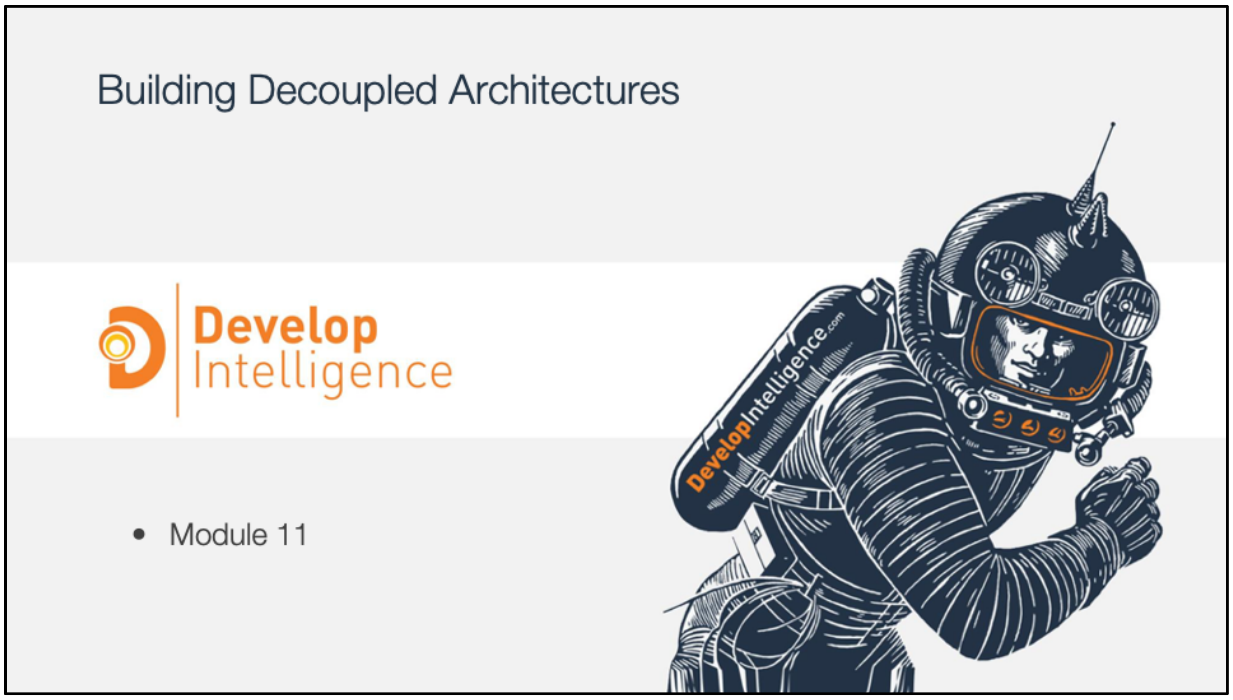
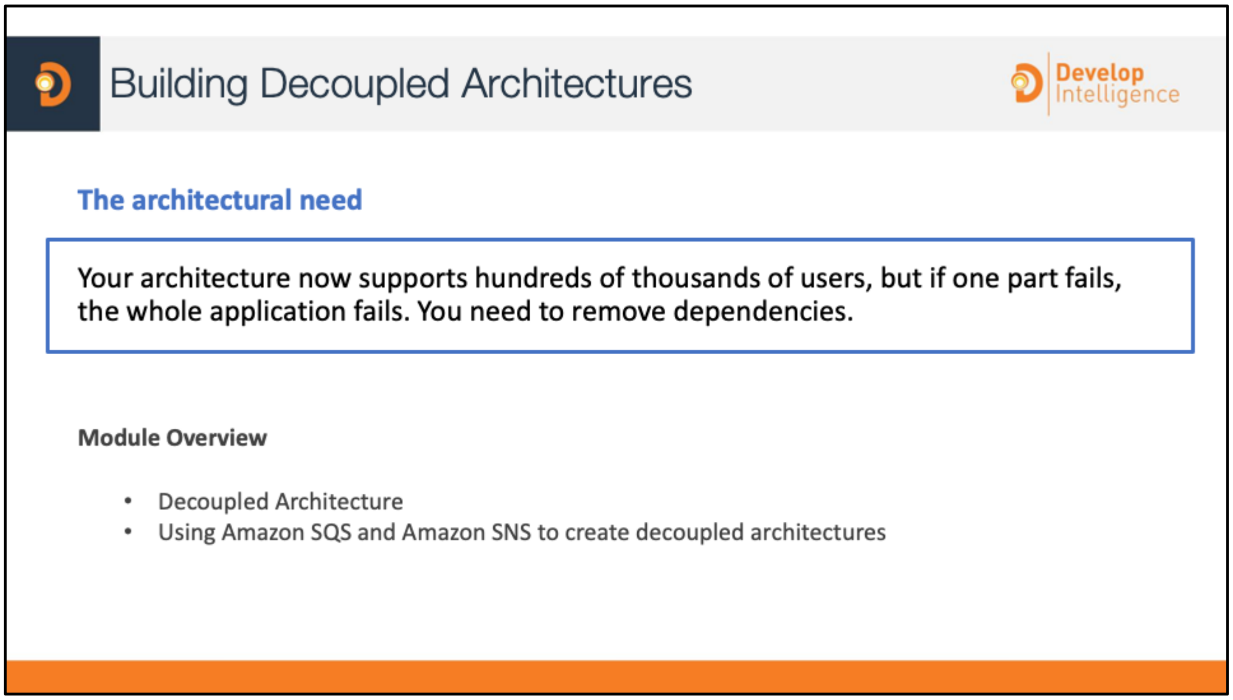


Building Decoupled Architectures

Develop Intelligence

- Module 11



Building Decoupled Architectures

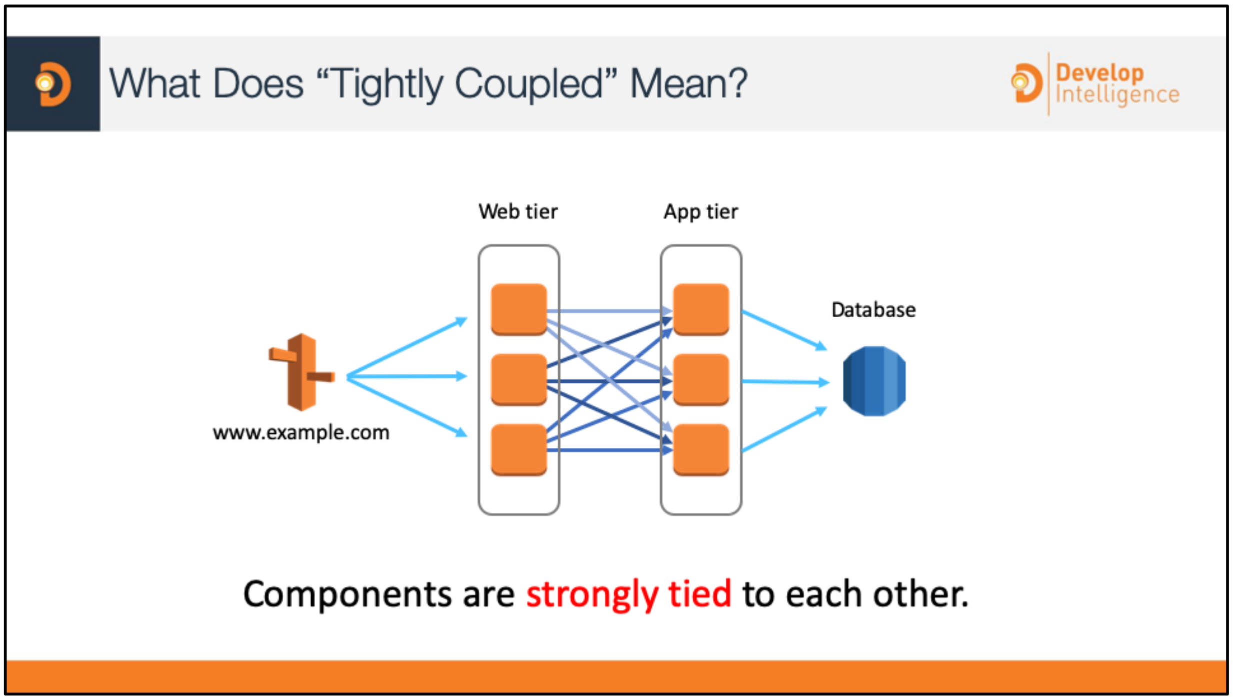
Develop Intelligence

The architectural need

Your architecture now supports hundreds of thousands of users, but if one part fails, the whole application fails. You need to remove dependencies.

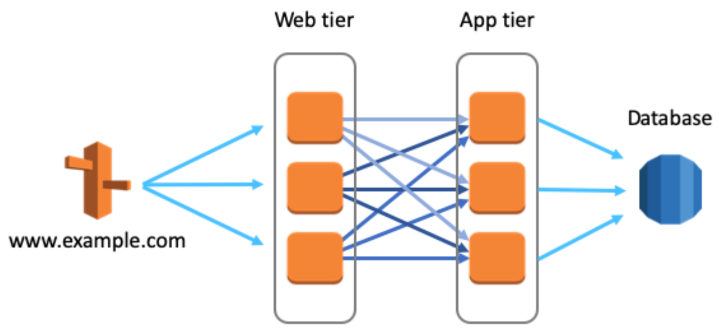
Module Overview

- Decoupled Architecture
- Using Amazon SQS and Amazon SNS to create decoupled architectures



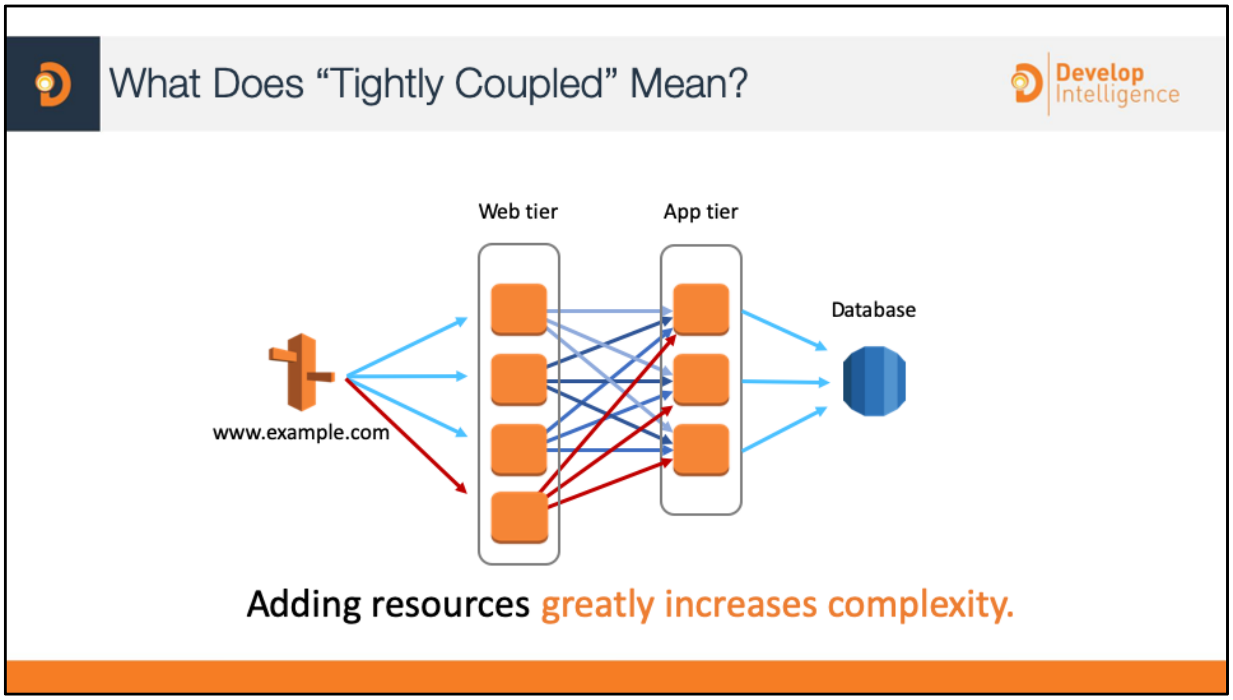
What Does “Tightly Coupled” Mean?

