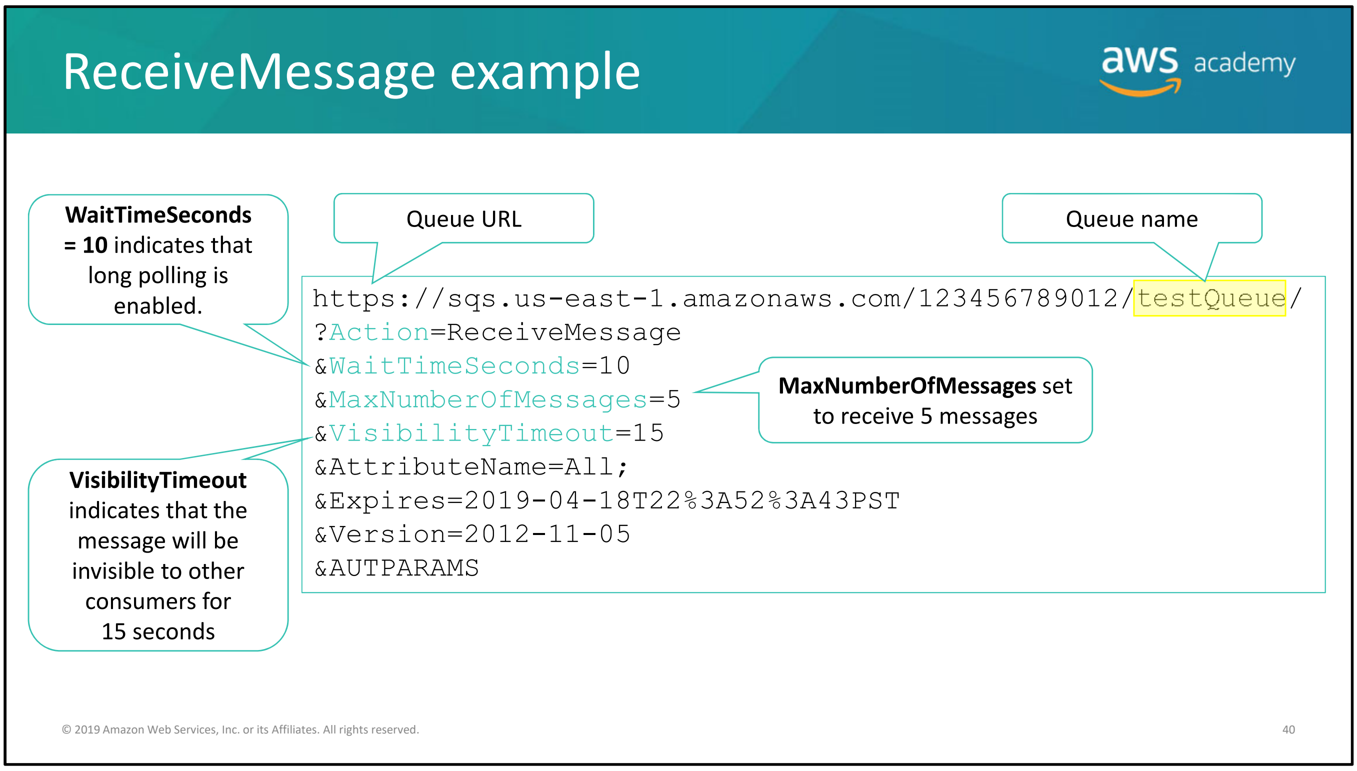


SQS queue
Consumer 1
Consumer 2
Consumer 3
ReceiveMessage
Messages received from sampled servers
A C D B E
All Amazon SQS servers are sampled

Long polling helps reduce cost and network traffic.

By contrast, with *long polling*, Amazon SQS queries *all* of the servers and waits until a message is available in the queue before it sends a response. Long polling helps reduce the cost of using Amazon SQS by eliminating the number of empty responses (when no messages are available for a ReceiveMessage request) and false empty responses (when messages are available, but aren't included in a response). You can enable long polling by setting the WaitTimeSeconds parameter of the ReceiveMessage request to a non-zero value between 1 and 20 seconds.

For more information about Amazon SQS long polling, see the AWS Documentation: https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-long-polling.html



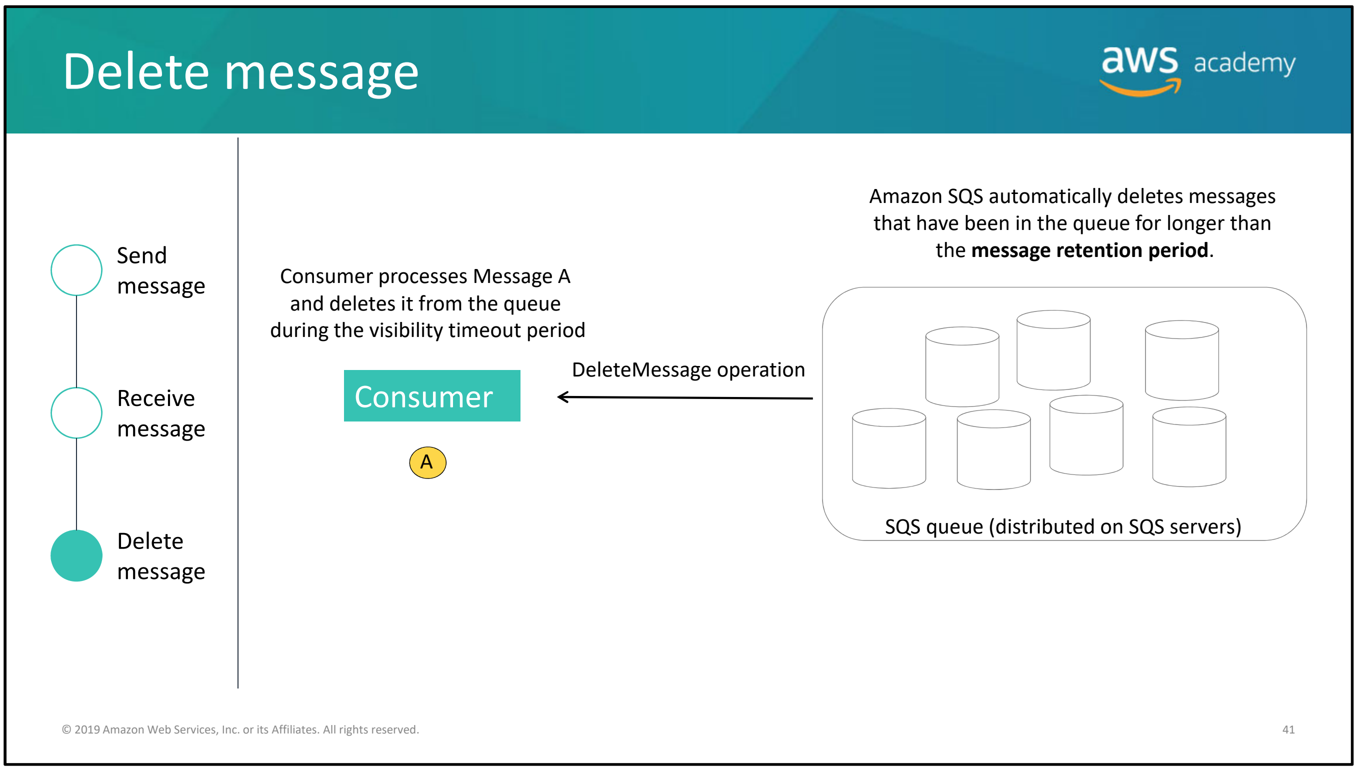
ReceiveMessage example

aws academy
WaitTimeSeconds = 10 indicates that long polling is enabled.
Queue URL
Queue name
MaxNumberOfMessages set to receive 5 messages
VisibilityTimeout indicates that the message will be invisible to other consumers for 15 seconds

```
https://sqs.us-east-1.amazonaws.com/123456789012/testQueue/
?Action=ReceiveMessage
&WaitTimeSeconds=10
&MaxNumberOfMessages=5
&VisibilityTimeout=15
&AttributeName=All;
&Expires=2019-04-18T22%3A52%3A43PST
&Version=2012-11-05
&AUTPARAMS
```

Here is an example of a query to retrieve messages from a queue named *testQueue*.

In this example:
- *MaxNumberOfMessages* parameter is set to 5, which indicates the maximum number of messages to return.
- *WaitTimeSeconds* parameter is set to 10, which indicates that long polling is enabled.
- *VisibilityTimeout* parameter is set to 15, which indicates that the message will be invisible to other consumers for processing for 15 seconds.



Delete message

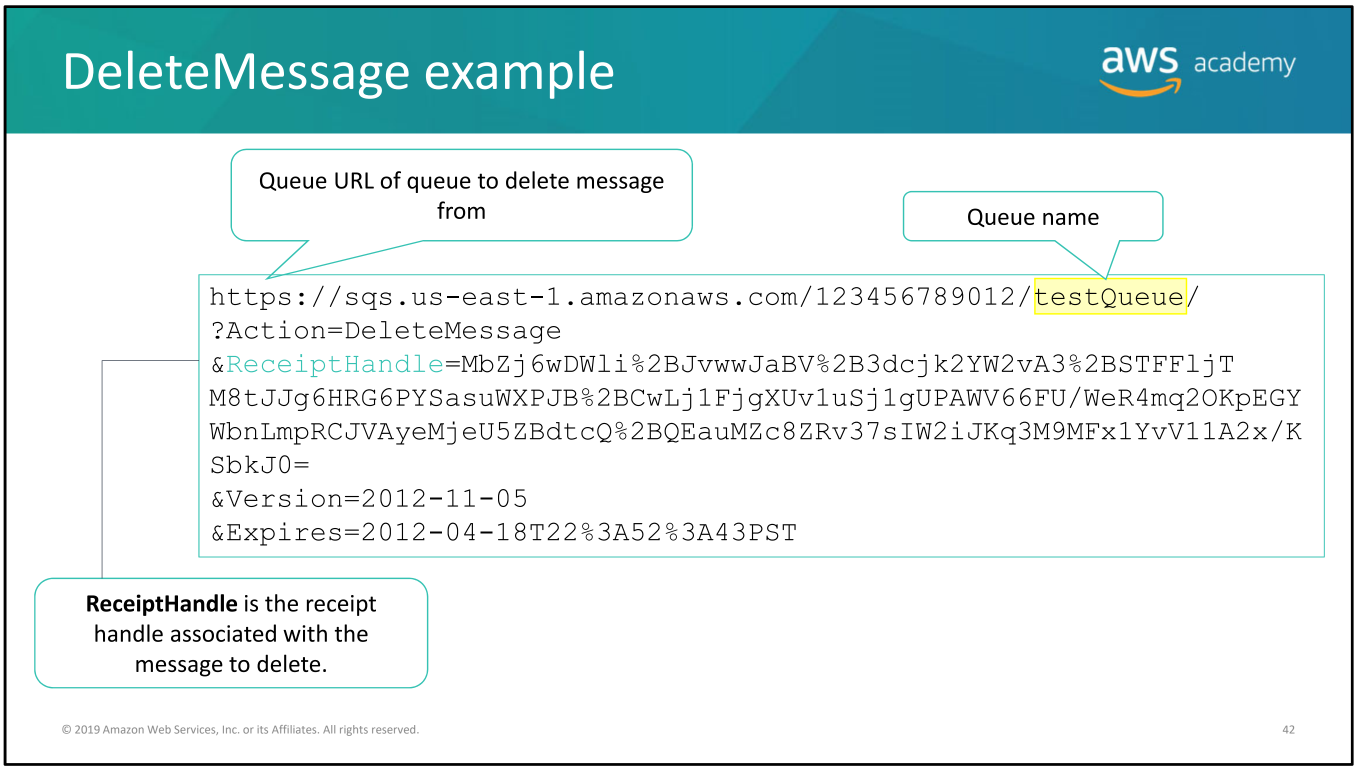
aws academy
Send message
Receive message
Delete message
Consumer processes Message A and deletes it from the queue during the visibility timeout period
Consumer
A
DeleteMessage operation
Amazon SQS automatically deletes messages that have been in the queue for longer than the **message retention period**.
SQS queue (distributed on SQS servers)

To prevent a message from being received and processed again when the visibility timeout expires, the consumer must delete the message. You can use the DeleteMessage operation to delete a specific message from a specific queue, or you can use DeleteMessageBatch to delete up to 10 messages. To select the message to delete, use the receipt handle of the message.

Amazon SQS automatically deletes messages that have been in a queue longer than the queue's configured message retention period. The default message retention period is 4 days. However, you can set the message retention period to a value from 60 seconds to 1,209,600 seconds (14 days) by using the SetQueueAttributes action.

See the AWS Documentation for more information on:

- DeleteMessage API call: https://docs.aws.amazon.com/AWSSimpleQueueService/latest/APIReference/API_DeleteMessage.html
- SetQueueAttributes action: https://docs.aws.amazon.com/AWSSimpleQueueService/latest/APIReference/API_SetQueueAttributes.html



DeleteMessage example

Queue URL of queue to delete message from

Queue name

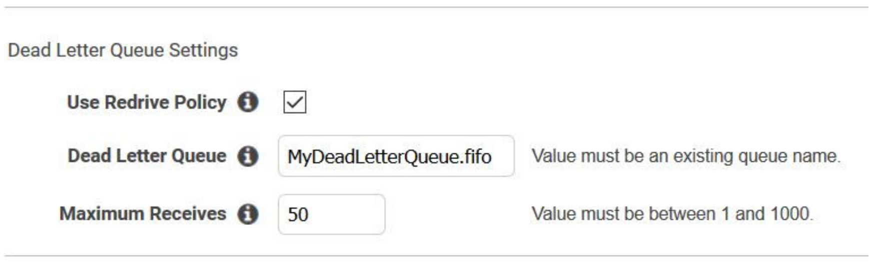
```
https://sqs.us-east-1.amazonaws.com/123456789012/testQueue/
?Action=DeleteMessage
&ReceiptHandle=MbZj6wDWli%2BJvwwJaBV%2B3dcjk2YW2vA3%2BSTFFljT
M8tJJg6HRG6PYSasuWXPJB%2BCwLj1FjgXUv1uSj1gUPAWV66FU/WeR4mq2OKpEGY
WbnLmpRCJVAyeMjeU5ZBdtcQ%2BQEauMZc8ZRv37sIW2iJKq3M9MFx1YvV11A2x/K
SbkJ0=
&Version=2012-11-05
&Expires=2012-04-18T22%3A52%3A43PST
```

ReceiptHandle is the receipt handle associated with the message to delete.

Here is an example of a query request that deletes a message from a queue named *testQueue*. The parameter *ReceiptHandle* is the receipt handle that's associated with the message to delete.

Dead-letter queues

- Queue of messages that could not be processed
- Use with standard queues
- Use to help troubleshoot incorrect message transmission operations



Dead Letter Queue Settings

Use Redrive Policy	✓	
Dead Letter Queue	MyDeadLetterQueue.fifo	Value must be an existing queue name.
Maximum Receives	50	Value must be between 1 and 1000.

For messages that cannot be processed, you can create a dead-letter queue in Amazon SQS to receive messages from other SQS queues (which are referred to as *source queues*) after the maximum number of processing attempts has been reached. Dead-letter queues can help you troubleshoot incorrect transmission operations.

Messages can be sent to and received from a dead-letter queue just like any other Amazon SQS queue. You can create a dead-letter queue from the Amazon SQS API or the Amazon SQS console.

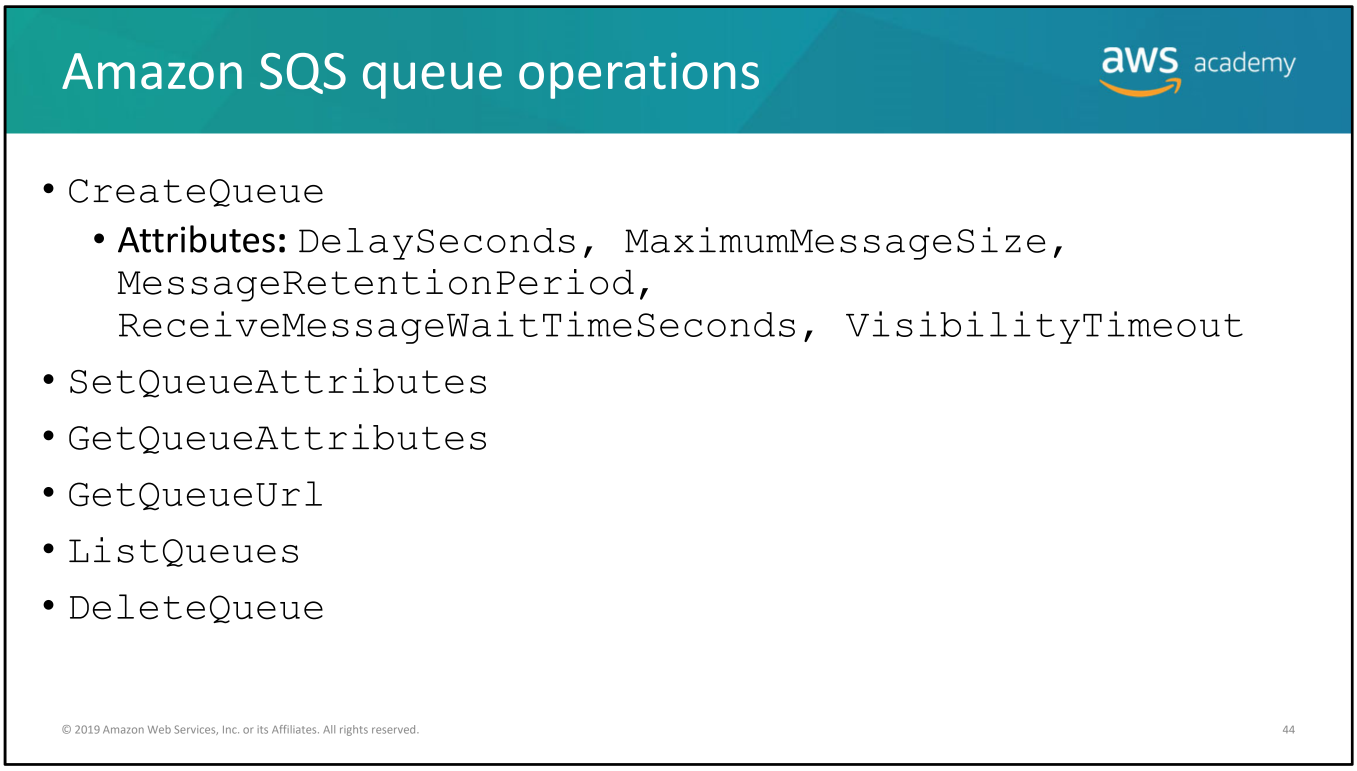
This slide illustrates how to configure a dead-letter queue from the console. To configure the dead-letter queue:

1. Choose **Use Redrive Policy**.
2. Enter the name of the existing **Dead Letter Queue** where you want source queues to send messages.
3. Set **Maximum Receives** to configure the number of times that a message can be received before it is sent to a dead-letter queue.

See the AWS Documentation for more information about:

- Amazon SQS dead-letter queues: https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-dead-letter-queues.html
- How to configure an Amazon SQS dead-letter queue: https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-

configure-dead-letter-queue.html


Amazon SQS queue operations

- `CreateQueue`
  - **Attributes:** `DelaySeconds, MaximumMessageSize, MessageRetentionPeriod, ReceiveMessageWaitTimeSeconds, VisibilityTimeout`
- `SetQueueAttributes`
- `GetQueueAttributes`
- `GetQueueUrl`
- `ListQueues`
- `DeleteQueue`

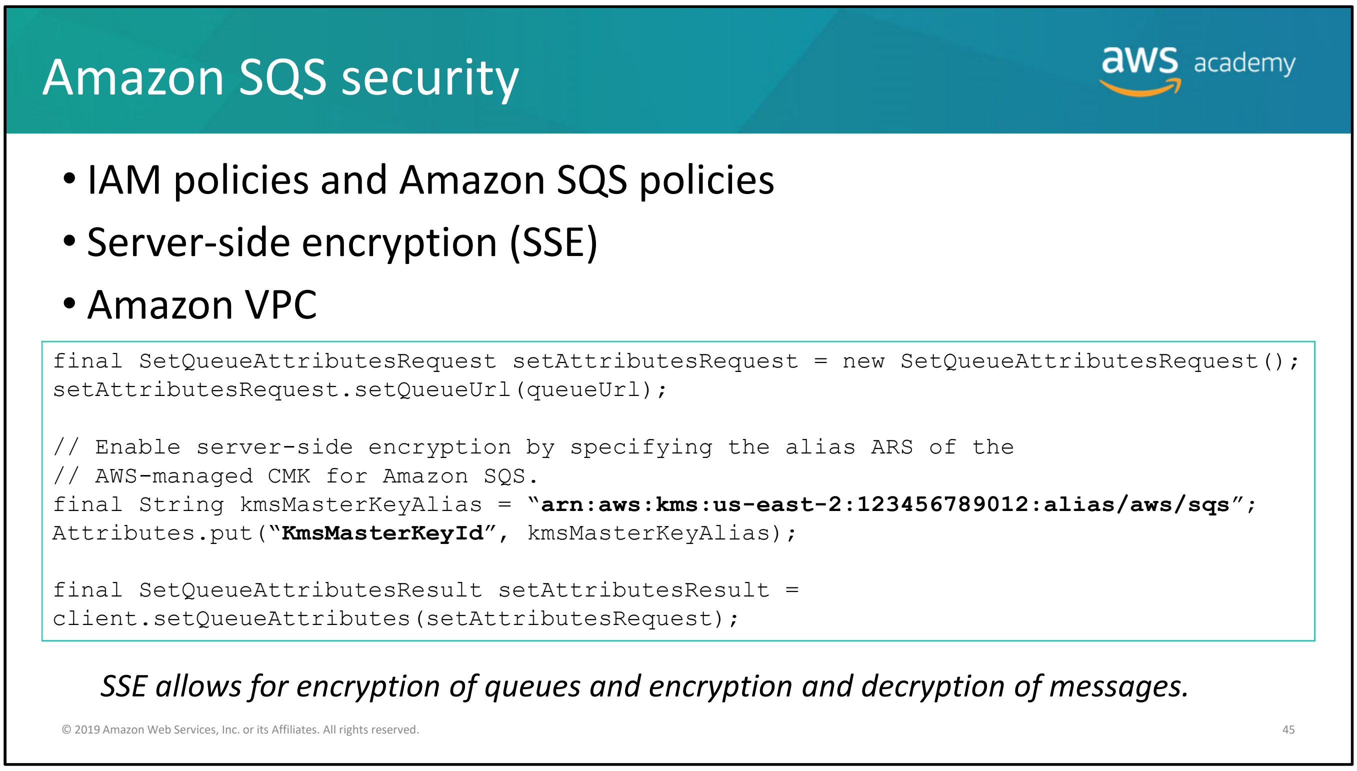
Though this section has focused on the SendMessage, ReceiveMessage, and DeleteMessage API calls, you can also use the *CreateQueue* API call to configure many of the same parameters. For example, you can enable long polling either by setting WaitTimeSeconds parameter for a *ReceiveMessage* call or by setting the `ReceiveMessageWaitTimeSeconds` parameter for a *CreateQueue* call.

There are also several other basic queue options that you should be familiar with:

- *SetQueueAttributes* and *GetQueueAttributes* – Set and retrieve the attributes of a queue, respectively. These options help you estimate the resources required to process SQS messages.
- *GetQueueUrl* – Retrieves the URL of an Amazon SQS queue.
- *ListQueues* – Retrieves a list of your queues.
- *DeleteQueue* – Deletes a queue regardless of whether the queue is empty. When a queue is deleted, any messages in the queue are no longer available. It is important to

note that Amazon SQS can delete your queue without notification if actions like *SendMessage, ReceiveMessage, DeleteMessage, GetQueueAttributes, SetQueueAttributes, AddPermission*, and *RemovePermissions* have not been performed on the queue for 30 consecutive days.

For more information about Amazon SQS API operations, see the AWS Documentation: https://docs.aws.amazon.com/AWSSimpleQueueService/latest/APIReference/Welcome.html



Amazon SQS security

- IAM policies and Amazon SQS policies
- Server-side encryption (SSE)
- Amazon VPC

```
final SetQueueAttributesRequest setAttributesRequest = new SetQueueAttributesRequest();
setAttributesRequest.setQueueUrl(queueUrl);

// Enable server-side encryption by specifying the alias ARS of the
// AWS-managed CMK for Amazon SQS.
final String kmsMasterKeyAlias = "arn:aws:kms:us-east-2:123456789012:alias/aws/sqs";
Attributes.put("KmsMasterKeyId", kmsMasterKeyAlias);

final SetQueueAttributesResult setAttributesResult =
client.setQueueAttributes(setAttributesRequest);
```

SSE allows for encryption of queues and encryption and decryption of messages.

There are three ways to secure your Amazon SQS resources:

- AWS Identity and Access Management (IAM) policies and Amazon SQS policies – Access to Amazon SQS requires credentials that AWS can use to authenticate your requests. These credentials must have permissions to access AWS resources, such as Amazon SQS queues and messages. Amazon SQS has its own resource-based permissions system that uses policies that are written in the same language that is used for IAM policies. Thus, you can achieve similar things with Amazon SQS policies and IAM policies.
- Server-side encryption (SSE) – Server-side encryption (SSE) lets you transmit sensitive data in encrypted queues. SSE protects the contents of messages in Amazon SQS queues by using keys that are managed in AWS Key Management Service (AWS KMS).
- Amazon Virtual Private Cloud (Amazon VPC) – If you use Amazon VPC to host your AWS resources, you can establish a connection between your VPC and Amazon SQS. You can use this connection to send messages to your Amazon SQS queues without crossing the public internet.

For more information about Amazon SQS security, see the AWS Documentation: https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-security.html

This week – Decoupled Architectures

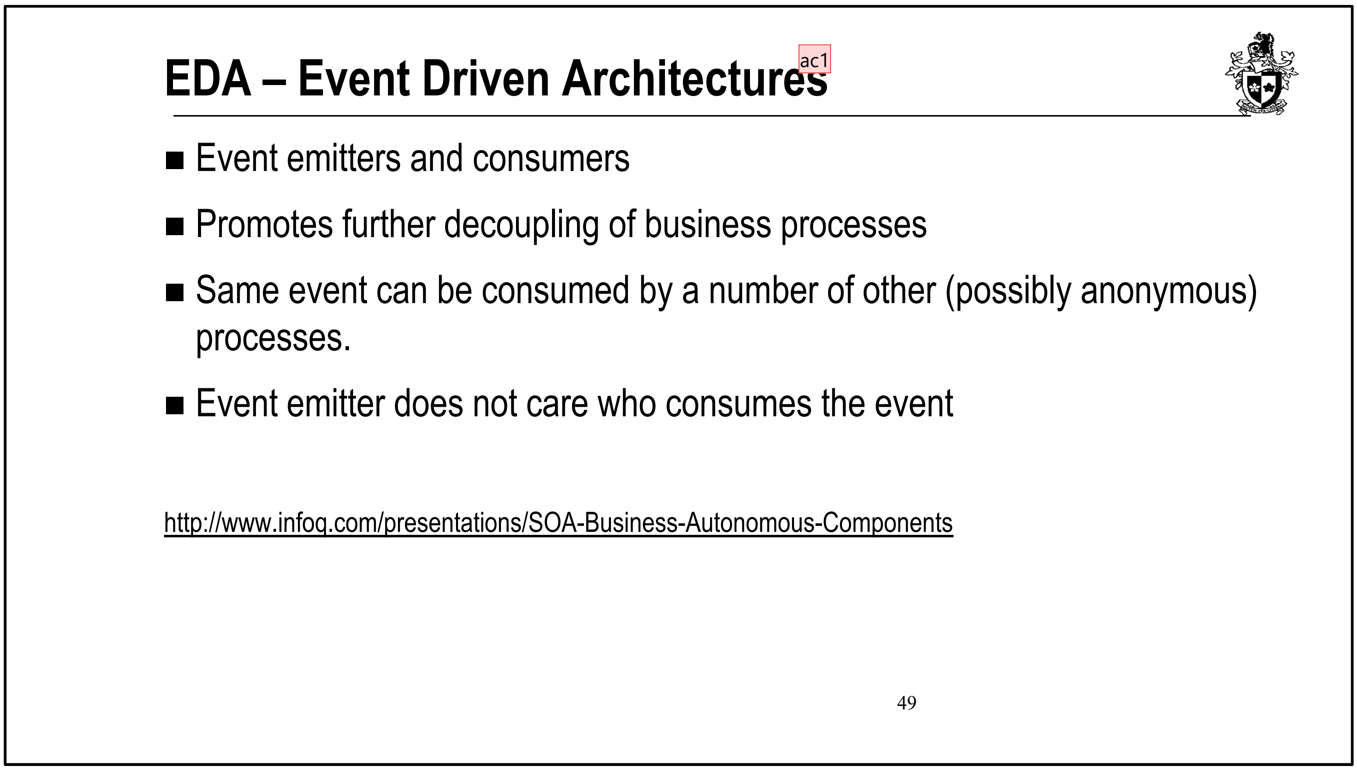
- ☐Loose coupling
- ☐Message Driven Architectures
  - ☐Message Queues
  - ☐MoM middleware, JMS, ActiveMQ
  - ☐AWS SQS (Simple Message Queue)
- ☐ Event Driven Architectures
  - ☐Events and Complex Event Processing
  - ☐Publish Subscribe (pub-sub) Systems
  - ☐AWS SNS (Simple Notification Service)
  - ☐ Serverless and event- driven applications

Events and Complex Events

- An event from a system's point of view is something that happens that
  - the systems senses
  - is of interest
- A complex event happens if other events happen
  - E.g. Car in Showroom that you like is only there because of a number of previous events
    - - events in inventory control of factory and dealer
    - - shipping events
  - Events related to each other by time, causality and aggregation.