Develop Intelligence



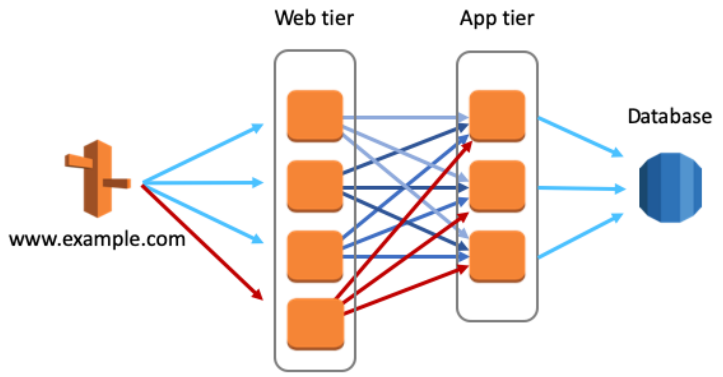
Components are strongly tied to each other.

Traditional infrastructures revolve around chains of tightly integrated servers, each with a specific purpose. When one of those components/layers goes down, however, the disruption to the system can ultimately be fatal. Additionally, it impedes scaling; if you add or remove servers at one layer, every server on every connecting layer has to be connected appropriately also.



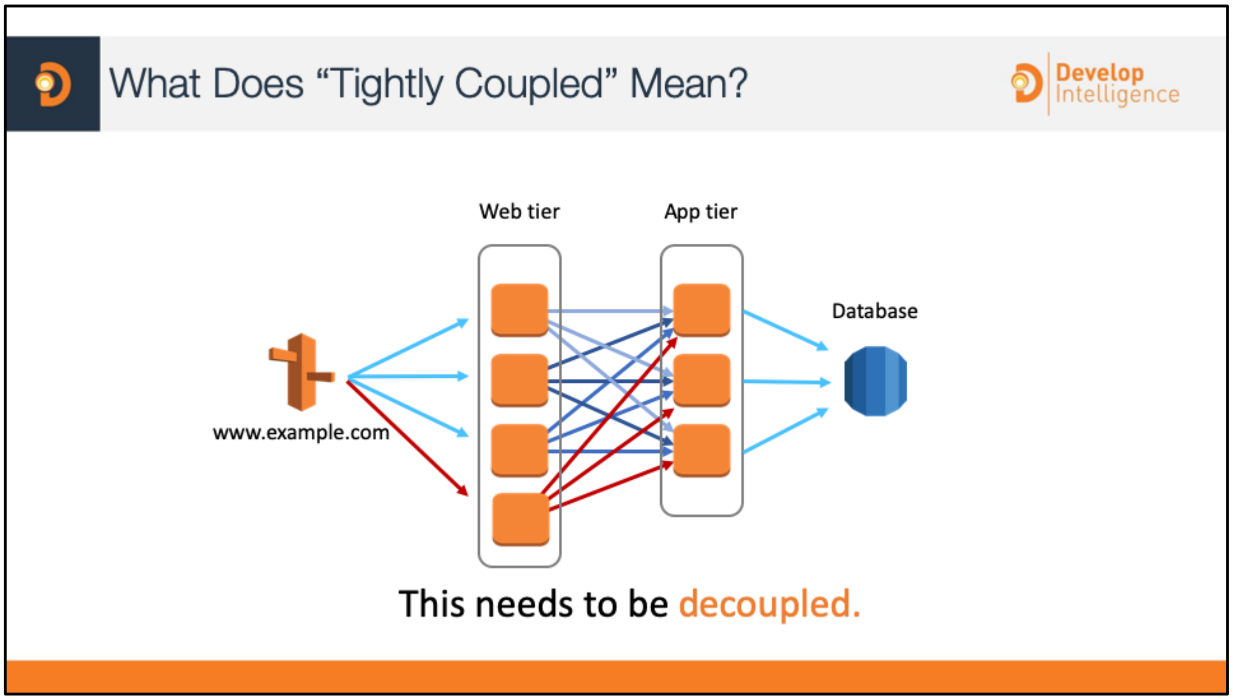
What Does "Tightly Coupled" Mean?

Develop Intelligence



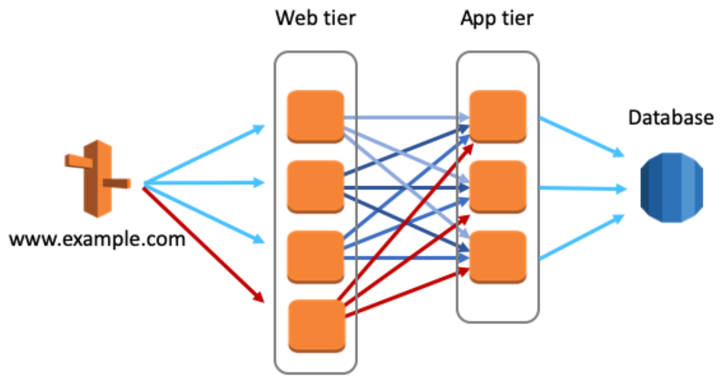
Adding resources greatly increases complexity.

You need to reduce interdependencies so that the 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.

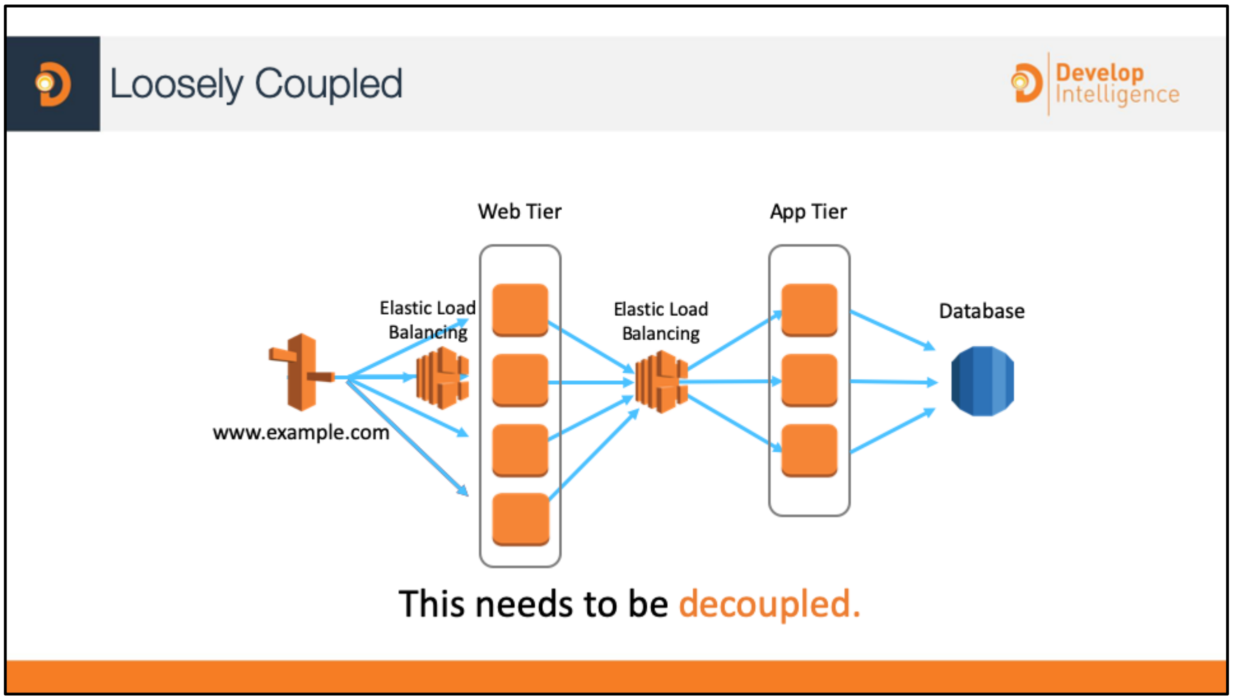


What Does "Tightly Coupled" Mean?

Develop Intelligence

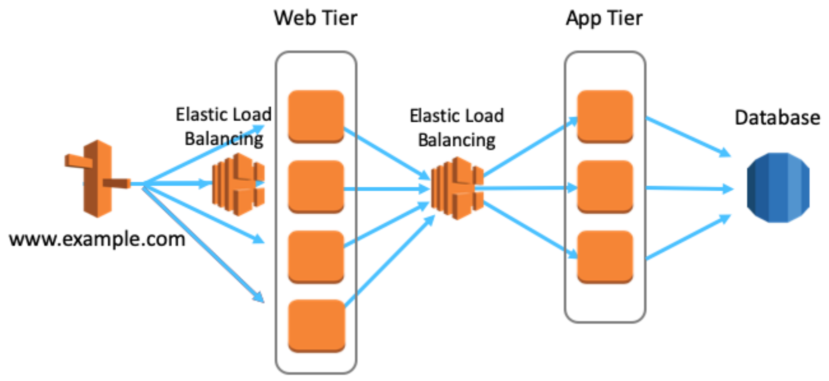


This needs to be decoupled.