Event-driven Asynchronous Architectures

- Event-Driven Architectures
  - Processes communicate by accessing shared events in an “event cloud” (not necessarily a overlay network as in pub-sub)
  - Rules based Complex Event Processing (CEP)
    - “Data is dynamic – Queries are Static”
- Publish-Subscribe Architectures
  - Type of architecture for subscribing to events and data of a network of routers
  - Network ‘Overlay’ that connects subscribers to publishers
  - Inter-process notification or information dissemination rather than inter-process commands
  - Topic/Content based



EDA – Event Driven Architectures

ac1

- Event emitters and consumers
- Promotes further decoupling of business processes
- Same event can be consumed by a number of other (possibly anonymous) processes.
- Event emitter does not care who consumes the event

http://www.infoq.com/presentations/SOA-Business-Autonomous-Components

49

Slide 49

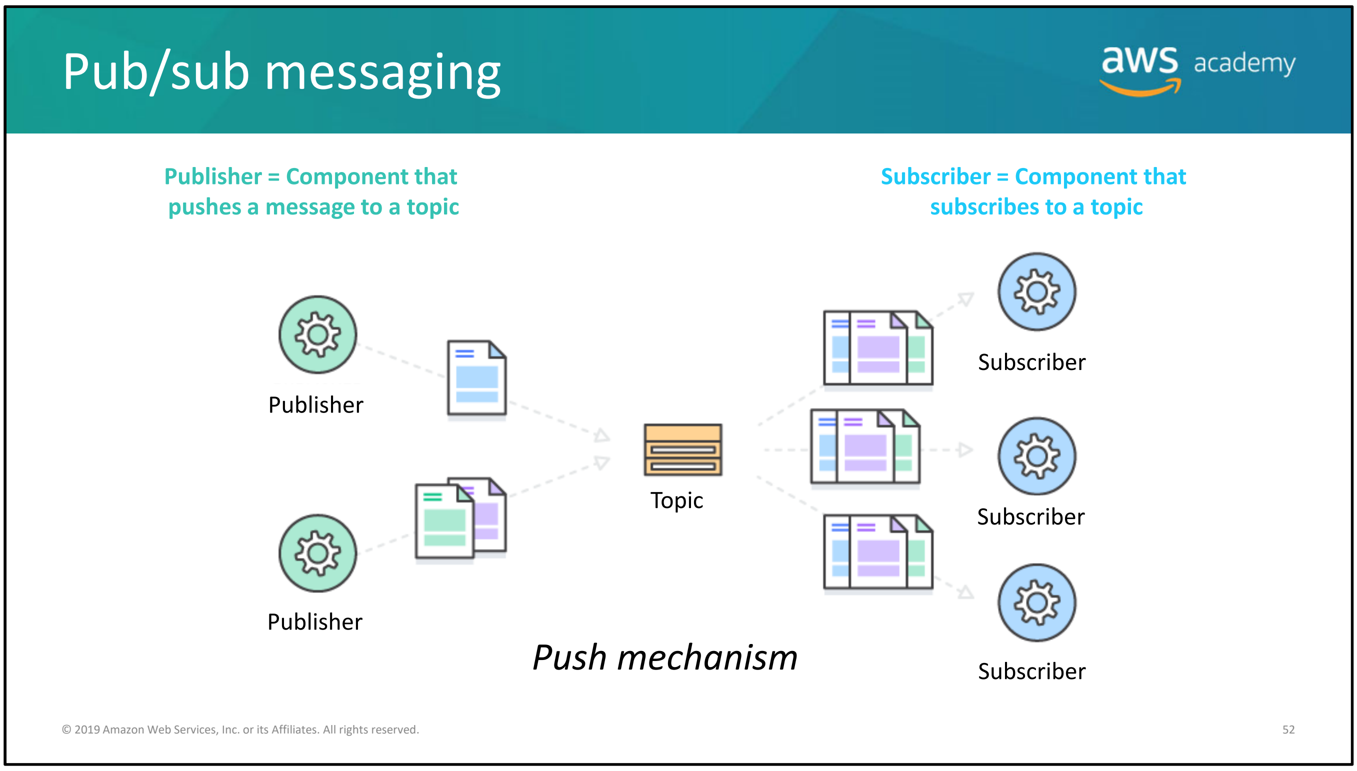
ac1 Slides from Sun
Alan Colman, 17/01/2013

EDA - Features

- Individually captures unpredictable, asynchronous events occurring in parallel
- Senses real-time events and conditions in business environment/databases
- Initiates appropriate response, action, or process
- Modifies processes in real-time for optimal response to changing conditions

This week – Decoupled Architectures

- ☐Loose coupling
- ☐Message Driven Architectures
  - ☐Message Queues
  - ☐MoM middleware, JMS, ActiveMQ
  - ☐AWS SQS (Simple Message Queue)
- ☐Event Driven Architectures
  - ☐Events and Complex Event Processing
  - ☐**Publish Subscribe (pub-sub) Systems**
  - ☐AWS SNS (Simple Notification Service)
  - ☐ Serverless and event- driven applications



Pub/sub messaging

aws academy
Publisher = Component that pushes a message to a topic
Subscriber = Component that subscribes to a topic
Publisher
Publisher
Topic
Subscriber
Subscriber
Subscriber
Push mechanism

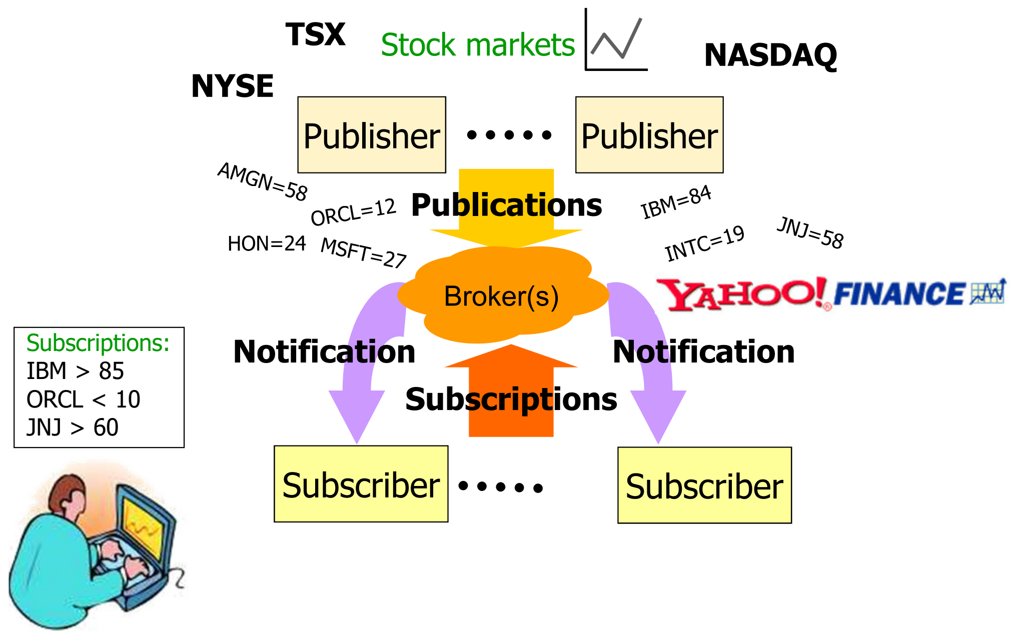
As mentioned earlier, in modern cloud architecture, applications are decoupled into smaller, independent building blocks that are easier to develop, deploy, and maintain. Publish/subscribe (pub/sub) messaging provides instant event notifications for distributed applications.

The pub/sub model allows messages to be broadcast to different parts of a system asynchronously. A message *topic* provides a lightweight mechanism to broadcast asynchronous event notifications, and it also provides endpoints that allow software components to connect to the topic so they can send and receive those messages. To broadcast a message, a component called a *publisher* simply 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 is broadcasting, 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.

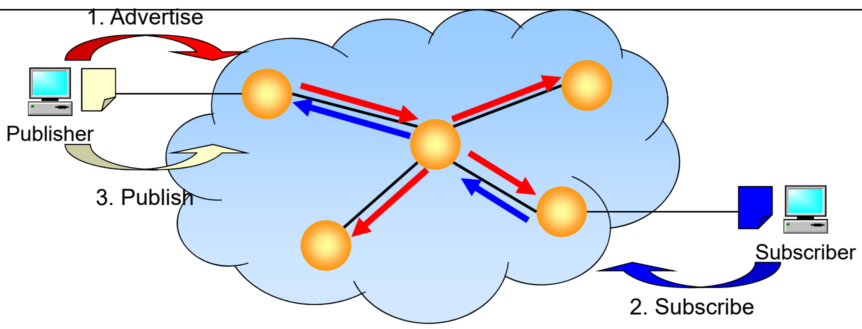
In the pub/sub messaging paradigm, notifications are delivered to clients using a *push mechanism* that eliminates the need to periodically check or poll for new information and updates.

Content-based Publish/Subscribe



Source: Hans-Arno Jacobsen

Content-based Message Routing



A: [class, =, stock], [name, =, HP], [price, >, 50]
S: [class, =, stock], [name, =, *], [price, >, 50]
P: [class, stock], [name, HP], [price, 55]

Source: Hans-Arno Jacobsen

Event-Based	Content Routing	Flexible
Decoupled	Declarative	Responsive

Publish/Subscribe Model

- Publish/subscribe is a communication paradigm in which the interaction between the information producer (*publisher*) and consumer (*subscriber*) is mediated by a set of *brokers*.
- Publishers *publish* events (aka *publications*) to the broker network, and subscribers *subscribe* to interesting events by submitting *subscriptions* to the broker network. It is the responsibility of the brokers to route each event to interested subscribers.
- In *content-based* publish/subscribe, subscribers can specify constraints on the content of the events, and the broker network is said to perform *content-based routing* of events.

The routing in the PADRES distributed content-based publish/subscribe system works as follows.

0) Publishers and subscribers connect to one of a set of brokers.
1) Publishers specify a template of their event space by submitting an advertisement message that is flooded through the broker network and creates a spanning tree rooted at the publisher.
2) Similarly, subscribers specify their interest by sending a subscription message that is forwarded along the reverse paths of intersecting advertisements, i.e., those with potentially interesting events.
3) Now publications from publishers are forwarded along the reverse paths of matching subscriptions to interested subscribers.

Publish/Subscribe benefits

- It is event-based, so it is a natural fit for the event-driven applications in the previous slide.

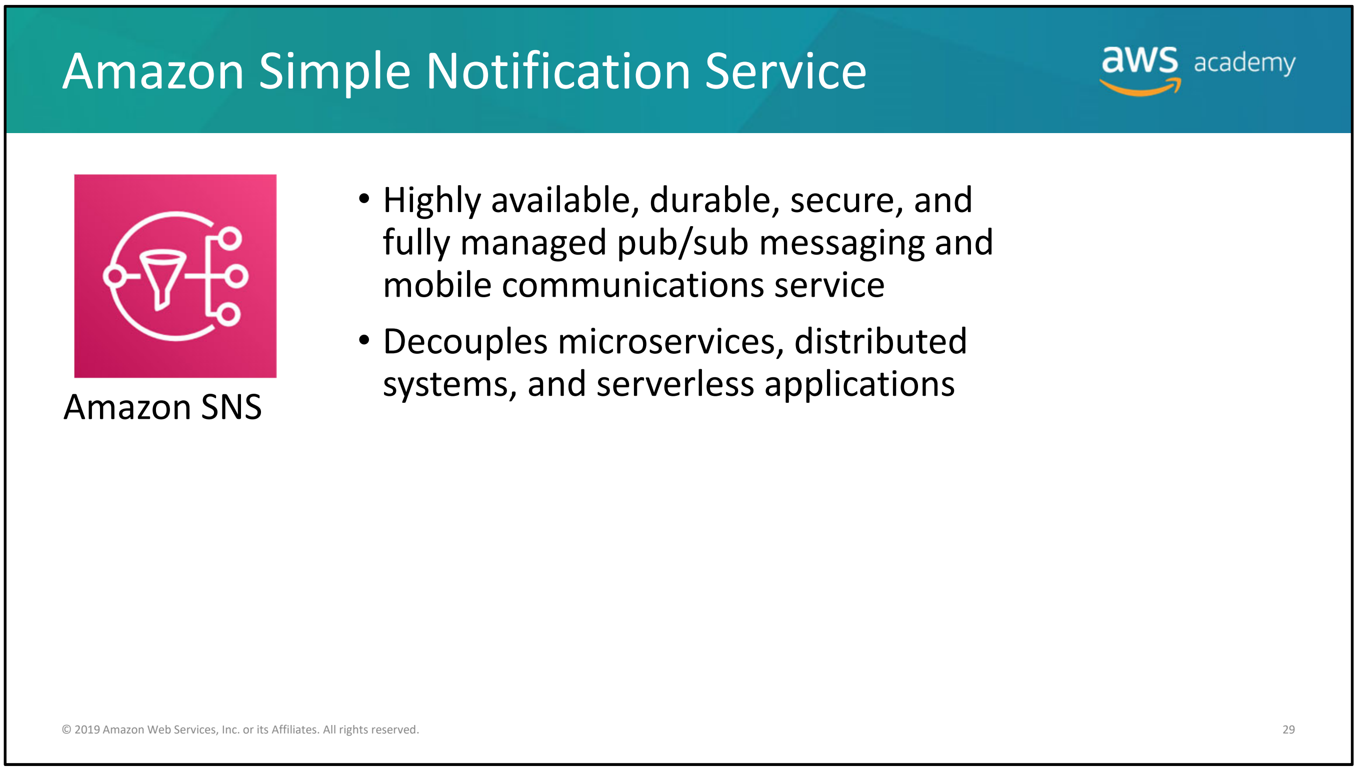
- It decouples clients through a declarative publish/subscribe interface.
- It performs content-based routing to support expressive queries.
- Interactions and queries are declarative (say *what* you want, not where/who to get it from).
- Content-based addresses allow for flexible deployment environments.
- It is responsive since events are pushed to clients immediately.

Conclusion: pub/sub is a sound model for these event-driven enterprise applications.

This week – Decoupled Architectures

- ☐Loose coupling
- ☐Message Driven Architectures
  - ☐ Message Queues
  - ☐ MoM middleware, JMS, ActiveMQ
  - ☐ AWS SQS (Simple Message Queue)
- ☐Event Driven Architectures
  - ☐ Events and Complex Event Processing
  - ☐ Publish Subscribe (pub-sub) Systems
  - ☐ **AWS SNS (Simple Notification Service)**
  - ☐ Serverless and event- driven applications

SWINBURNE UNIVERSITY OF TECHNOLOGY



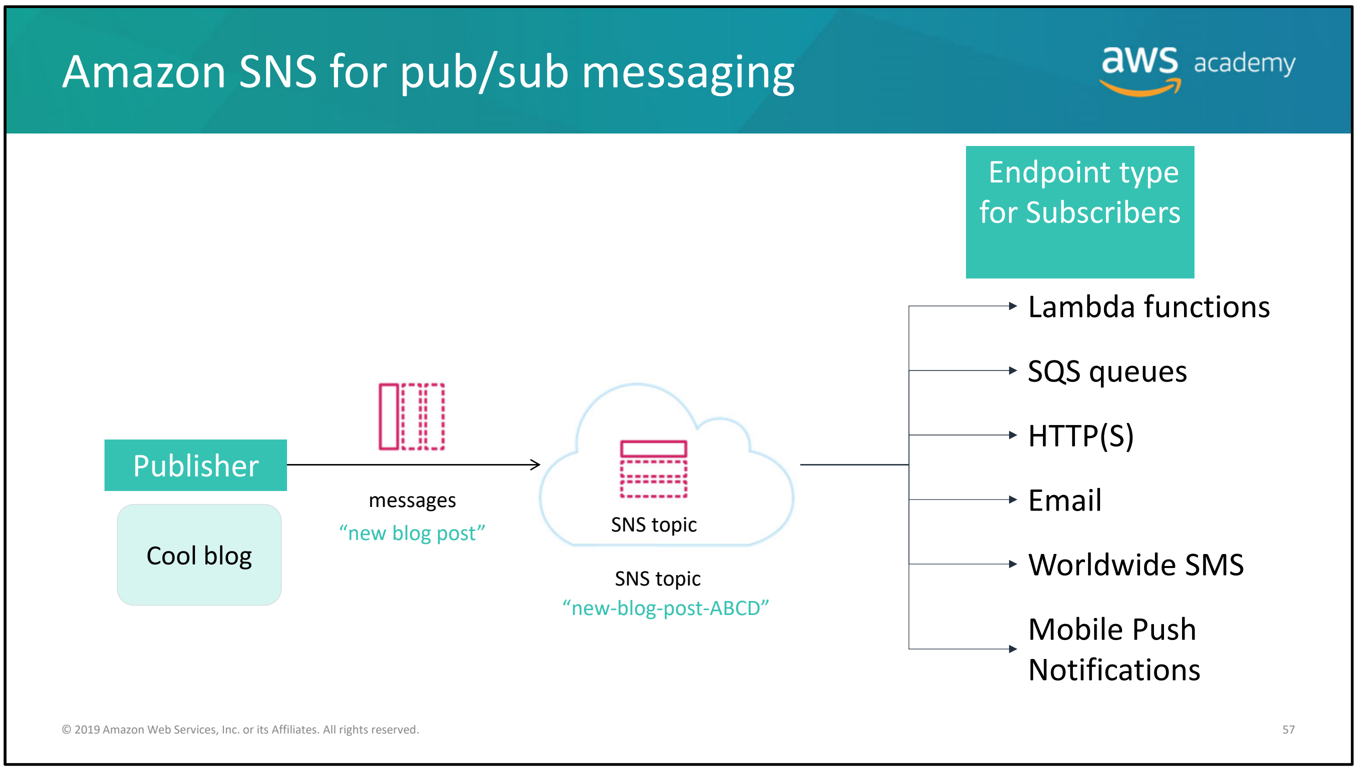
Amazon Simple Notification Service

Amazon SNS

- Highly available, durable, secure, and fully managed pub/sub messaging and mobile communications service
- Decouples microservices, distributed systems, and serverless applications

Amazon Simple Notification Service (Amazon SNS) is a highly available, durable, secure, fully managed pub/sub messaging service that enables you to decouple microservices, distributed systems, and serverless applications. Amazon SNS provides topics for high-throughput, push-based, many-to-many messaging.

For more information, see the Amazon SNS product page: https://aws.amazon.com/sns/



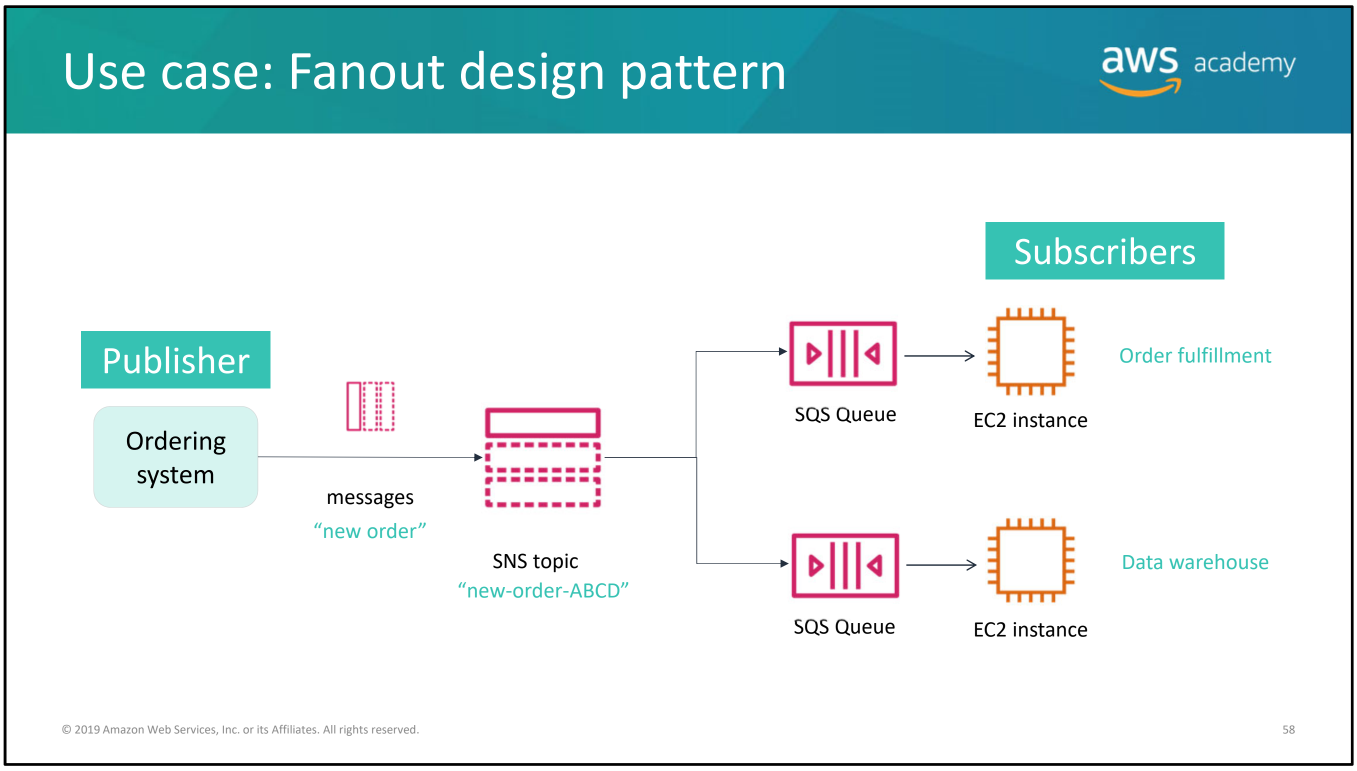
Amazon SNS for pub/sub messaging

Amazon SNS coordinates and manages the delivery or sending of messages to subscribing endpoints or clients. In Amazon SNS, there are two types of clients—publishers and subscribers. Publishers communicate asynchronously with subscribers by producing and sending a message to a topic, which is a logical access point and communication channel. Subscribers consume or receive the message or notification over one of the supported protocols. Amazon SNS supports notification deliveries to the following endpoint types: Lambda functions, SQS queues, HTTP(S), email, worldwide short message service (SMS), and mobile push notifications.

When you create an Amazon SNS topic, you can control access to it by defining policies that determine which publishers and subscribers can communicate with the topic. A publisher can send messages to topics they have created or to topics they have permission to publish to. Instead of including a specific destination address in each message, a publisher sends a message to the topic. Amazon SNS matches the topic to a list of subscribers who have subscribed to that topic. It then 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.

See the AWS Documentation for more information on using Amazon SNS for system-to-system messaging with:

- Amazon SQS queue as a subscriber: https://docs.aws.amazon.com/sns/latest/dg/sns-sqs-as-subscriber.html
- AWS Lambda function as a subscriber: https://docs.aws.amazon.com/sns/latest/dg/sns-lambda-as-subscriber.html
- HTTP(S) as a subscriber: https://docs.aws.amazon.com/sns/latest/dg/sns-http-https-endpoint-as-subscriber.html

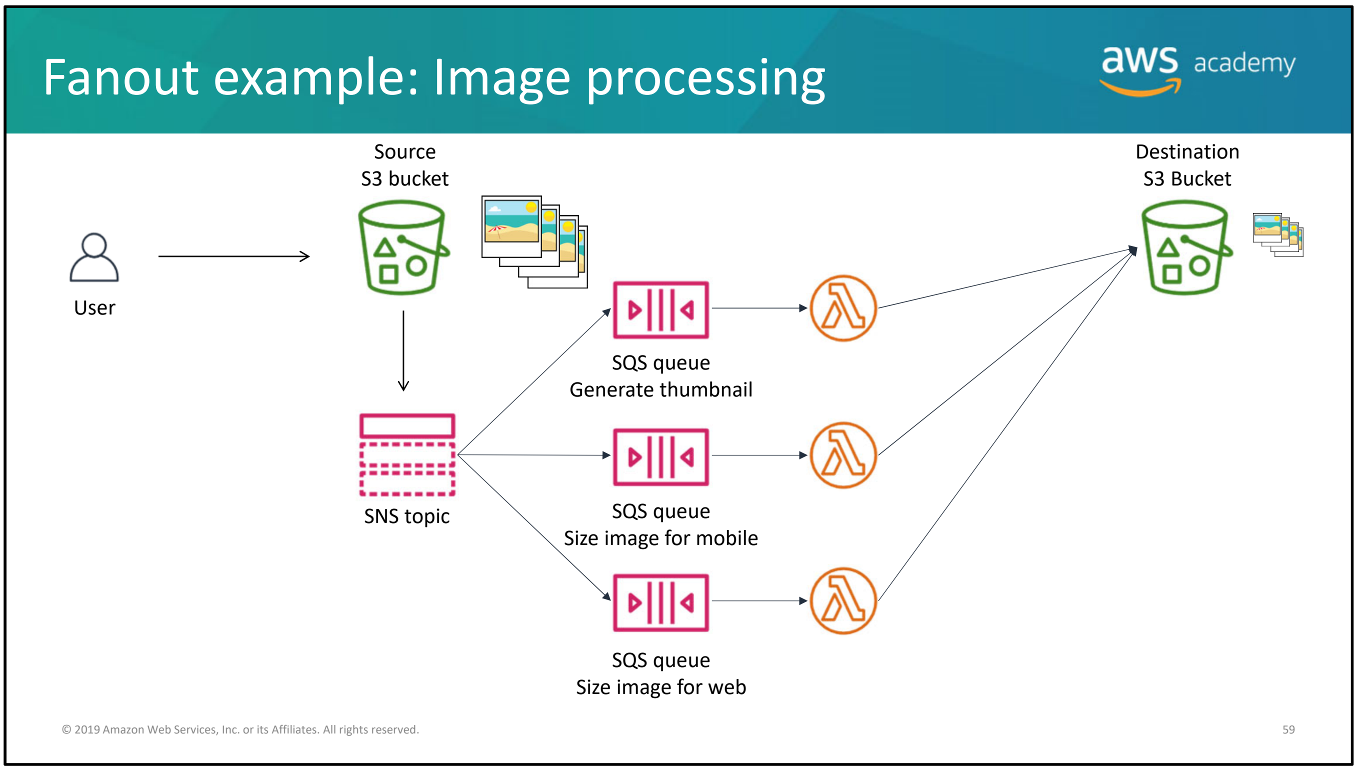


Use case: Fanout design pattern

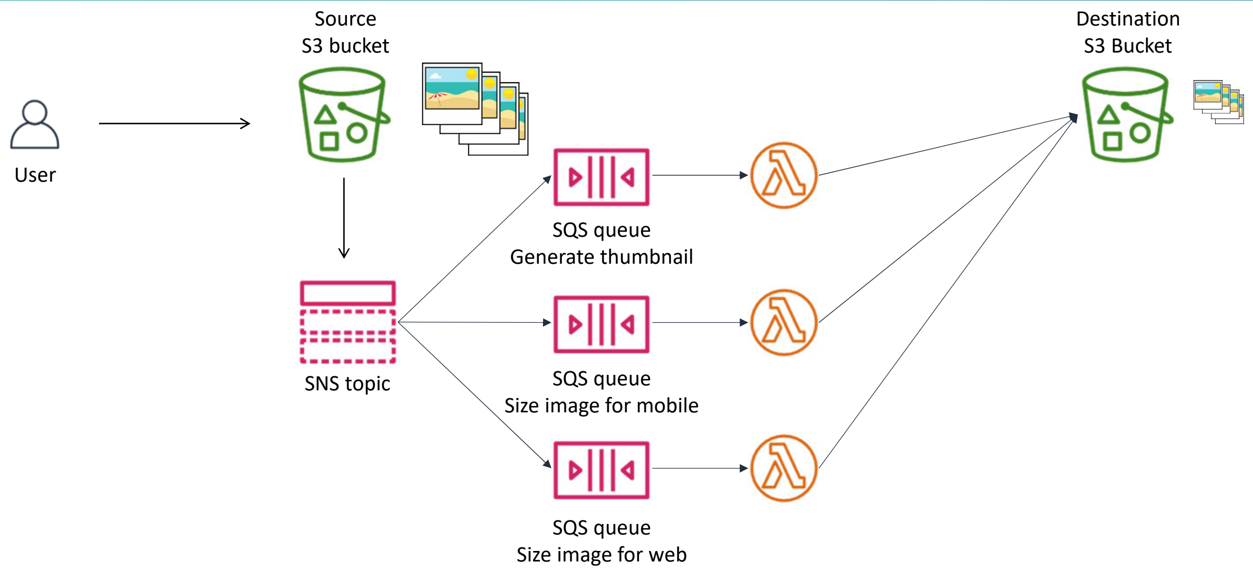
When a message must be processed by more than one consumer, you can combine a message queue with pub/sub messaging in a *fanout design pattern*. *Fanout* is when an Amazon SNS message is sent to a topic and then replicated and pushed to multiple Amazon SQS queues, HTTP endpoints, or email addresses. This allows for parallel asynchronous processing.

For example, you could develop an application that sends an Amazon SNS message to a topic when an order is placed for a product. Then, the Amazon SQS queues that are subscribed to that topic would receive identical notifications for the new order. The Amazon EC2 server instance that is attached to one of the queues could handle the processing or fulfillment of the order. The other server instance could be attached to a data warehouse for analysis of all orders that are received.

You could also use fanout to replicate data that is sent to your production environment to your development environment. For example, you could subscribe yet another queue to the same topic for new incoming orders. Then, by attaching this new queue to your development environment, you could continue to improve and test your application by using data received from your production environment.