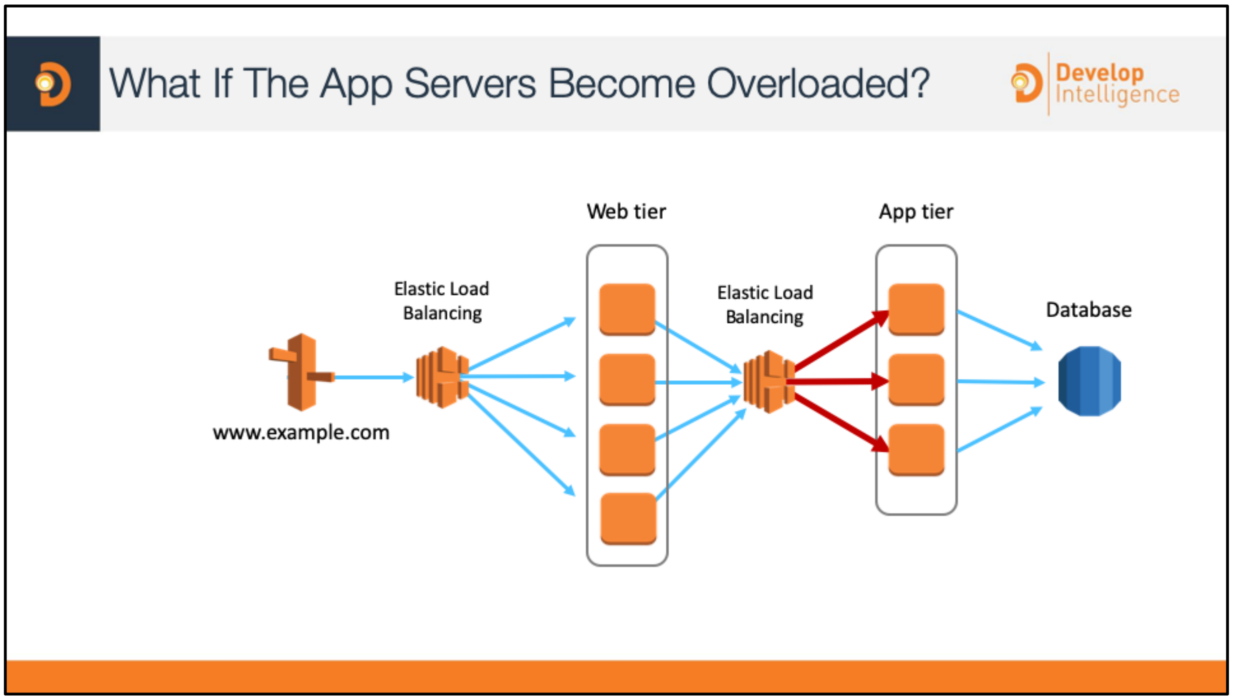


Loosely Coupled

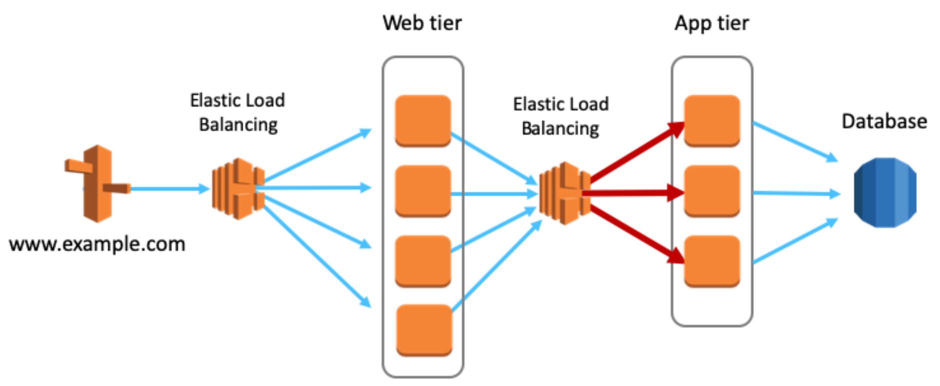
Develop Intelligence



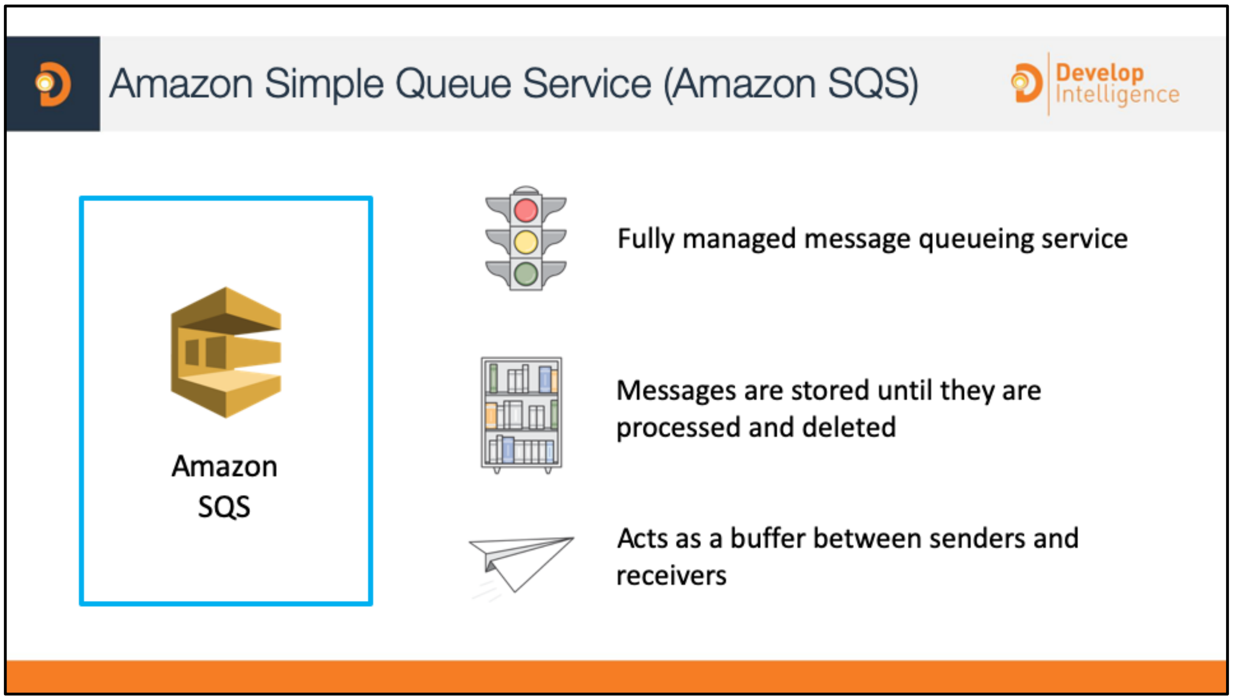
This needs to be decoupled.



What If The App Servers Become Overloaded?



Consider a web application that processes customer orders. One potential point of vulnerability in the order processing workflow is in saving the order in the database. As a new requirement, the business expects that every order has been persisted into the database. However, any potential deadlock, race condition, or network issue could cause the persistence of the order to fail. Then, the order is lost with no recourse to restore the order.



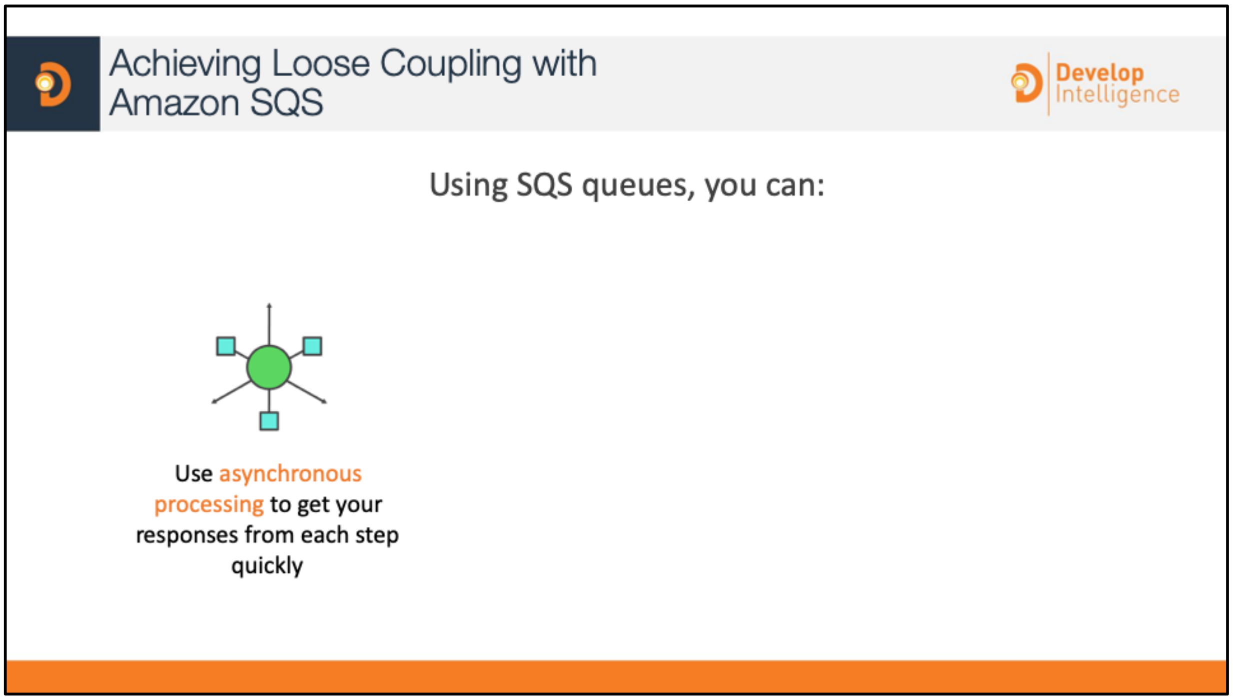
Amazon Simple Queue Service (Amazon SQS)

Develop Intelligence
Amazon SQS

- Fully managed message queueing service
- Messages are stored until they are processed and deleted
- Acts as a buffer between senders and receivers

Amazon SQS is a fully-managed service that requires no administrative overhead and little configuration. The service works on a massive scale, processing billions of messages per day. It stores all message queues and messages within a single, highly-available AWS Region with multiple redundant Availability Zones, so that no single computer, network, or Availability Zone failure can make messages inaccessible. Messages can be sent and read simultaneously.

Developers can securely share Amazon SQS queues anonymously or with specific AWS accounts. Queue sharing can also be restricted by IP address and time-of-day. Server-side encryption (SSE) protects the contents of messages in Amazon SQS queues using keys managed in the AWS Key Management Service (AWS KMS). SSE encrypts messages as soon as Amazon SQS receives them. The messages are stored in encrypted form and Amazon SQS decrypts messages only when they are sent to an authorized consumer.

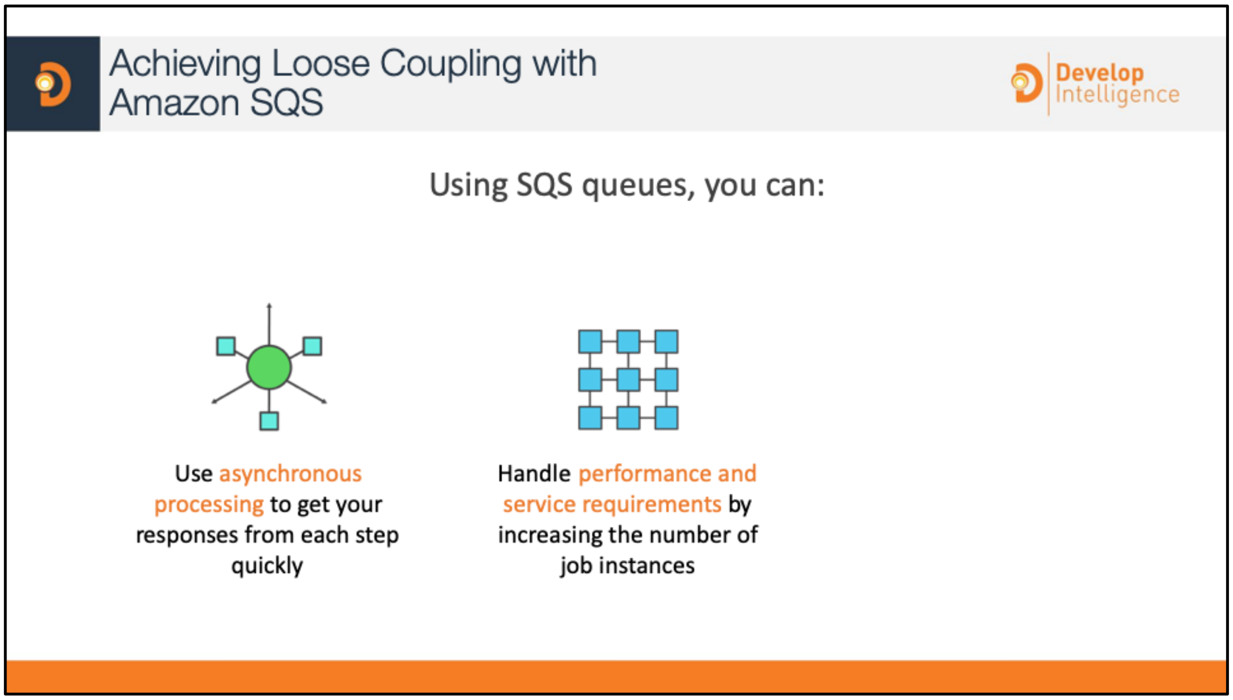


Achieving Loose Coupling with Amazon SQS

Develop Intelligence

Using SQS queues, you can:

Use asynchronous processing to get your responses from each step quickly

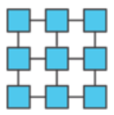


Achieving Loose Coupling with Amazon SQS

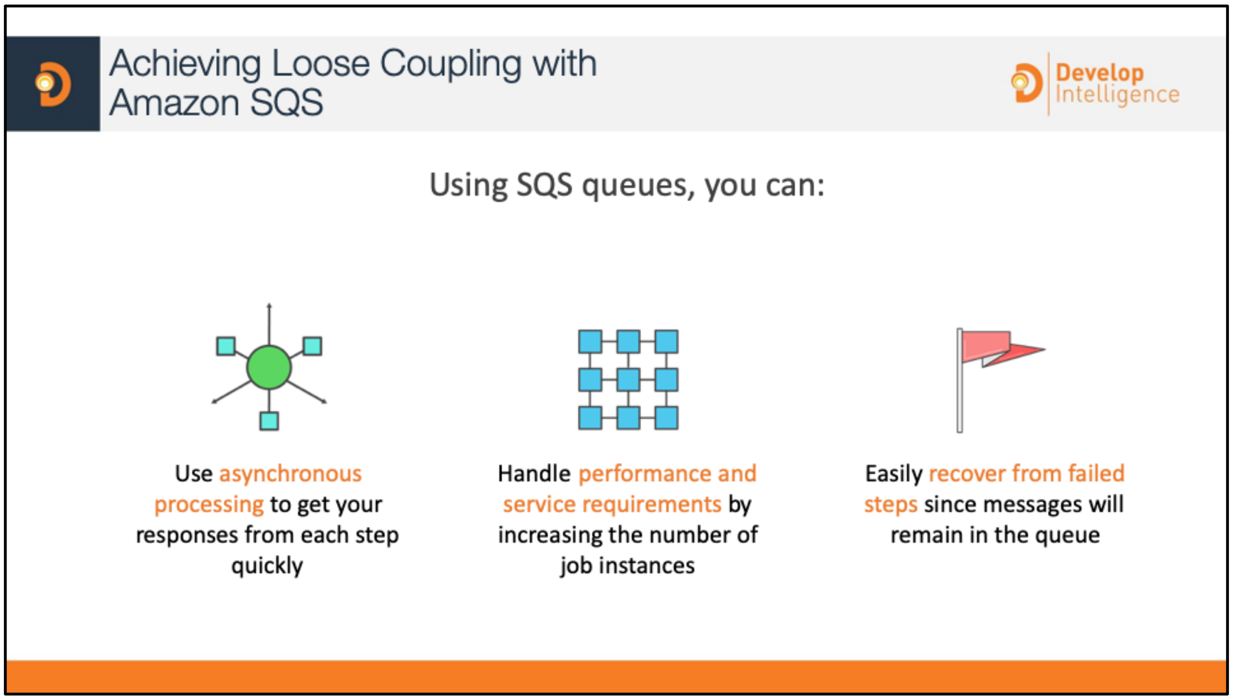
Develop Intelligence

Using SQS queues, you can:

Use asynchronous processing to get your responses from each step quickly



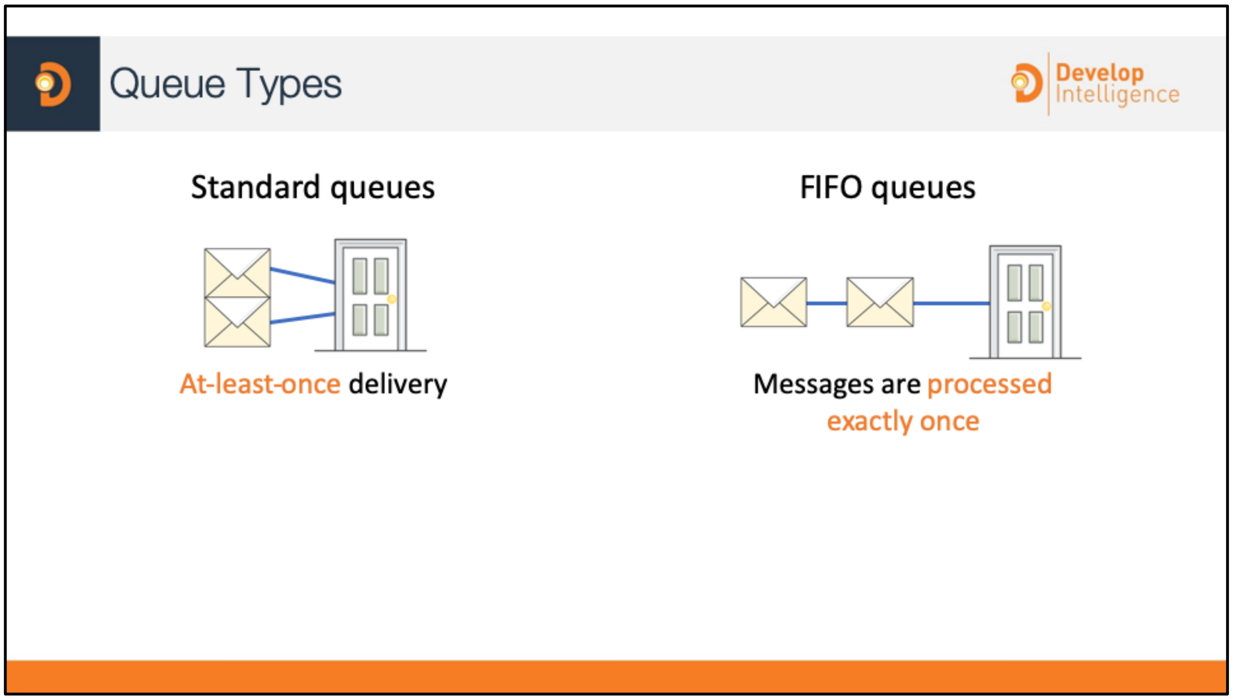
Handle performance and service requirements by increasing the number of job instances



Achieving Loose Coupling with Amazon SQS

Using SQS queues, you can:

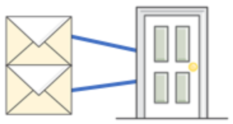
- Use asynchronous processing to get your responses from each step quickly
- Handle performance and service requirements by increasing the number of job instances
- Easily recover from failed steps since messages will remain in the queue



Queue Types

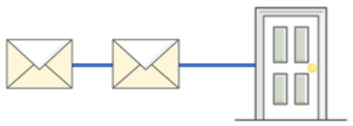
Develop Intelligence

Standard queues



At-least-once delivery

FIFO queues

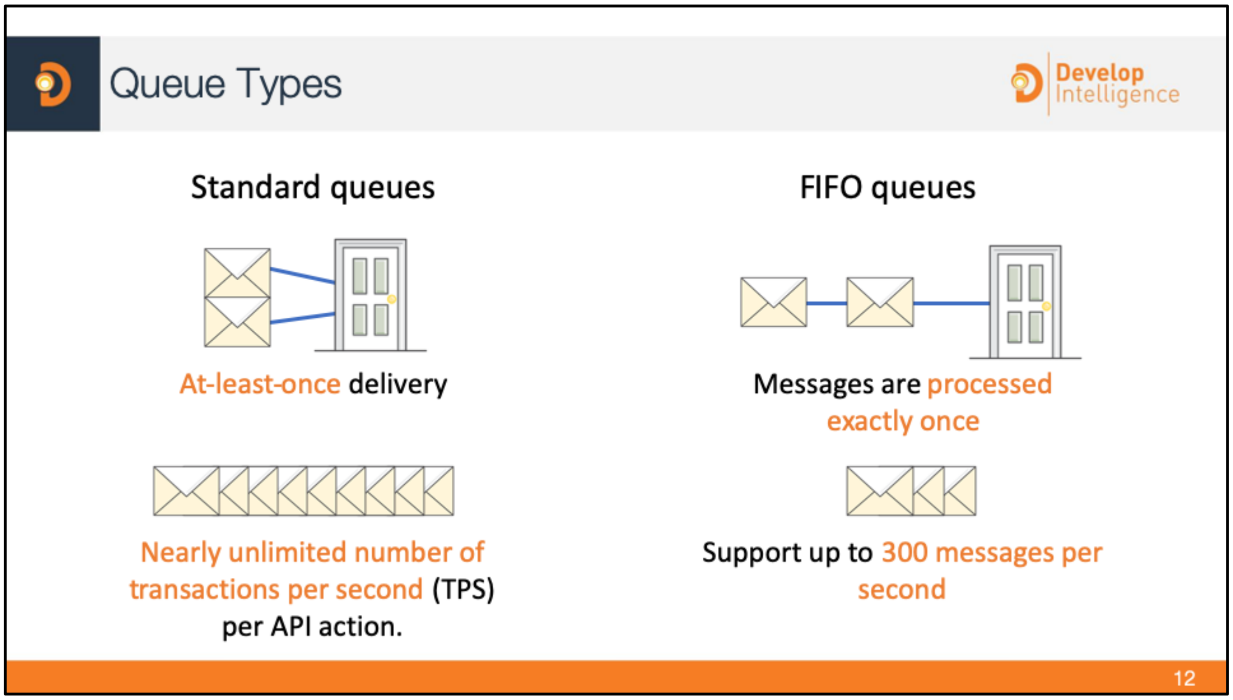


Messages are processed exactly once

There are two types of SQS Queues: Standard and FIFO.

Standard queues offer at-least-once delivery and best-effort ordering.
At-Least-Once Delivery means that occasionally more than one copy of a message is delivered.
Best-Effort Ordering means occasionally messages might be delivered in an order different from which they were sent.

FIFO (*first in, first out*) queues are designed to guarantee that messages are processed exactly once, in the exact order that they are sent.



Queue Types

Develop Intelligence

Standard queues

At-least-once delivery

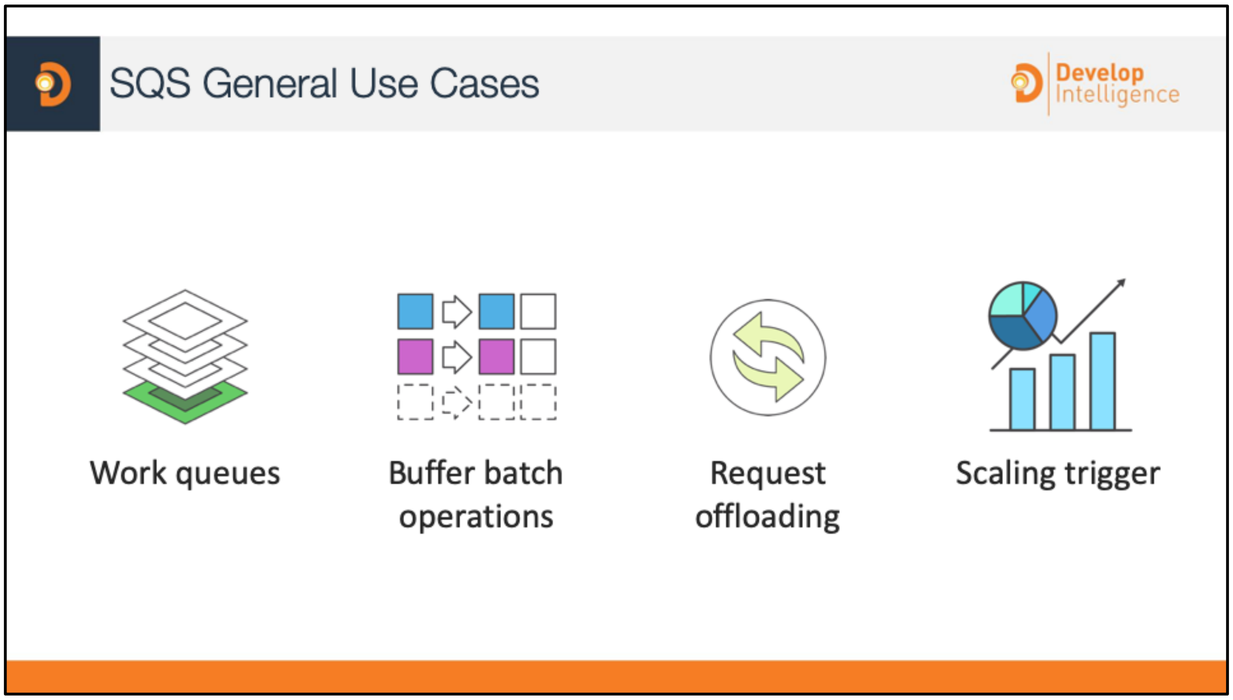
Nearly unlimited number of transactions per second (TPS) per API action.

FIFO queues

Messages are processed exactly once

Support up to 300 messages per second

However, standard queues offer maximum throughput, while 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.