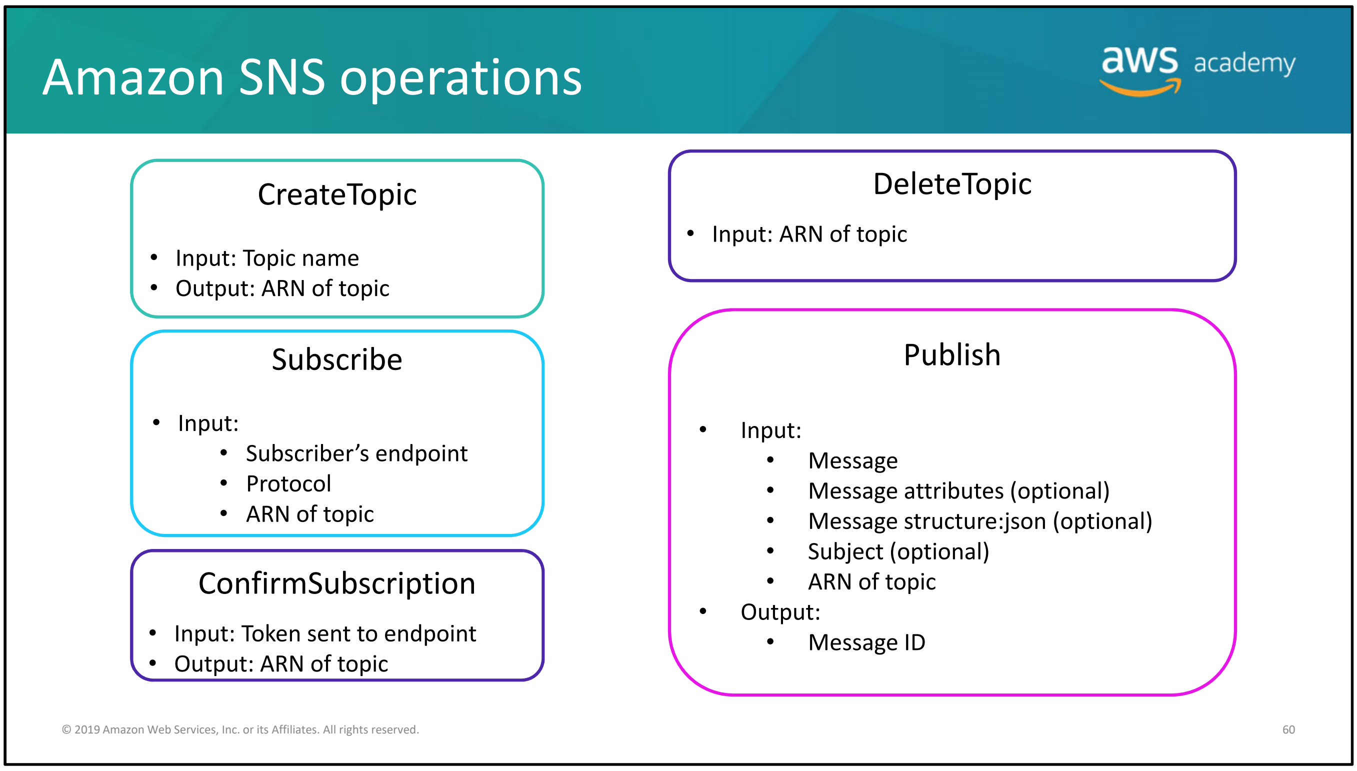


Fanout example: Image processing



Here is an example of how you can use fanout for image processing. In this example, when a user uploads images to an Amazon Simple Storage Service (Amazon S3) bucket for image processing, a message about the event is published to an SNS topic. Multiple SQS queues are subscribed to that topic. When the SQS queues receive the message, they each invoke a Lambda function with the payload of the published message. The Lambda functions process the image (for example, generate thumbnail images, size images for mobile applications, or size images for web applications) and send the results to another S3 bucket.

For more information on fanout and other common Amazon SNS scenarios, see the AWS Documentation: https://docs.aws.amazon.com/sns/latest/dg/sns-common-scenarios.html



Amazon SNS operations

CreateTopic

- Input: Topic name
- Output: ARN of topic

Subscribe

- Input:
  - Subscriber’s endpoint
  - Protocol
  - ARN of topic

ConfirmSubscription

- Input: Token sent to endpoint
- Output: ARN of topic

DeleteTopic

- Input: ARN of topic

Publish

- Input:
  - Message
  - Message attributes (optional)
  - Message structure:json (optional)
  - Subject (optional)
  - ARN of topic
- Output:
  - Message ID

As a developer, there are a few common API calls that you should be familiar with when you work with Amazon SNS:

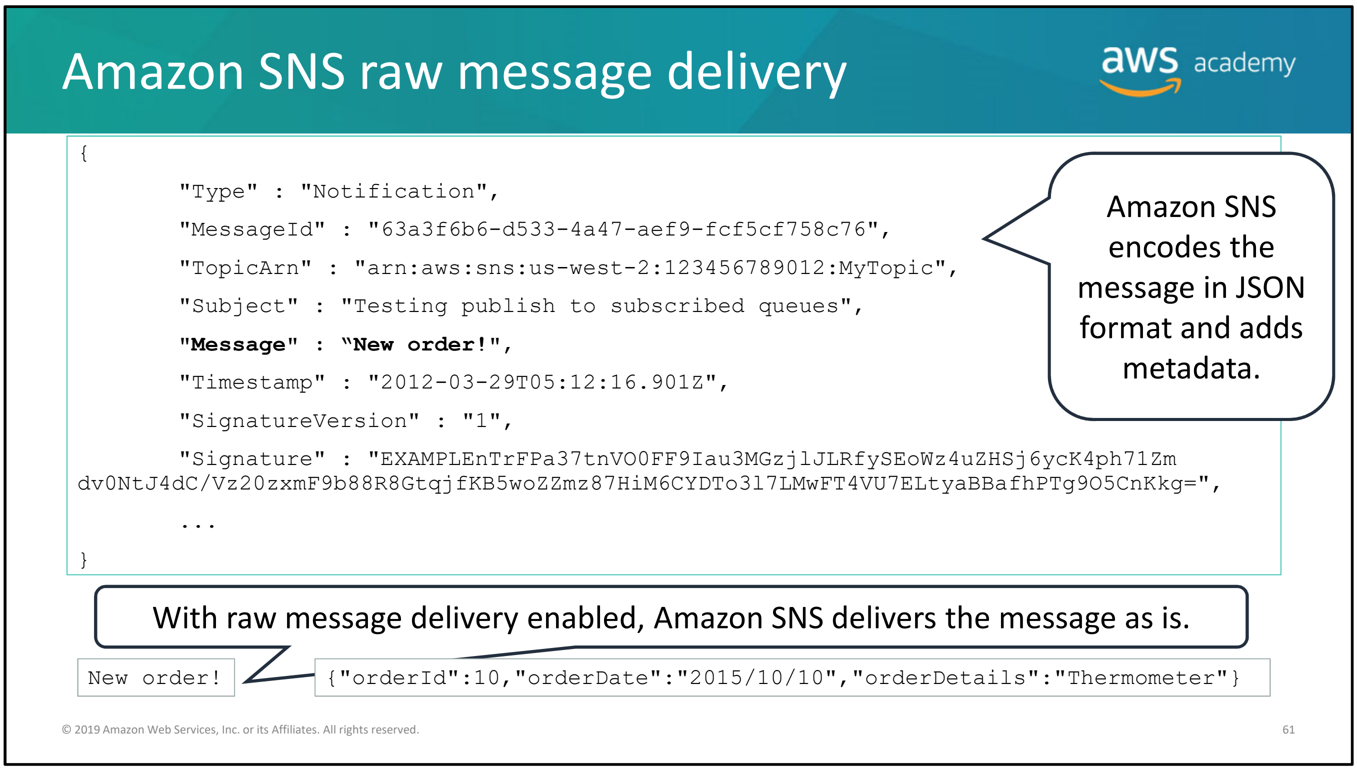
- `CreateTopic` – Creates a topic where notifications can be published. Users can create a maximum of 3,000 topics. If a requester already owns a topic with the specified name, that topic's Amazon Resource Name (ARN) is returned without creating a new topic.
- `Subscribe` – Prepares to subscribe to an endpoint by sending a confirmation message to the endpoint. If the service was able to create a subscription immediately (without requiring endpoint owner confirmation), the response of the `Subscribe` request includes the ARN of the subscription. To actually create a subscription, the endpoint owner must call the `ConfirmSubscription` action with the token from the confirmation message.
- `ConfirmSubscription` – Verifies an endpoint owner's intent to receive messages by validating the token that was sent to the endpoint by an earlier `Subscribe` action. If the token is valid, the action creates a new subscription and returns its ARN.
- `DeleteTopic` – Deletes a topic and all its subscriptions. Deleting a topic might prevent some messages that were previously sent to the topic from being delivered to

subscribers.

- *`Publish`* – Sends a message to all of a topic's subscribed endpoints. When a `message ID` is returned, the message has been saved and Amazon SNS will attempt to deliver it to the topic's subscribers.

See the AWS Documentation for more information about:

- Amazon SNS API operations: https://docs.aws.amazon.com/sns/latest/api/Welcome.html
- How to work with Amazon SNS using the AWS Management Console, the AWS software development kit (SDK) for Java, and the AWS SDK for .NET: https://docs.aws.amazon.com/sns/latest/dg/sns-tutorials.html



Amazon SNS raw message delivery

```
{
        "Type" : "Notification",
        "MessageId" : "63a3f6b6-d533-4a47-aef9-fcf5cf758c76",
        "TopicArn" : "arn:aws:sns:us-west-2:123456789012:MyTopic",
        "Subject" : "Testing publish to subscribed queues",
        "Message" : “New order!",
        "Timestamp" : "2012-03-29T05:12:16.901Z",
        "SignatureVersion" : "1",
        "Signature" : "EXAMPLEnTrFPa37tnVO0FF9Iau3MGzjlJLRfySEoWz4uZHSj6ycK4ph71Zm
dv0NtJ4dC/Vz20zxmF9b88R8GtqjfKB5woZZmz87HiM6CYDTo3l7LMwFT4VU7ELtyaBBafhPTg9O5CnKkg=",
        ...
}
```

Amazon SNS encodes the message in JSON format and adds metadata.

With raw message delivery enabled, Amazon SNS delivers the message as is.

```
New order!
```

```
{"orderId":10,"orderDate":"2015/10/10","orderDetails":"Thermometer"}
```

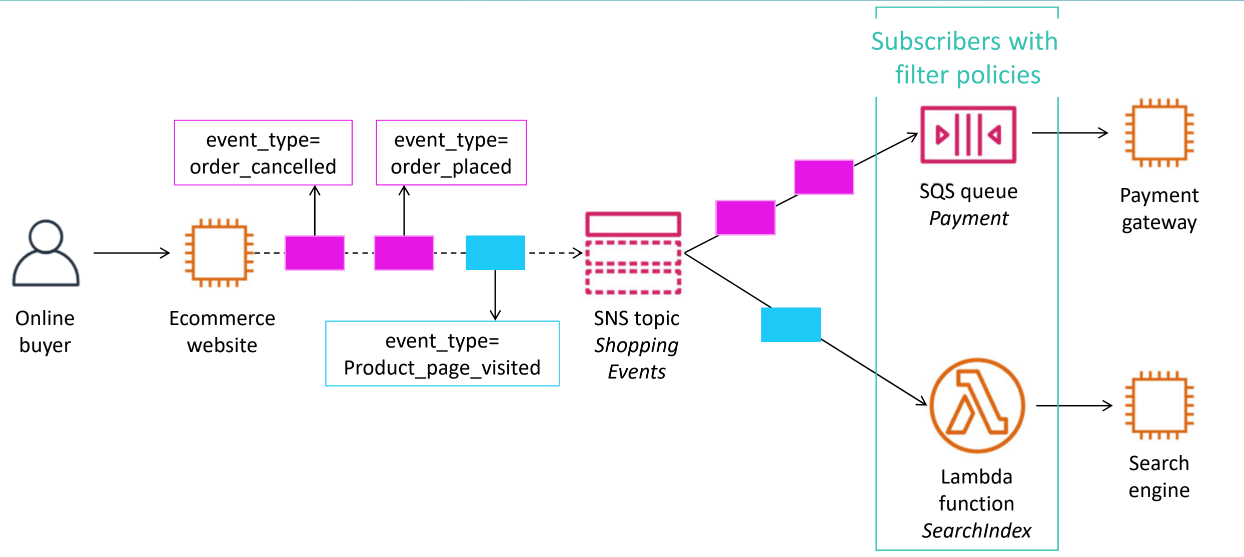
Except for SMS messages, Amazon SNS messages can contain up to 256 KB of text data, including Extensible Markup Language (XML), JavaScript Object Notation (JSON), and unformatted text. By default, when Amazon SNS sends a message to an Amazon SQS queue, it encodes the message as a JSON document that contains the message and metadata about the message. Enabling raw message delivery reduces the need for endpoints to process JSON formatting. For example, when you enable raw message delivery for an Amazon SQS endpoint, the Amazon SNS metadata is not included and the published message is delivered to the subscribed Amazon SQS endpoint as-is. In the *New order* example, the publisher sent a JSON-formatted message to an Amazon SNS topic. Amazon SNS then published the message as-is to subscribers without adding Amazon SNS metadata.

See the AWS Documentation for more information on:

- Sending Amazon SNS messages to Amazon SQS queues: https://docs.aws.amazon.com/sns/latest/dg/sns-sqs-as-subscriber.html
- Raw message delivery: https://docs.aws.amazon.com/sns/latest/dg/sns-large-payload-

raw-message-delivery.html

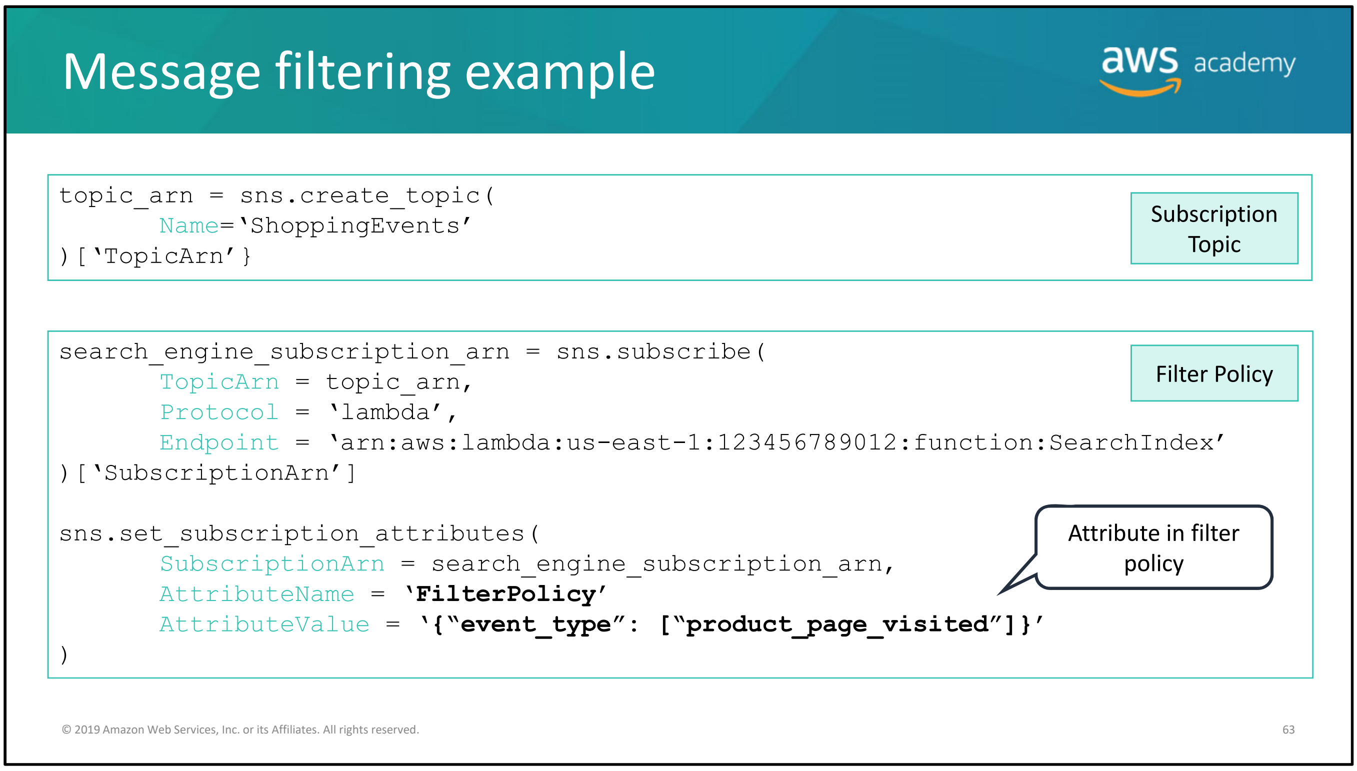
Filter policies


Subscribers with filter policies
event_type= order_cancelled
event_type= order_placed
Online buyer
Ecommerce website
event_type= Product_page_visited
SNS topic *Shopping Events*
SQS queue *Payment*
Payment gateway
Lambda function *SearchIndex*
Search engine

By default, an Amazon SNS topic subscriber receives every message that is published to the topic. To receive a subset of the messages, a subscriber must assign a *filter policy* to the topic subscription. A filter policy is a simple JSON object that contains attributes that define which messages the subscriber receives. When you publish a message to a topic, Amazon SNS compares the message attributes to the attributes in the filter policy for each of the topic's subscriptions. If any of the attributes match, Amazon SNS sends the message to the subscriber. Otherwise, Amazon SNS skips the subscriber without sending the message. If a subscription doesn't have a filter policy, the subscription receives every message that is published to its topic.