SQS General Use Cases

Develop Intelligence

Work queues

Buffer batch operations

Request offloading

Scaling trigger

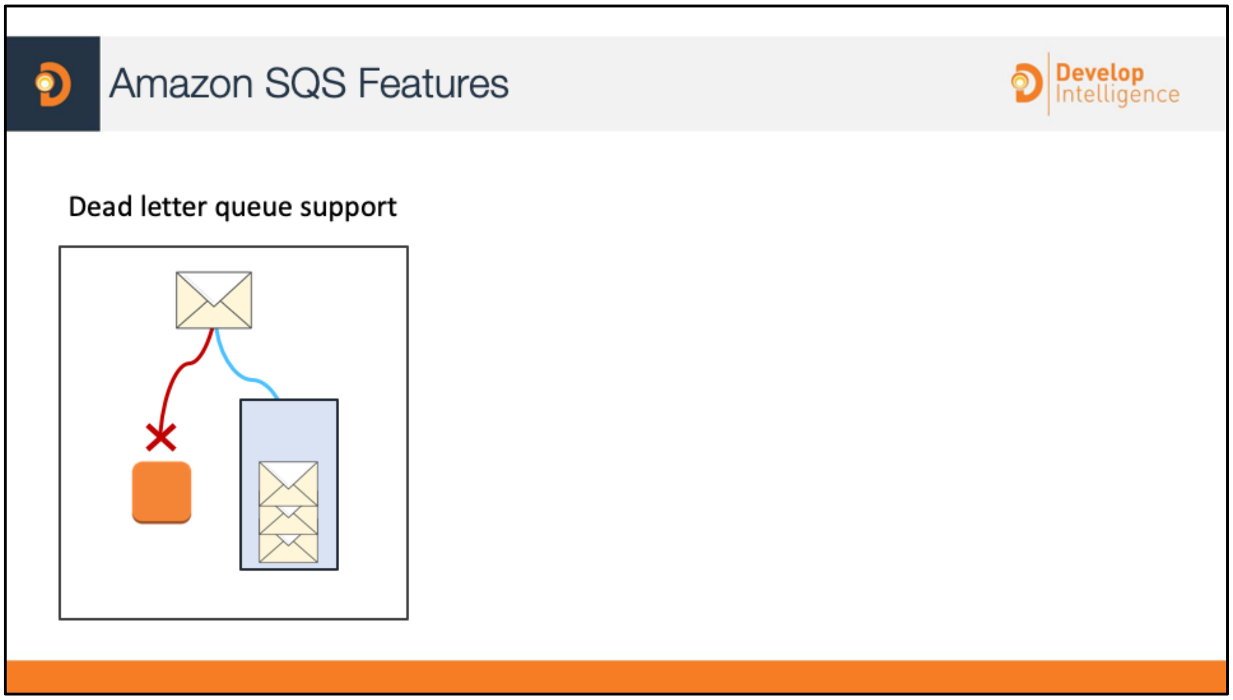
There are many different ways to use SQS queues.

Work queues: Decouple components of a distributed application that may 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: Move slow operations off of interactive request paths by enqueueing the request.

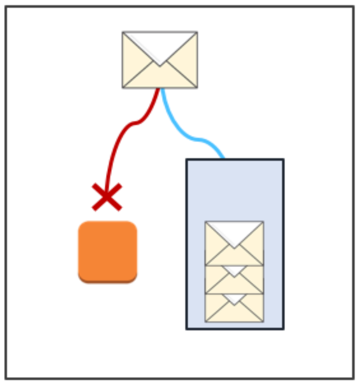
Auto Scaling: Use Amazon SQS queues to help determine the load on an application, and when combined with Auto Scaling, you can scale the number of Amazon EC2 instances out or in, depending on the volume of traffic.



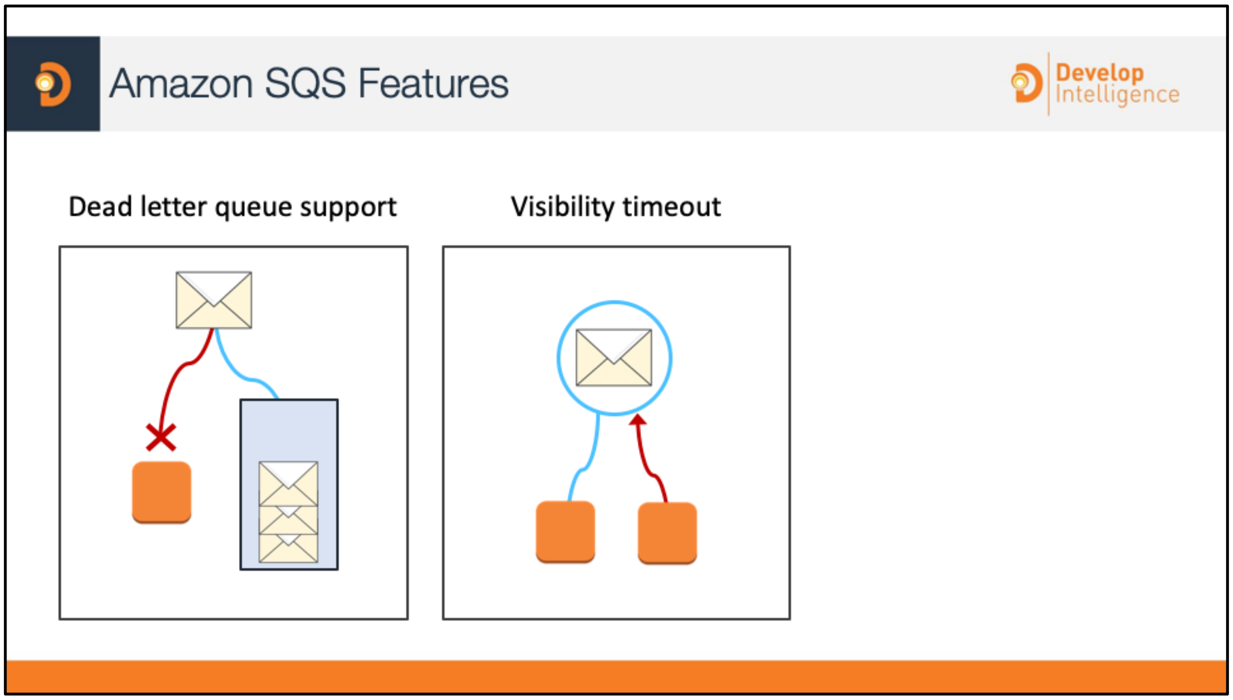
Amazon SQS Features

Develop Intelligence

Dead letter queue support



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



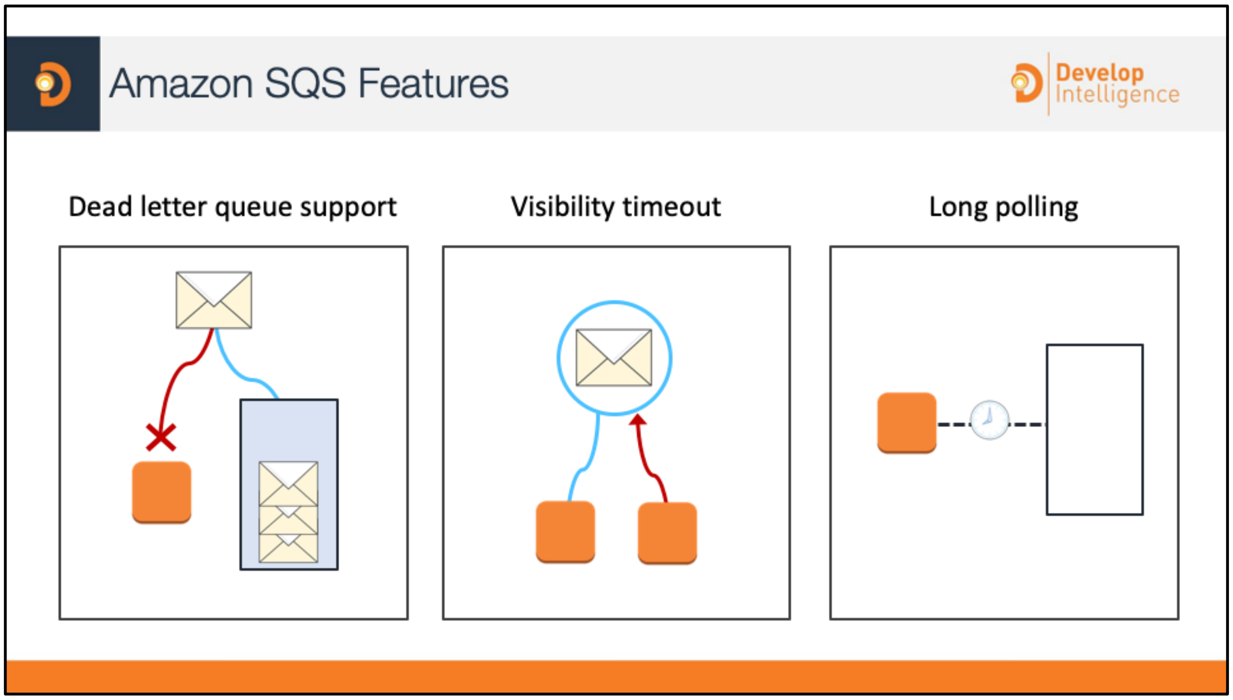
Amazon SQS Features

Develop Intelligence

Dead letter queue support

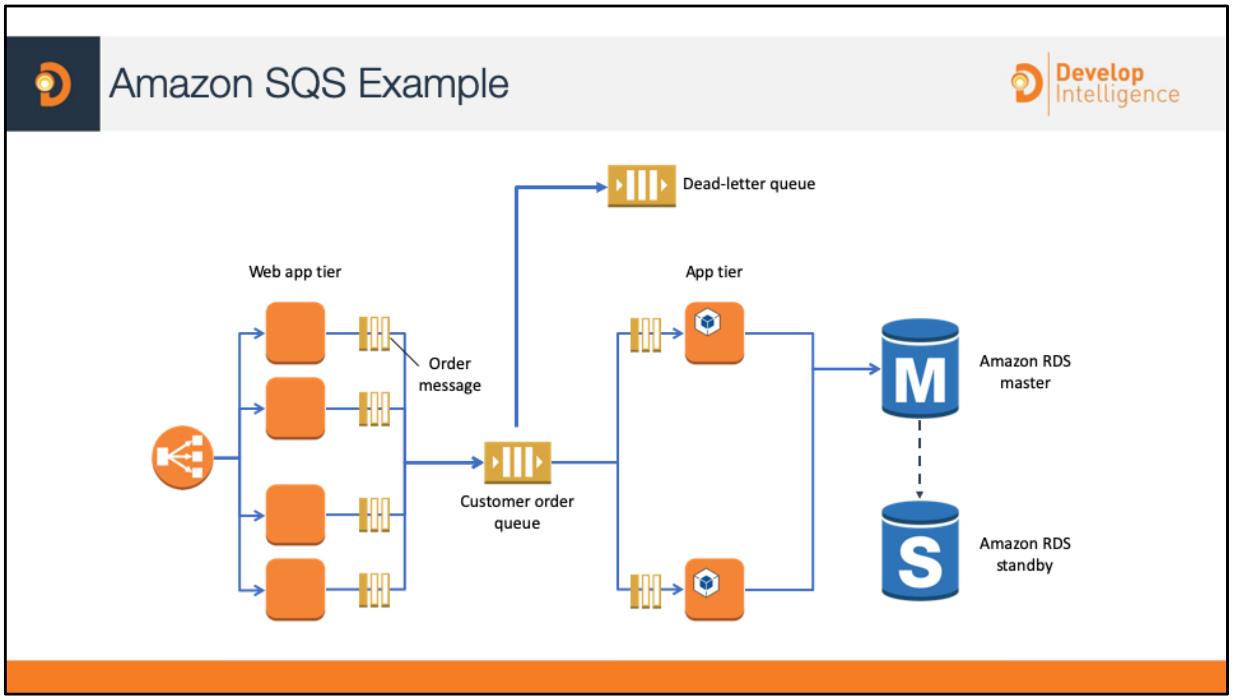
Visibility timeout

Visibility timeout is the period of time that a message is "invisible" to the rest of your application after an application component gets it from the queue. During the visibility timeout, the component that received the message usually processes it and then deletes it from the queue. This prevents multiple components from processing the same message. When the application needs more time for processing, the "invisible" timeout can be modified.