In this example, an online buyer visits a website, cancels one order, and places another order. These messages are published to the SNS topic *Shopping Events*. The *Payment* SQS queue subscriber has a filter policy applied so it receives only the messages about the orders. The Lambda function subscriber has a filter policy to receive only the message that the product page was visited.

To learn more about Amazon SNS message filtering, see the AWS Documentation: https://docs.aws.amazon.com/sns/latest/dg/sns-message-filtering.html



Message filtering example

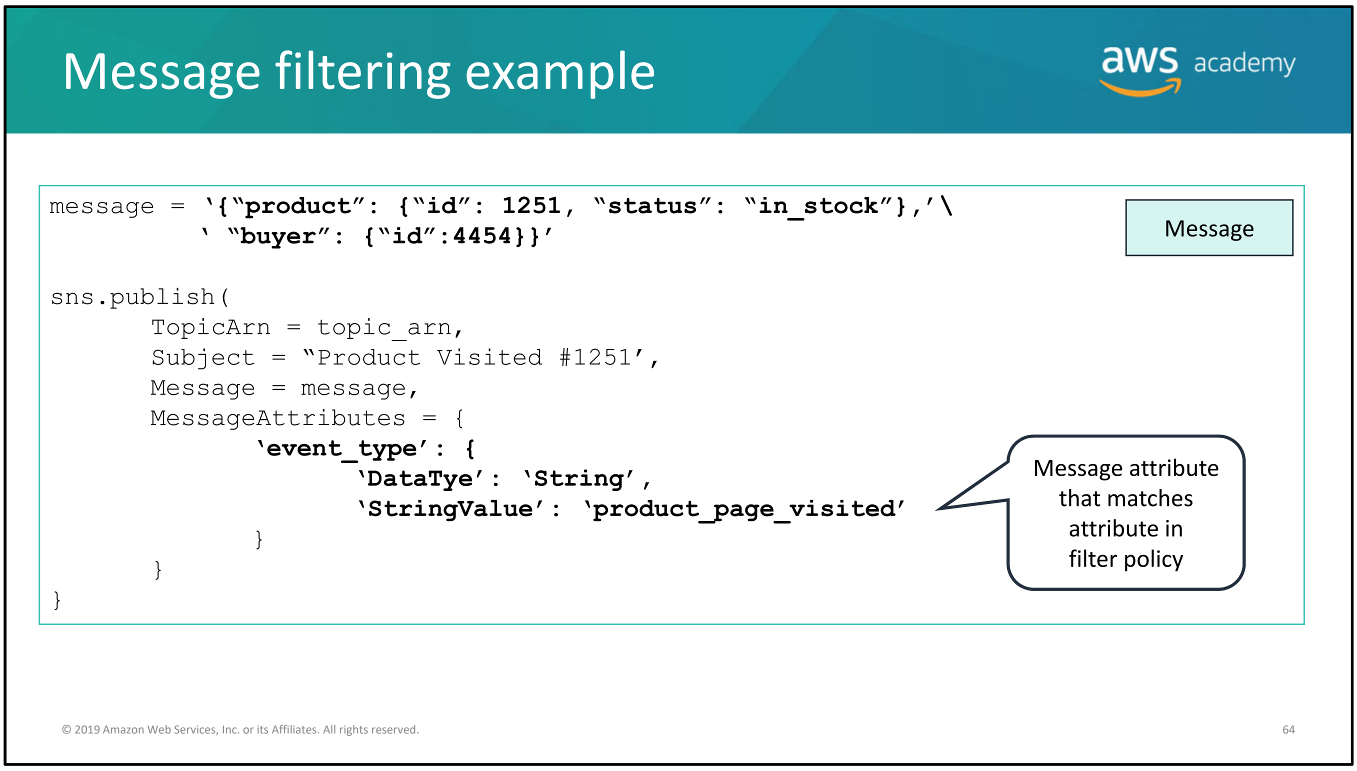
```
topic_arn = sns.create_topic(
        Name='ShoppingEvents'
)['TopicArn']}
```

Subscription Topic

```
search_engine_subscription_arn = sns.subscribe(
        TopicArn = topic_arn,
        Protocol = 'lambda',
        Endpoint = 'arn:aws:lambda:us-east-1:123456789012:function:SearchIndex'
)['SubscriptionArn']

sns.set_subscription_attributes(
        SubscriptionArn = search_engine_subscription_arn,
        AttributeName = 'FilterPolicy'
        AttributeValue = '{"event_type": ["product_page_visited"]}'
)
```

A subscription filter policy allows you to specify attribute names and assign a list of values to each attribute name. To continue the ecommerce example, the Lambda function *SearchIndex* is subscribed to the SNS topic *ShoppingEvents* and a filter policy is attached to it. The filter policy has the attribute "**event_type**" with the value "**product_page_visited**".



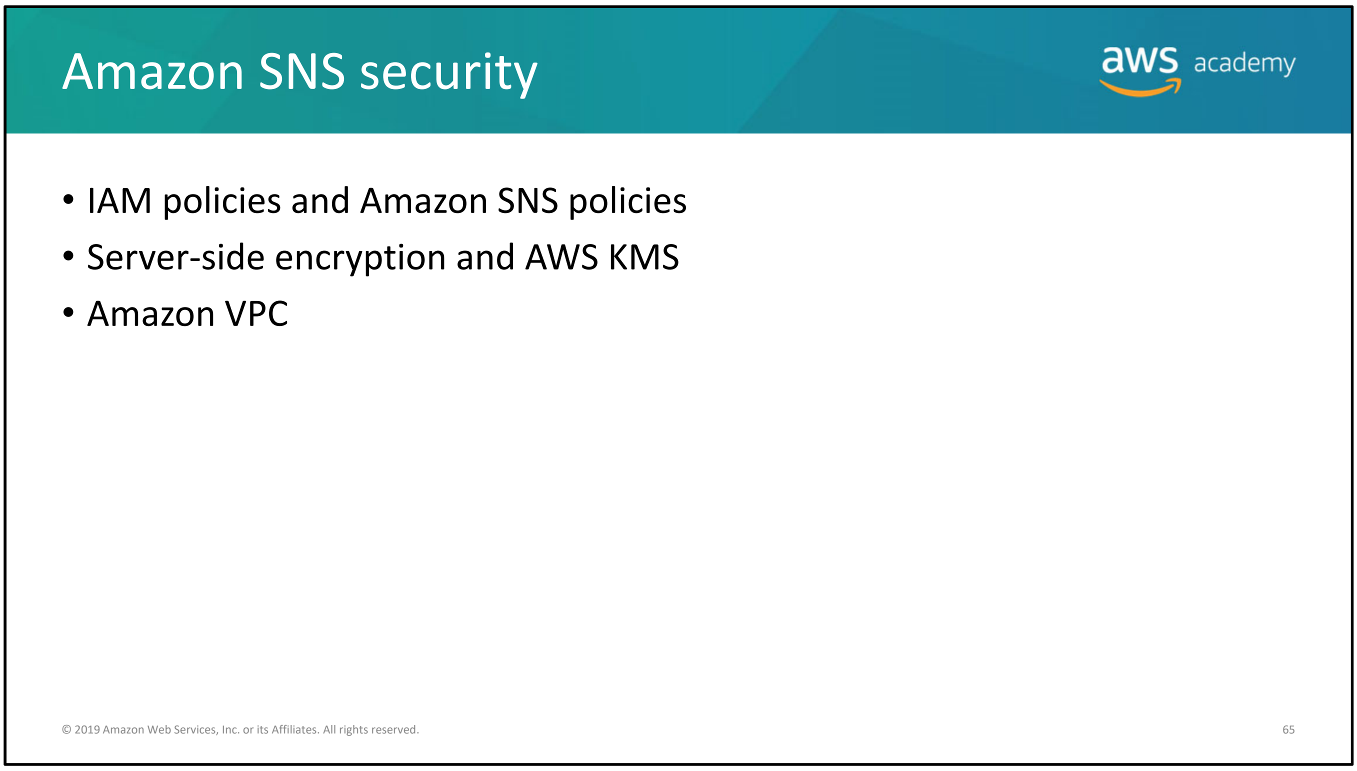
Message filtering example

```
message = '{"product": {"id": 1251, "status": "in_stock"},'\
          ' "buyer": {"id":4454}}'

sns.publish(
       TopicArn = topic_arn,
       Subject = "Product Visited #1251',
       Message = message,
       MessageAttributes = {
              'event_type': {
                     'DataTye': 'String',
                     'StringValue': 'product_page_visited'
              }
       }
}
```

Message
Message attribute that matches attribute in filter policy

When you publish a message to a topic, Amazon SNS compares the message attributes to the attributes in the filter policy for each of the topic's subscriptions. If any of the attributes match, Amazon SNS sends the message to the subscriber. In the ecommerce example, the message attribute "**event_type**" with the value "**product_page_visited**" matches only the filter policy that is associated with the search engine subscription. Therefore, only the Lambda function that is subscribed to the SNS topic is notified about this navigation event, and the *Payment* SQS queue is not notified.



Amazon SNS security

- IAM policies and Amazon SNS policies
- Server-side encryption and AWS KMS
- Amazon VPC

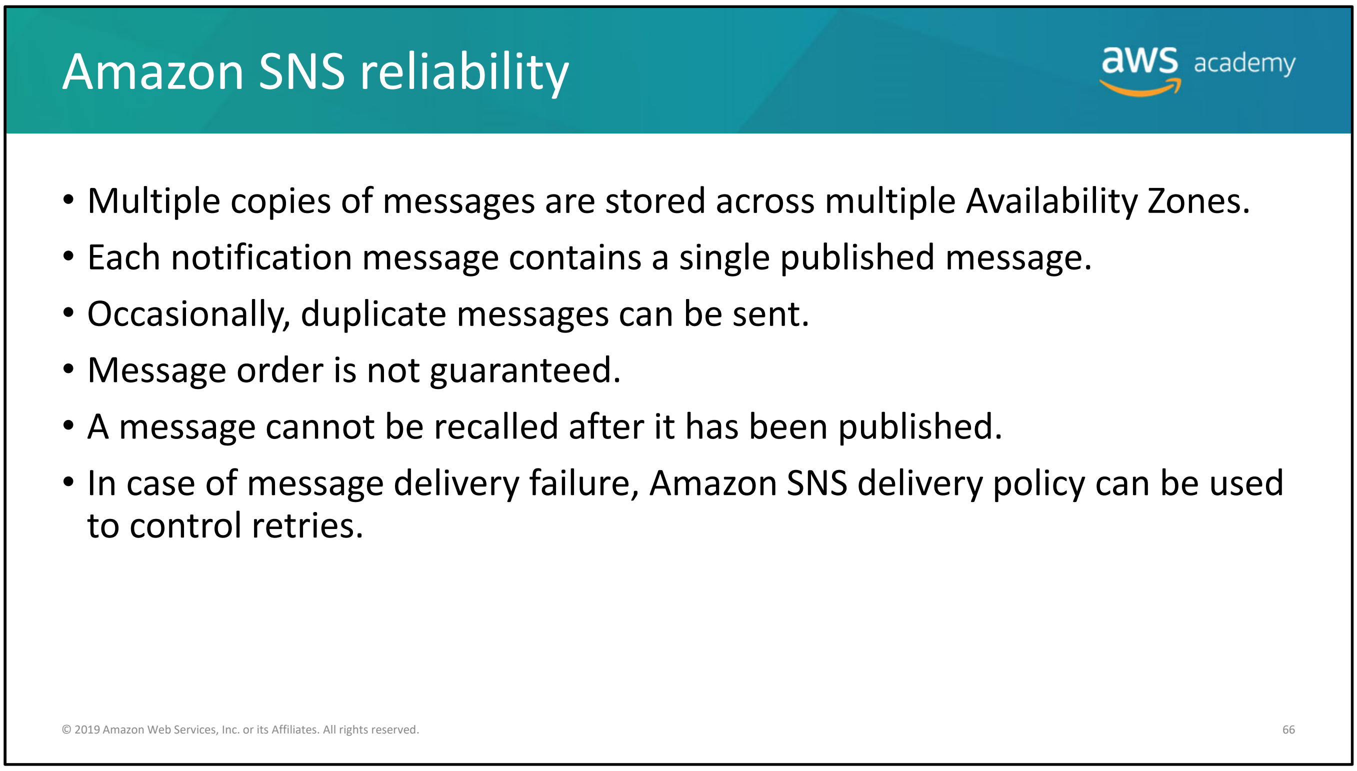
You have detailed control over which endpoints a topic allows, who is able to publish to a topic, and under what conditions.

There are three ways to secure your Amazon SNS resources:

- *AWS Identity and Access Management (IAM) policies and Amazon SNS policies together* – Amazon SNS is integrated with IAM so that you can specify which Amazon SNS actions a user in your AWS account can perform with your Amazon SNS resources. You can specify a particular topic in the policy. For example, when you create an IAM policy, you can use variables that give certain users permission to use the Publish action with specific topics in your AWS account. You can use an *IAM policy* to restrict your users' access to Amazon SNS actions and topics. An IAM policy can restrict access only to users within your AWS account, not to other AWS accounts. You can use an *Amazon SNS policy* with a particular topic to restrict who can work with that topic (for example, who can publish messages to it and who can subscribe to it). Amazon SNS policies can give access to other AWS accounts, or to users within your own AWS account.
- *Server-side encryption (SSE) and AWS Key Management Service (AWS KMS)* – Server-side encryption (SSE) lets you transmit sensitive data in encrypted topics. SSE protects the contents of messages in Amazon SNS topics by using keys that are managed in AWS KMS.
- *Amazon Virtual Private Cloud (Amazon VPC)* – If you use Amazon VPC to host your AWS resources, you can establish a private connection between your VPC and Amazon SNS. With this connection, you can publish messages to your Amazon SNS topics without sending them through the public internet.