Amazon SQS Features

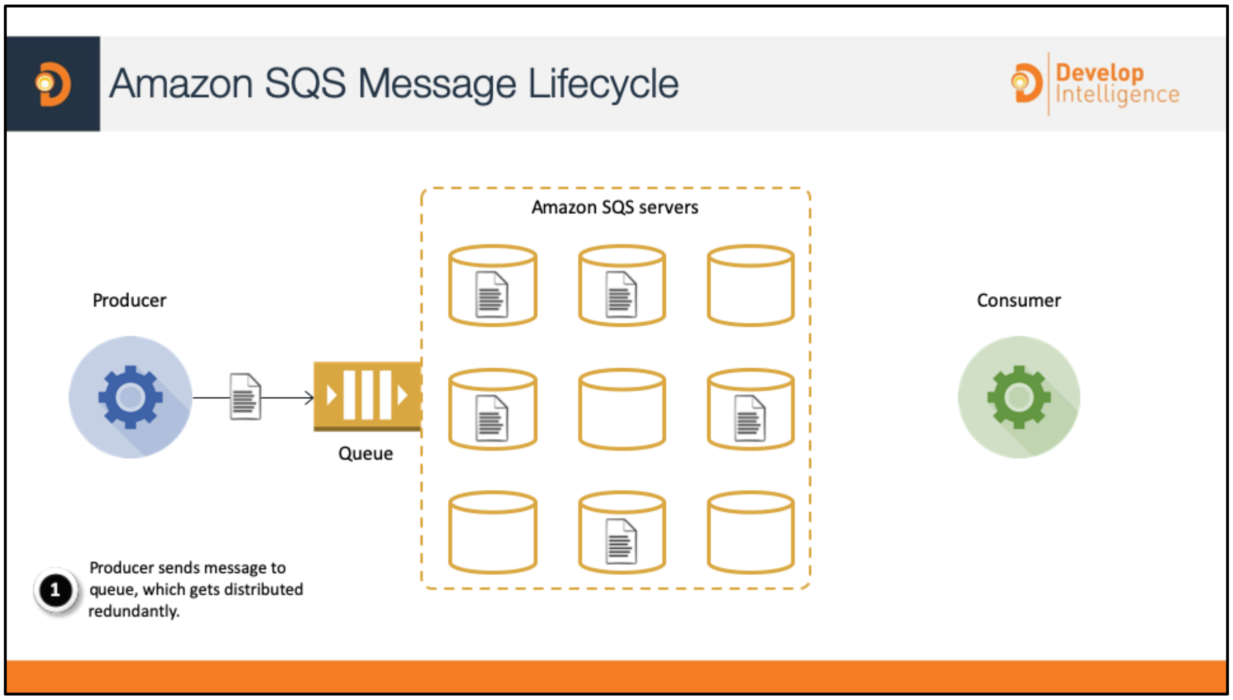
Long polling is a way to retrieve messages from your Amazon SQS queues. The default of short polling returns immediately, even if the message queue being polled is empty. However, long polling doesn't return a response until a message arrives in the message queue, or the long poll times out. Long polling makes it inexpensive to retrieve messages from your Amazon SQS queue as soon as the messages are available.



Amazon SQS Example

Introducing an SQS queue helps improve your ordering application. Using the queue isolates the processing logic into its own component and runs it in a separate process from the web application. This, in turn, allows the system to be more resilient to spikes in traffic, while allowing work to be performed only as fast as necessary in order to manage costs. In addition, you now have a mechanism for persisting orders as messages (with the queue acting as a temporary database), and have moved the scope of your transaction with your database further down the stack. In the event of an application exception or transaction failure, this ensures that the order processing can be retired or redirected to the Amazon SQS Dead Letter Queue (DLQ), for re-processing at a later stage.

For more information regarding this use case, see https://aws.amazon.com/blogs/compute/building-loosely-coupled-scalable-c-applications-with-amazon-sqs-and-amazon-sns/

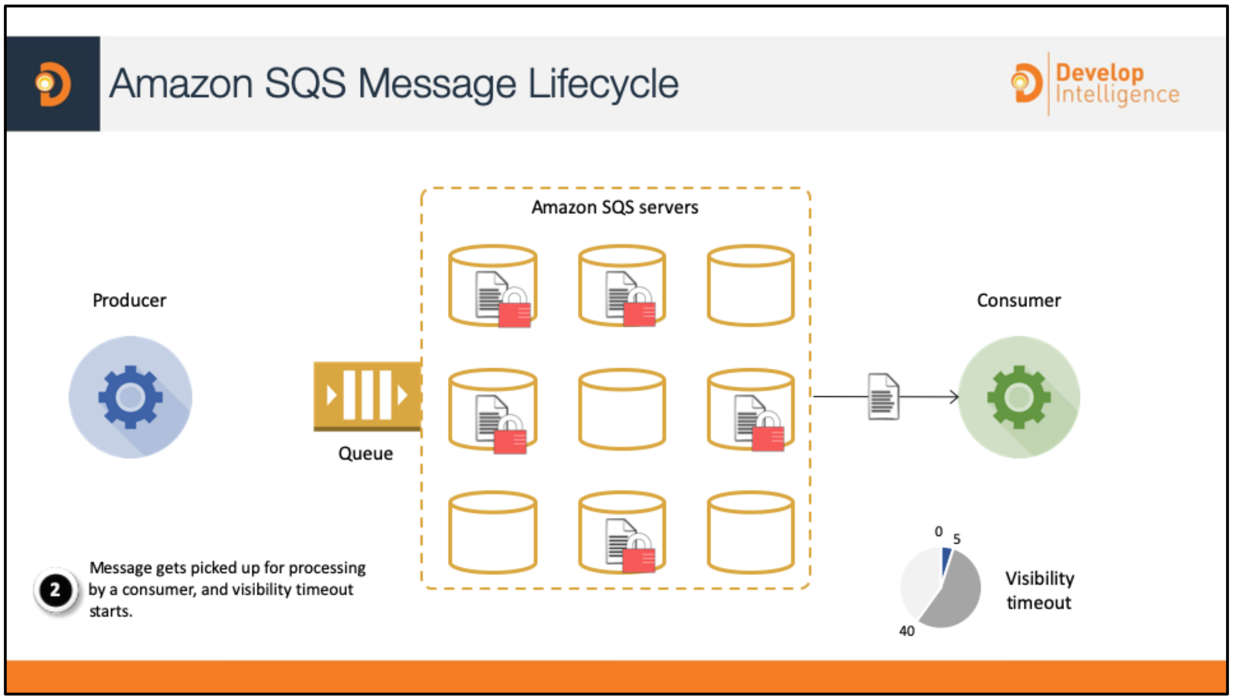


Amazon SQS Message Lifecycle

Develop Intelligence
Amazon SQS servers
Producer
Consumer
Queue
1
Producer sends message to queue, which gets distributed redundantly.

Amazon Simple Queue Service (Amazon SQS) is a distributed queue system that enables web service applications to queue messages that one component in the application generates to be consumed by another component. A *queue* is a temporary repository for messages that are waiting to be processed, keeping messages from 1 to 14 days (default is 4 days). Using Amazon SQS, you can decouple the components of an application so they run independently. Messages can contain up to 256 KB of text in any format. Amazon SQS supports multiple producers and consumers interacting with the same queue. Amazon SQS can be used with several AWS services including: Amazon EC2, Amazon S3, Amazon ECS, AWS Lambda, and Amazon DynamoDB.

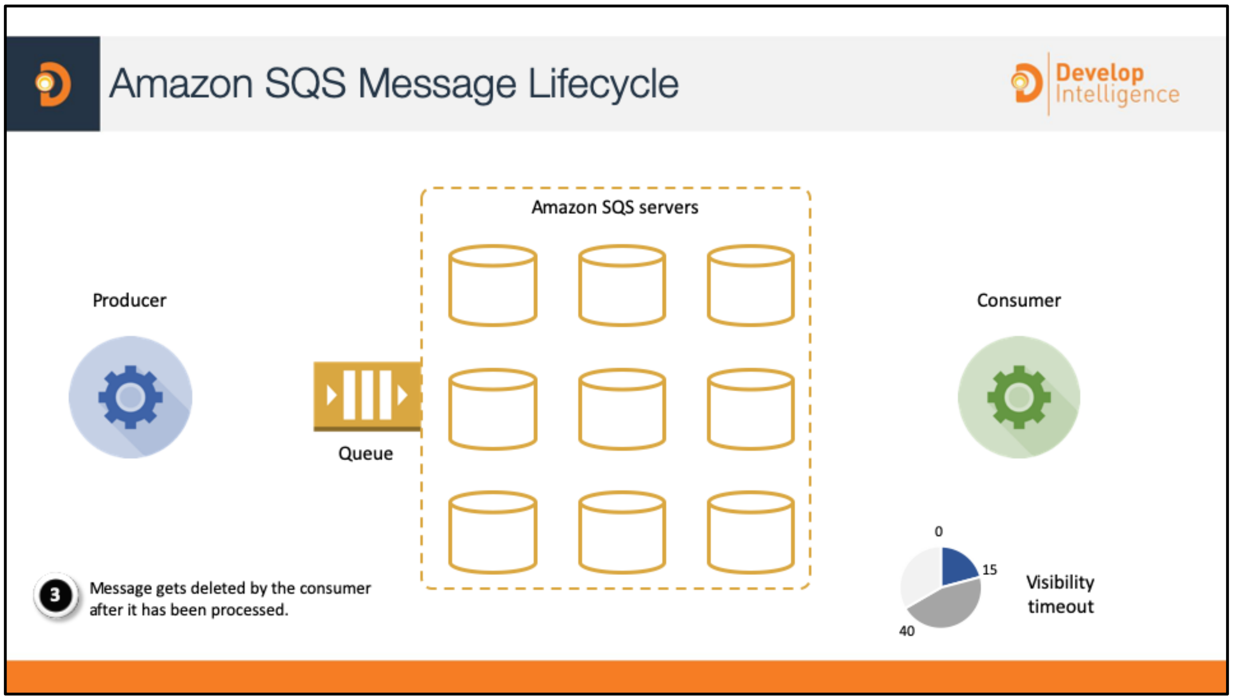
Amazon SQS offers two types of message queues. Standard queues offer maximum throughput, best-effort ordering, and at-least-once delivery. Amazon SQS FIFO queues are designed to guarantee that messages are processed exactly once, in the exact order that they are sent, with limited throughput. The following scenario describes the lifecycle of an Amazon SQS message in a queue, from creation to deletion. Here we have a producer sending a message to a queue, and the message is distributed across the Amazon SQS servers redundantly.



Amazon SQS Message Lifecycle

2 Message gets picked up for processing by a consumer, and visibility timeout starts.

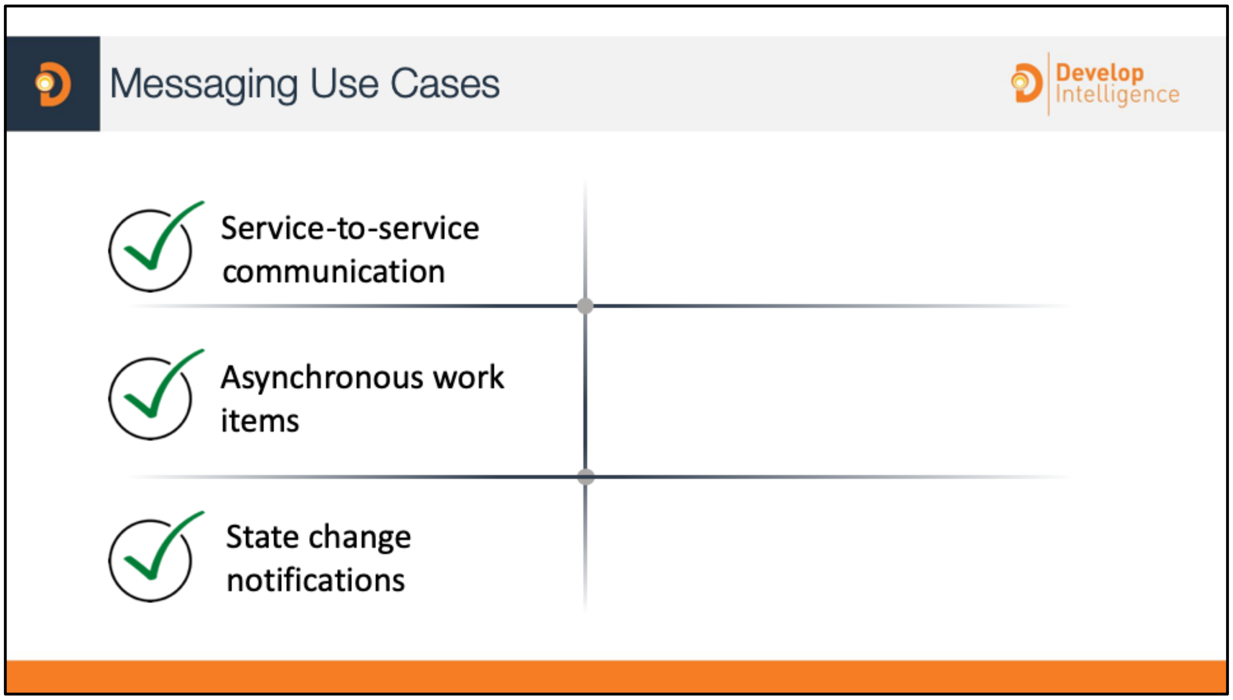
When a consumer is ready to process messages, it retrieves the message from the queue. While the message is being processed, it remains in the queue. To prevent other consumers from processing the message again, Amazon SQS sets a *visibility timeout*, a period of time during which Amazon SQS prevents other consumers from receiving and processing the message. The default visibility timeout for a message is 30 seconds. In this case, we have it set up for 40 seconds. The maximum is 12 hours. If the consumer fails before deleting the message before the visibility timeout expires, the message becomes visible to other consumers and the message may 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 Message Lifecycle

3 Message gets deleted by the consumer after it has been processed.

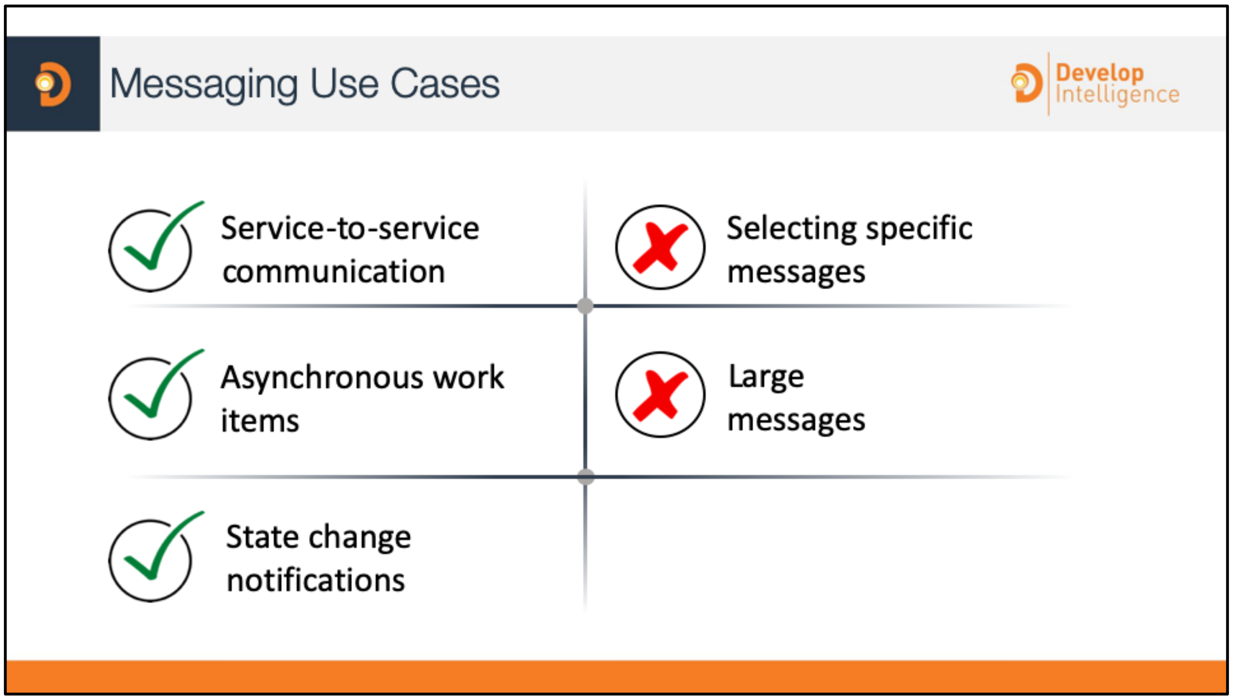
Amazon SQS doesn't automatically delete the message. Because Amazon SQS is a distributed system, there's no guarantee that the consumer actually receives the message (for example, due to a connectivity issue, or due to an issue in the consumer application). Thus, the consumer must delete the message from the queue after receiving and processing it.



Messaging Use Cases

- ✓ Service-to-service communication
- ✓ Asynchronous work items
- ✓ State change notifications

There are some common use cases where messaging services are a great fit. When there are two services or systems that need to communicate with each other. Let's say a website (the frontend) has to update customer's delivery address in a customer relationship management (CRM) system (the backend). Alternatively, you can also set up a queue and have the frontend website code send messages to the queue and have the backend CRM service to consume them. Or let's say a hotel booking system needs to cancel a booking and this process takes a long time. Alternatively, you can put a message into a queue and have the same hotel booking system consume messages from that queue and perform asynchronous cancellations. Also, messaging services are great for change notifications. You have a service that manages some resource and other services that receive updates about changes to those resources. For example, an inventory system may publish notifications when a certain item is low and needs ordering.

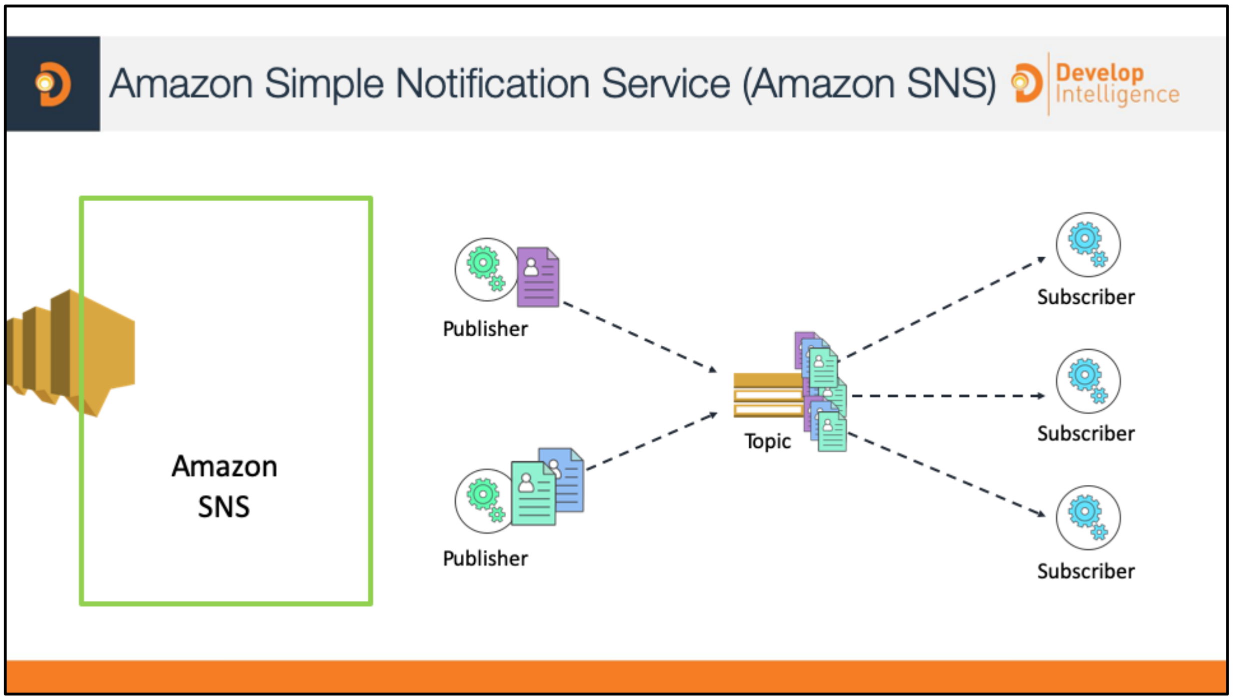


Messaging Use Cases

Develop Intelligence

✓ Service-to-service communication	✗ Selecting specific messages
✓ Asynchronous work items	✗ Large messages
✓ State change notifications	

It’s also important to know when a particular technology won’t fit well with your use case. Messaging has its own set of commonly encountered anti-patterns. It’s tempting to have the ability to receive messages selectively from a queue that match a particular set of attributes, or even 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. This can lead to a scenario where there are messages in the queue that no one is polling for and are never consumed.** 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 that store in the message itself.



Amazon Simple Notification Service (Amazon SNS)

Amazon Simple Notification Service (Amazon SNS) is a web service that makes it easy to set up, operate, and send notifications from the cloud. The service follows the "publish-subscribe" (pub-sub) messaging paradigm, with notifications being delivered to clients using a "push" mechanism.

You create a topic and control access to it by defining policies that determine which publishers and subscribers can communicate with the topic. A publisher sends messages to topics they have created or to topics they have permission to publish to.

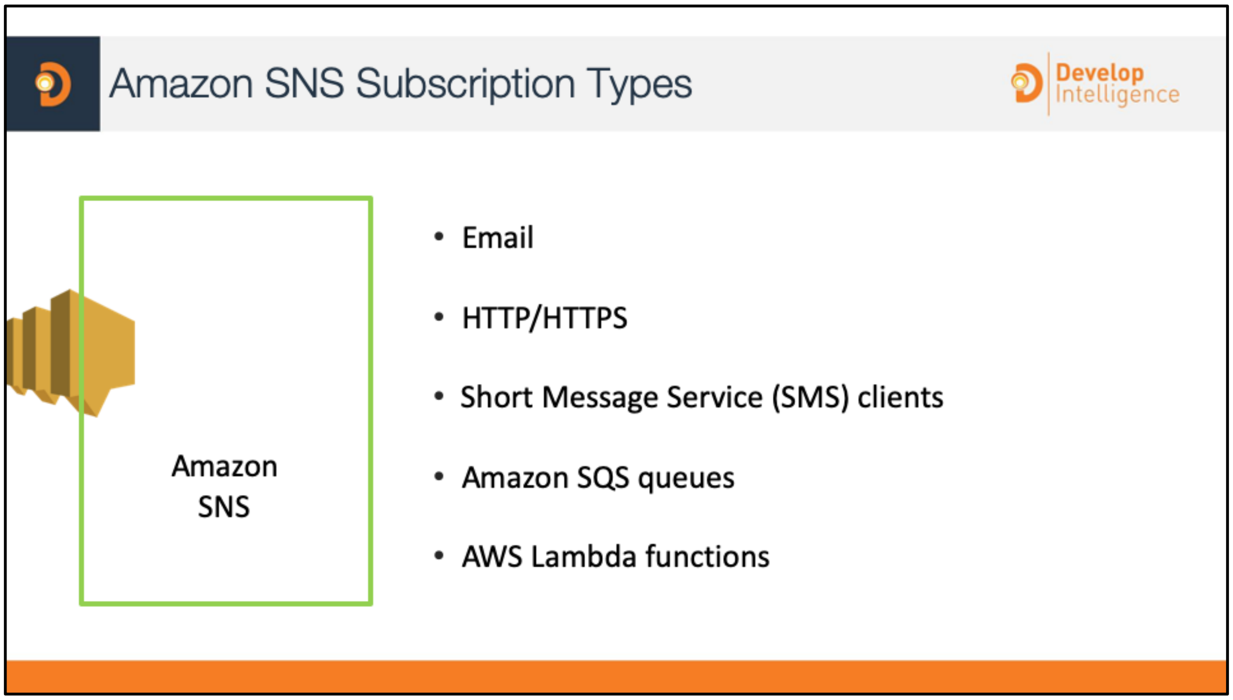
Instead of including a specific destination address in each message, a publisher sends a message to the topic. Amazon SNS matches the topic to a list of subscribers who have subscribed to that topic and delivers the message to each of those subscribers.