For more information about Amazon SNS security, see the AWS Documentation:

https://docs.aws.amazon.com/sns/latest/dg/sns-security.html



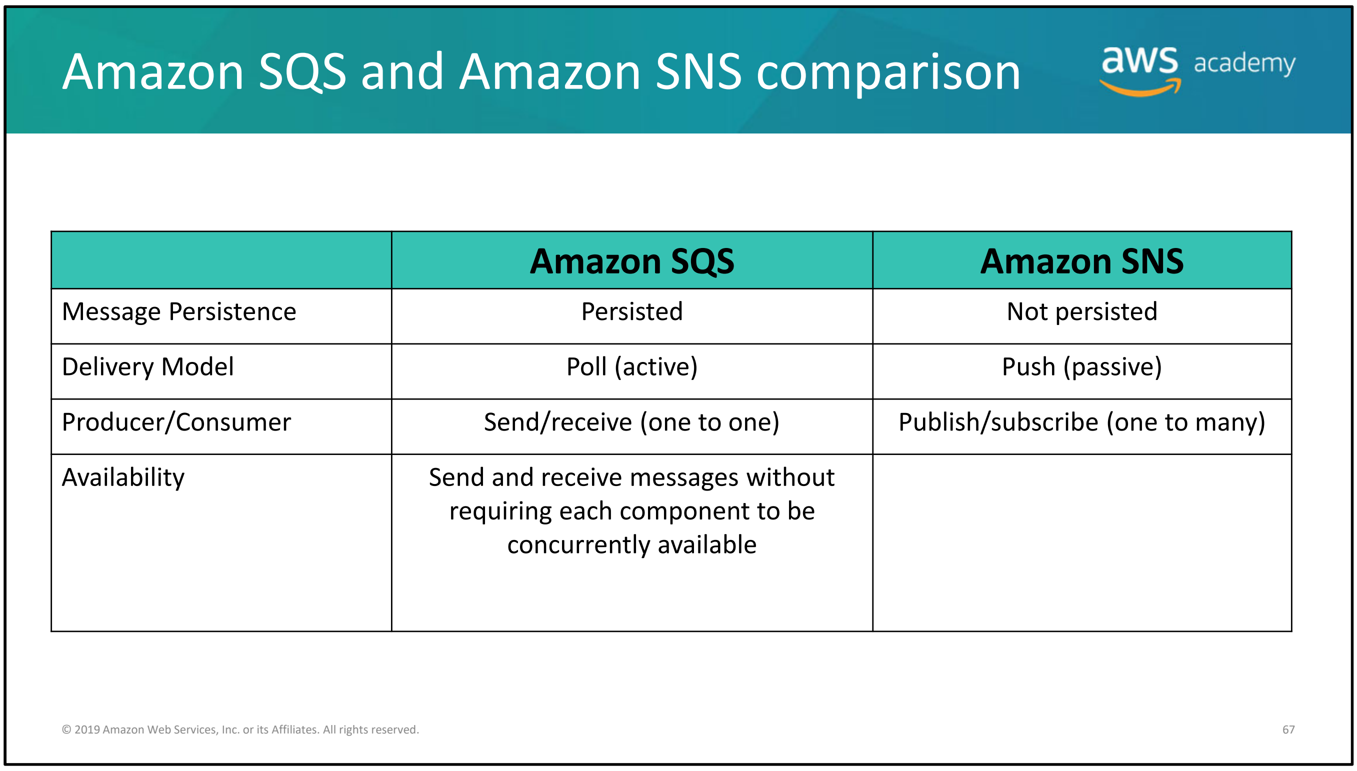
Amazon SNS reliability

- Multiple copies of messages are stored across multiple Availability Zones.
- Each notification message contains a single published message.
- Occasionally, duplicate messages can be sent.
- Message order is not guaranteed.
- A message cannot be recalled after it has been published.
- In case of message delivery failure, Amazon SNS delivery policy can be used to control retries.

Amazon SNS offers reliability in the following ways:

- Amazon SNS provides durable storage of all messages that it receives. When it receives a publish request, Amazon SNS stores multiple copies (to disk) of the message across multiple Availability Zones before it acknowledges receipt of the request to the sender.
- Each notification message contains a single published message.
- Though each message will be delivered to your application exactly once most of the time, the distributed nature of Amazon SNS and transient network conditions could result in occasional, duplicate messages at the subscriber end. You should design your applications so that processing a message more than once does not create any errors or inconsistencies.
- The Amazon SNS service will attempt to deliver messages from the publisher in the order they were published into the topic. However, network issues could potentially result in out-of-order messages at the subscriber end.
- After a message has been successfully published to a topic, it cannot be recalled.
- When a message is published to a topic, Amazon SNS will attempt to deliver notifications to all subscribers that are registered for that topic. Because of potential internet issues or email delivery restrictions, sometimes the notification might not successfully reach an HTTP or email endpoint. With HTTP, an Amazon SNS delivery policy can be used to control retries.

For more information about Amazon SNS, see the FAQ: https://aws.amazon.com/sns/faqs/



Amazon SQS and Amazon SNS comparison

	Amazon SQS	Amazon SNS
Message Persistence	Persisted	Not persisted
Delivery Model	Poll (active)	Push (passive)
Producer/Consumer	Send/receive (one to one)	Publish/subscribe (one to many)
Availability	Send and receive messages without requiring each component to be concurrently available	

Amazon SQS and Amazon SNS are both messaging services within AWS, but they provide different benefits for developers. Amazon SQS is a message queue service that is used by distributed applications to exchange messages through a polling model, and it can be used to decouple sending and receiving components. Amazon SQS provides flexibility for distributed components of applications to send and receive messages without requiring each component to be concurrently available. Amazon SNS allows applications to send time-critical messages to multiple subscribers through a push mechanism, which reduces the need to periodically check or poll for updates.

This week – Decoupled Architectures

- ☐ Loose coupling
- ☐ Message Driven Architectures
  - ☐ Message Queues
  - ☐ MoM middleware, JMS, ActiveMQ
  - ☐ AWS SQS (Simple Message Queue)
- ☐ Event Driven Architectures
  - ☐ Events and Complex Event Processing
  - ☐ Publish Subscribe (pub-sub) Systems
  - ☐ AWS SNS (Simple Notification Service)
  - ☐ **Serverless and event-driven applications**

Serverless and Event Driven Applications

- Serverless ≠ Event driven
- Events can trigger both
  - ☐ Infrastructure changes
  - ☐ Driven application logic
- Examples
  - ☐ Events to working with autoscaling EC2 instances (ACA Lab5)
  - ☐ Processing files written to S3 (ACA Lab 7)

SWINBURNE UNIVERSITY OF TECHNOLOGY

Scenario

- Imagine an order fulfilment process. Suppose we have three services
  - ☐ Order Service
  - ☐ Inventory Service
  - ☐ Shipping Service
- The task is to register the order and make a corresponding reservation of the stock level.
- *How would we implement this using an activity centric Request-Response pattern vs Event-driven pattern?*

Example from http://www.udidahan.com/2008/11/01/soa-eda-and-cep-a-winning-combo/

Example request-response solution

- Centralised orchestration
  1. The “process/application” (e.g. BPEL composite service) sends a message (sync or async) to “registerOrder” on the Order Service.
  2. The “process/application” sends another message (sync or async) to “reserveStock” on the Inventory Service.

Event-driven solution

Order Service

- When the order is submitted and goes through all internal validation, the Order service raises an **OrderTentativelyAccepted** event.

Inventory Service

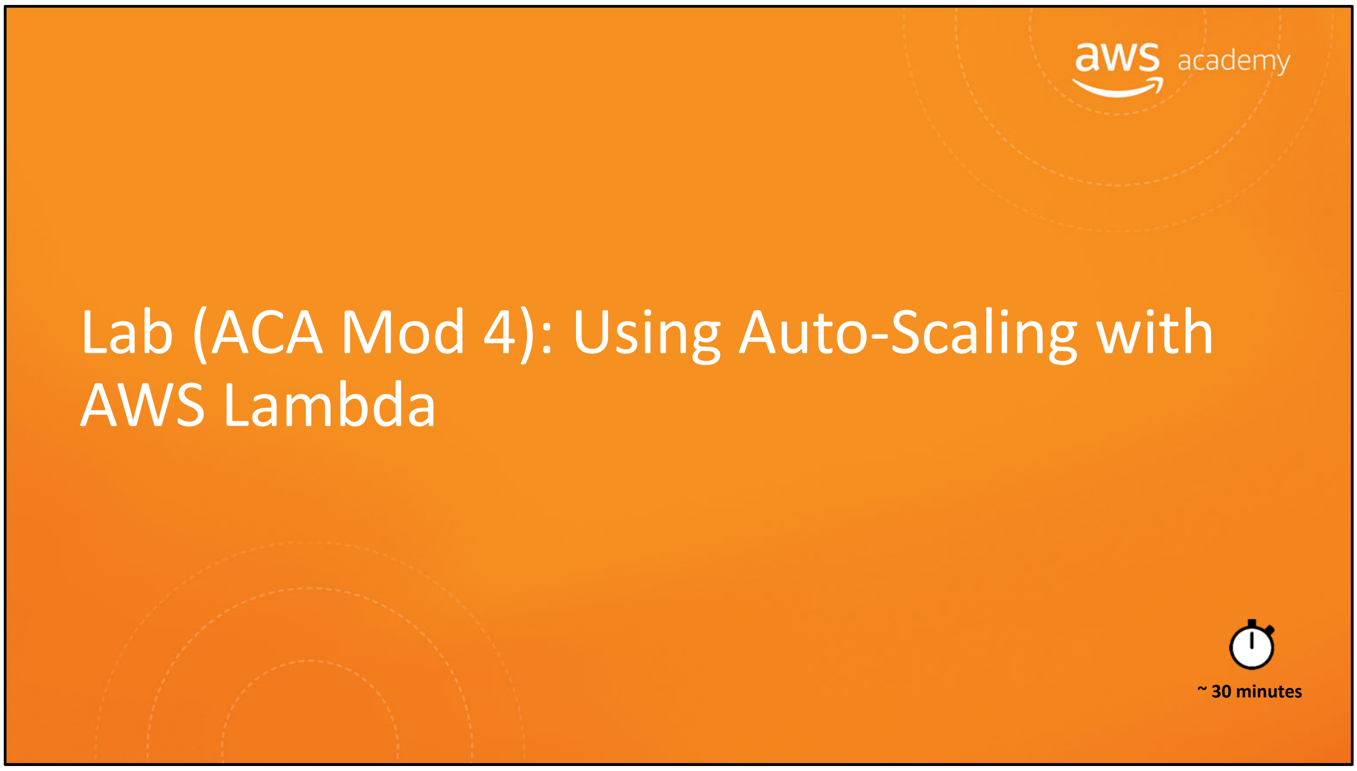
- The Inventory service, which is subscribed to this event, checks if it has everything in stock for the order. For every item in the order on stock, it allocates that stock to the order and publishes the **InventoryAllocatedToOrder** event for it. For items/quantities not in stock, it starts a long running (internal) process which watches for inventory changes.
- When an **InventoryChanged** event occurs, Inventory matches that against orders requiring allocation – if it finds one that requires stock, based on some logic to choose which order gets precedence, it publishes the **InventoryAllocatedToOrder** event.
- The Order service, which is subscribed to the **InventoryAllocatedToOrder** event, upon receiving all events pertaining to the order tentatively accepted, will publish an **OrderAccepted** event.

Extending event-driven solutions

We can now easily extend this process to include shipping the order (remember the other services do not need to know about the shipping service)

Orders and Shipping

- When Inventory receives the **OrderAccepted** event, it generates the pick list to bring all the stock from the warehouses to the loading docks, finally publishing the **PickListGenerated** event containing target docks.
- When Shipping receives the **PickListGenerated** event, it starts the yard management necessary to bring the needed kinds of trucks to the docks.

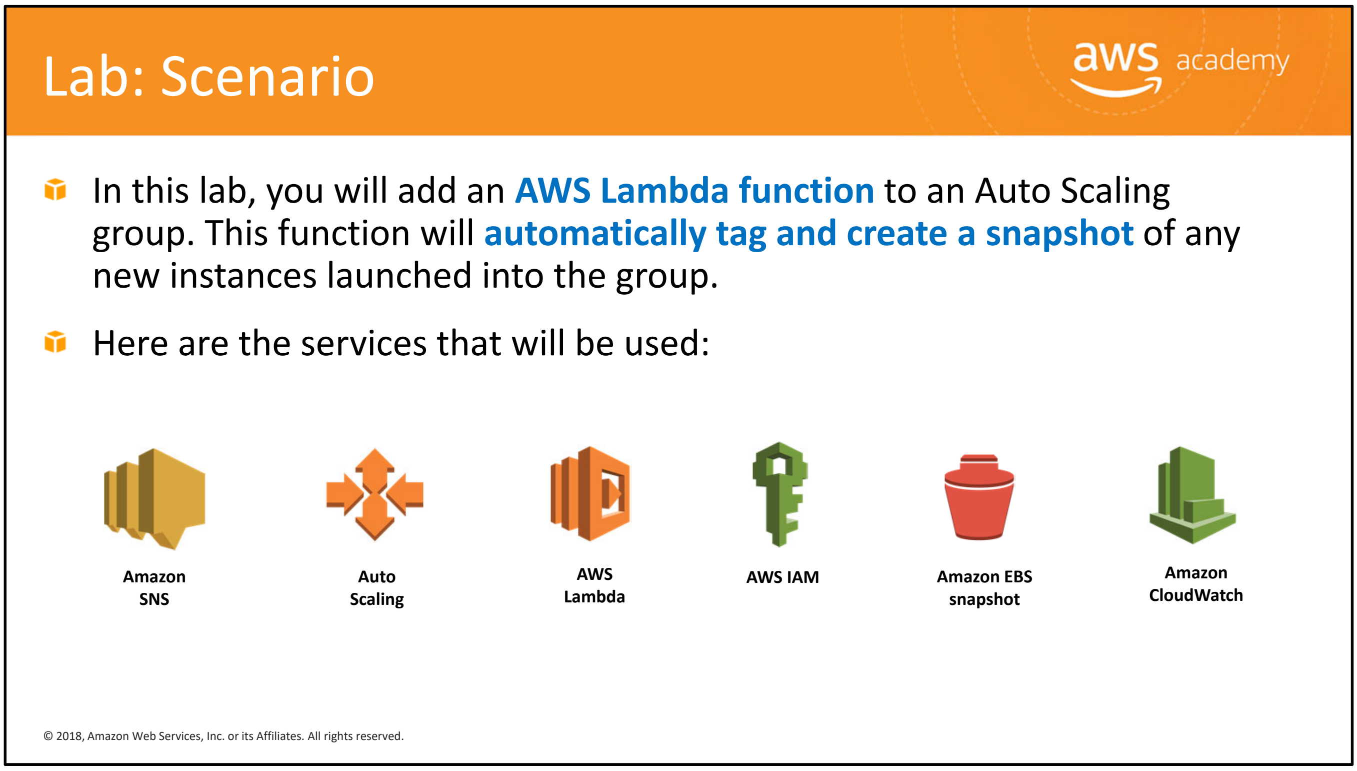


aws academy

Lab (ACA Mod 4): Using Auto-Scaling with AWS Lambda

~ 30 minutes

Now, it's time for a lab that uses Auto Scaling with AWS Lambda. This lab will take approximately 30 minutes.



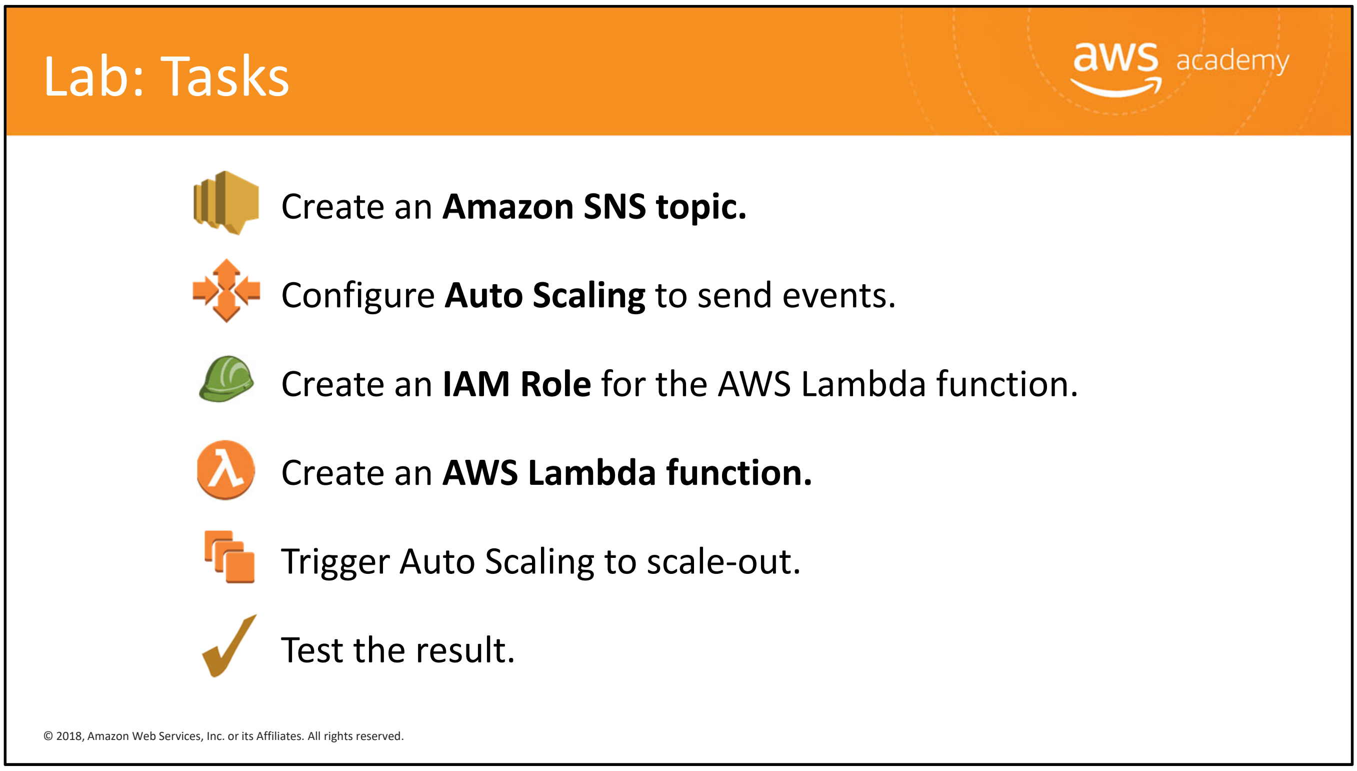
Lab: Scenario

- In this lab, you will add an **AWS Lambda function** to an Auto Scaling group. This function will **automatically tag and create a snapshot** of any new instances launched into the group.
- Here are the services that will be used:

Amazon SNS
Auto Scaling
AWS Lambda
AWS IAM
Amazon EBS snapshot
Amazon CloudWatch

In this lab, you will add an AWS Lambda function to an Auto Scaling group. This function will automatically tag and create a snapshot of any new instances launched into the group.

The services that you will use include Amazon SNS, Auto Scaling, AWS Lambda, IAM, an Amazon EBS snapshot, and CloudWatch.



Lab: Tasks

aws academy

- Create an **Amazon SNS topic.**
- Configure **Auto Scaling** to send events.
- Create an **IAM Role** for the AWS Lambda function.
- Create an **AWS Lambda function.**
- Trigger Auto Scaling to scale-out.
- Test the result.

First, create an **Amazon SNS** topic.

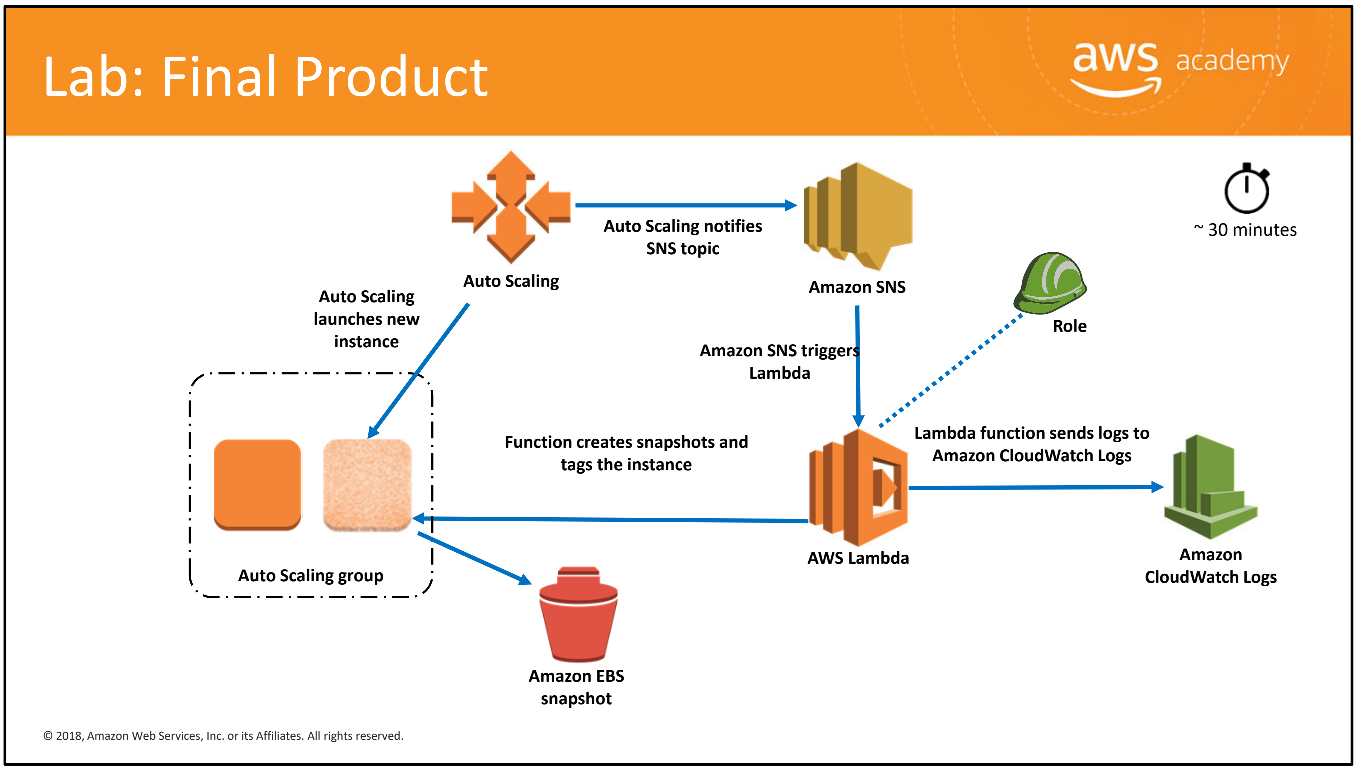
Next, configure **Auto Scaling** to send events.

Next, create an **IAM Role** for the AWS Lambda function.

Create an **AWS Lambda** function.

Trigger Auto Scaling to scale out.

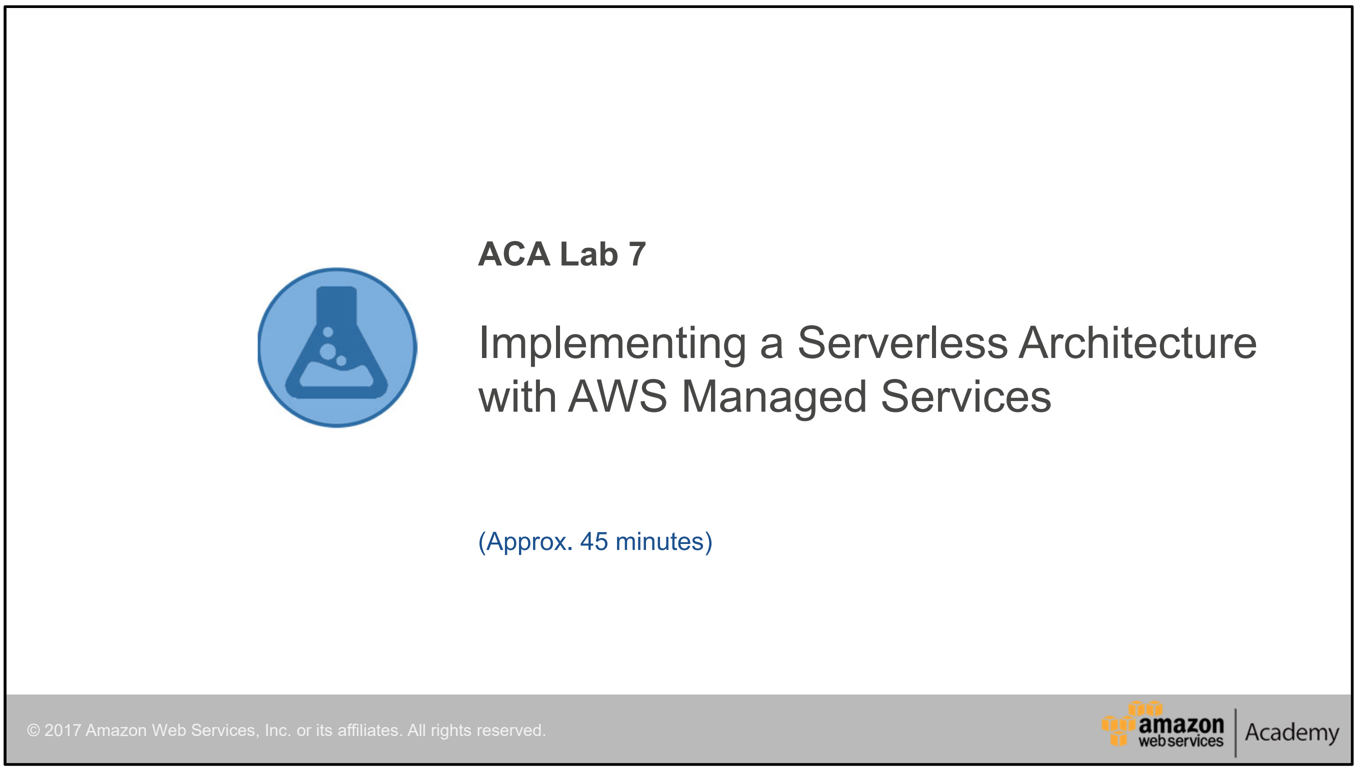
And finally, test the result.



Lab: Final Product

aws academy
~ 30 minutes
Auto Scaling
Auto Scaling notifies SNS topic
Amazon SNS
Role
Auto Scaling launches new instance
Amazon SNS triggers Lambda
Function creates snapshots and tags the instance
Lambda function sends logs to Amazon CloudWatch Logs
AWS Lambda
Amazon CloudWatch Logs
Auto Scaling group
Amazon EBS snapshot

This slide shows the final product for this lab.



ACA Lab 7

Implementing a Serverless Architecture with AWS Managed Services

(Approx. 45 minutes)

amazon web services | Academy

ACA Lab 7: Scenario

In this lab, you will put together a serverless architecture that processes a text file, stores the information intelligently into DynamoDB tables, and sends notifications based on a condition within one of the tables.

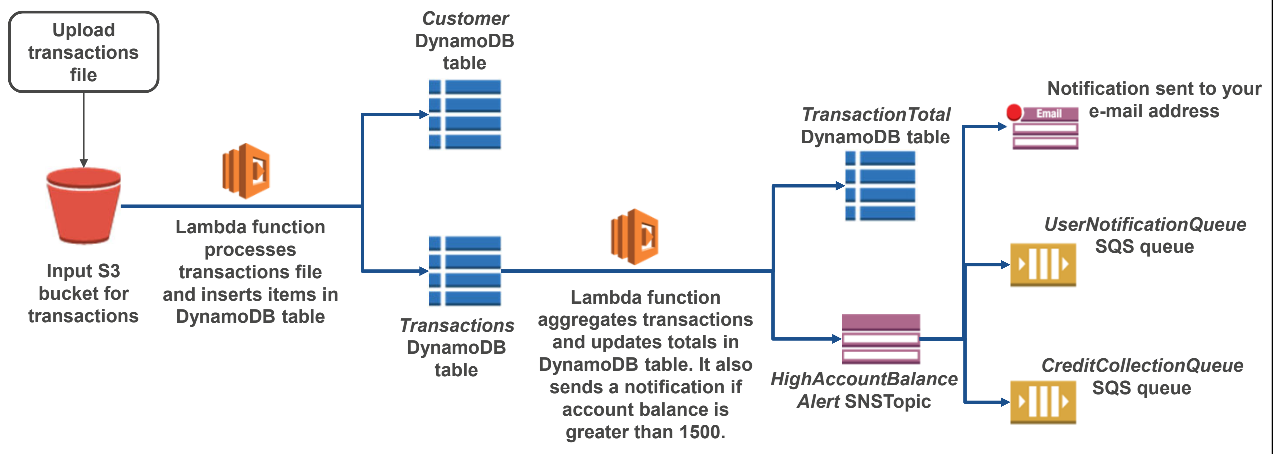
Here are the services that will be used:

Amazon SNS   Amazon S3   AWS Lambda   AWS IAM   Amazon DynamoDB   Amazon SQS

ACA Lab 7: Tasks

1. Inspect the resources created automatically for your lab.
2. Create an SNS topic to receive notifications from one of the AWS Lambda functions and subscribe to it with your e-mail.
3. Create two SQS queues and subscribe them to the SNS topic.
4. Create an AWS Lambda function that will detect an upload to an Amazon S3 bucket, process the file, store the customer identification data it finds into one DynamoDB table, and store the purchase amounts into another DynamoDB table.
5. Create a second AWS Lambda function that detects additions to the second table, adds the values together, and sends a notification to the SNS topic if the total purchases amount to more than $1500.
6. Test the environment by uploading a sample text file to your bucket.

ACA Lab 7: Final Product


Upload transactions file
Input S3 bucket for transactions
Lambda function processes transactions file and inserts items in DynamoDB table
Customer DynamoDB table
Transactions DynamoDB table
Lambda function aggregates transactions and updates totals in DynamoDB table. It also sends a notification if account balance is greater than 1500.
TransactionTotal DynamoDB table
HighAccountBalance Alert SNSTopic
Email
Notification sent to your e-mail address
UserNotificationQueue SQS queue
CreditCollectionQueue SQS queue

While using another load balancer for the private instances would make this more highly available, in the interest of time we have omitted that step.