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 published to the topics to which they subscribe, and all subscribers to a topic receive the same messages.

Amazon SNS supports encrypted topics. When you publish messages to encrypted topics, Amazon SNS uses customer master keys (CMK), powered by AWS KMS (https://aws.amazon.com/kms/) , to encrypt your messages.

Amazon SNS supports customer-managed as well as AWS-managed CMKs. As soon as Amazon SNS receives your messages, the encryption takes place on the server, using a 256-bit AES-GCM algorithm. The messages are stored in encrypted form across multiple Availability Zones (AZs) for durability and are decrypted just before being delivered to subscribed endpoints, such as Amazon Simple Queue Service (Amazon SQS) queues, AWS Lambda functions, and HTTP and HTTPS webhooks.

https://aws.amazon.com/blogs/compute/encrypting-messages-published-to-amazon-sns-with-aws-kms/



Amazon SNS Subscription Types

Develop Intelligence
Amazon SNS

- Email
- HTTP/HTTPS
- Short Message Service (SMS) clients
- Amazon SQS queues
- AWS Lambda functions

Customers can select one the following transports as part of the subscription requests:

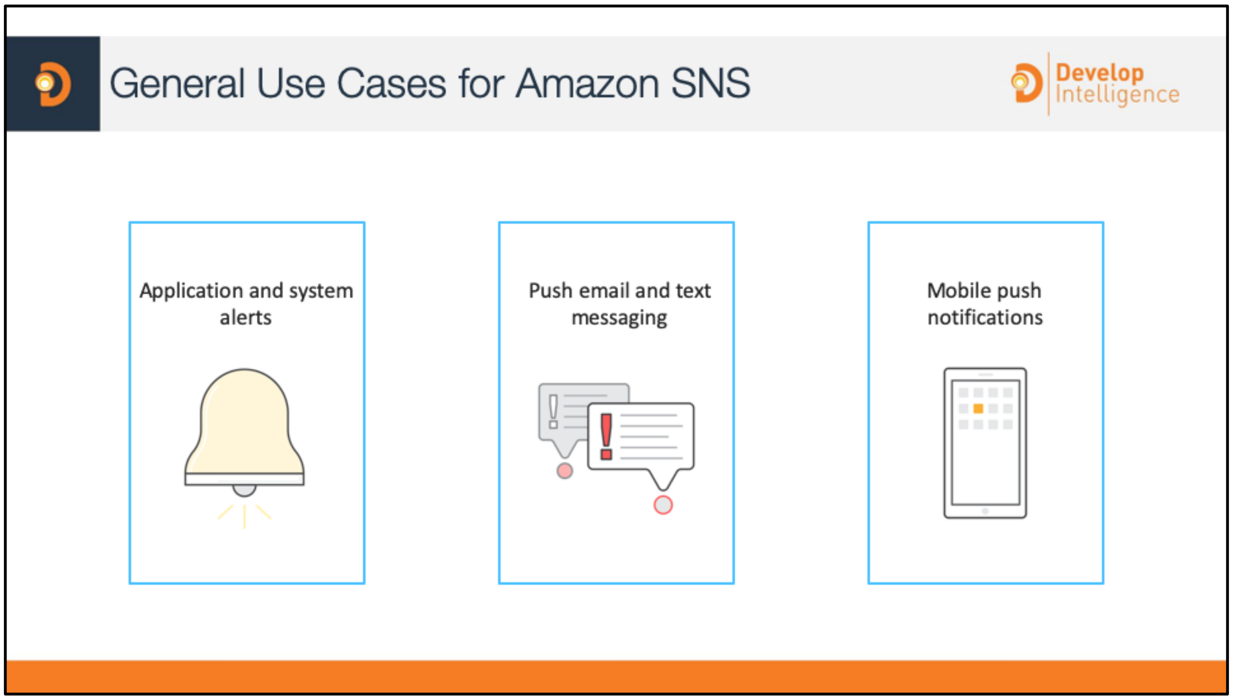
”Email” or “Email-JSON” – Messages are sent to registered addresses as email. Email-JSON sends notifications as a JSON object, while Email sends text-based email.

“HTTP” or “HTTPS” – Subscribers specify a URL as part of the subscription registration; notifications will be delivered through an HTTP POST to the specified URL.

“SMS” – Messages are sent to registered phone numbers as SMS text messages.

“SQS” – Users can specify an SQS standard queue as the endpoint; Amazon SNS will enqueue a notification message to the specified queue. Note that FIFO queues are not currently supported.

Furthermore, messages can also be delivered to AWS Lambda functions for handling message customizations, enabling message persistence or communicating with other AWS service.



General Use Cases for Amazon SNS

Develop Intelligence

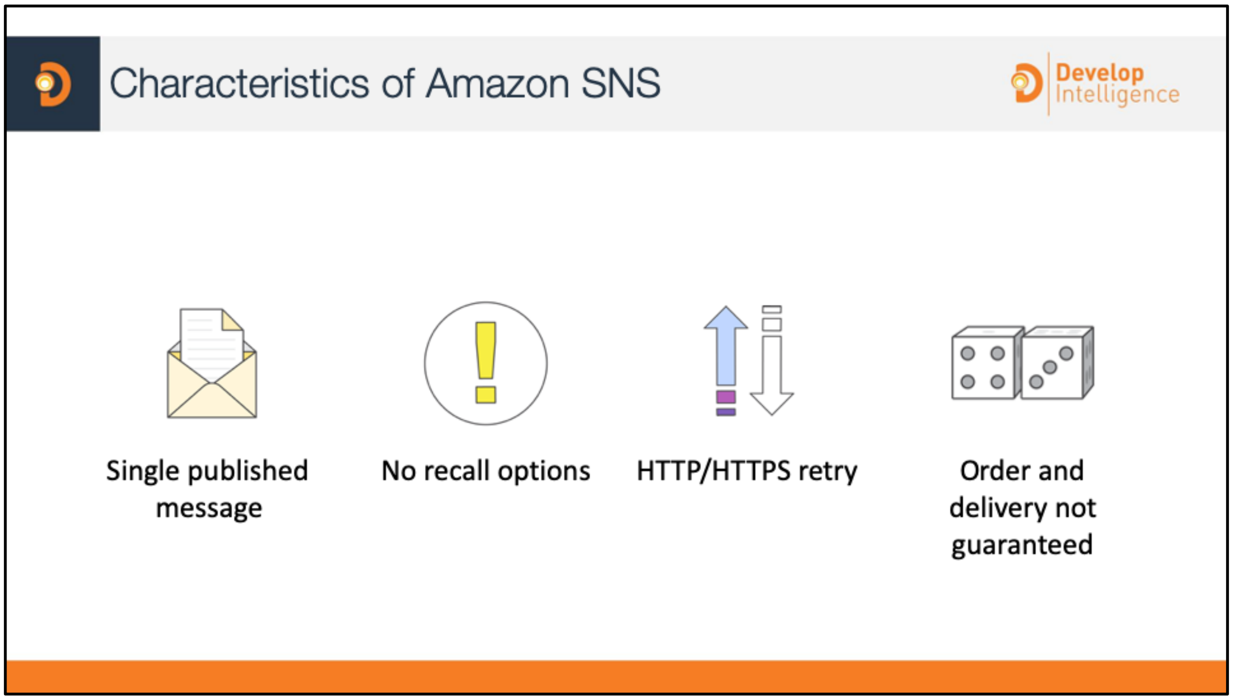
Application and system alerts

Push email and text messaging

Mobile push notifications

There are many ways to use Amazon SNS notifications.

- You could receive immediate notification when an event occurs, such as a specific change to your AWS Auto Scaling group.
- You could use Amazon SNS to push targeted news headlines to subscribers by email or SMS. Upon receiving the email or SMS text, interested readers could then choose to learn more by visiting a website or launching an application.
- You could send notifications to an app, indicating that an update is available. The notification message can include a link to download and install the update.



Characteristics of Amazon SNS

Develop Intelligence

Single published message

No recall options

HTTP/HTTPS retry

Order and delivery not guaranteed

All notification messages will contain a single published message.

Amazon SNS 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.

When a message is delivered successfully, there is no way to recall it.

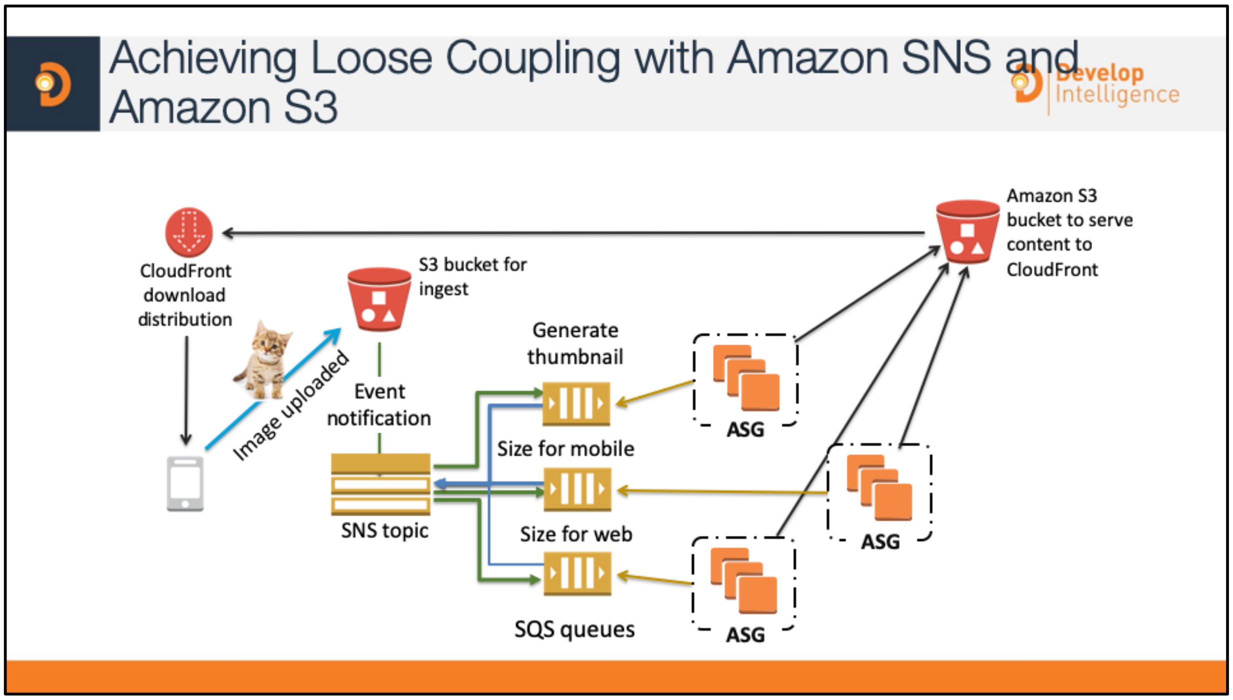
An Amazon SNS Delivery Policy can be used to control the retry pattern (linear, geometric, exponential back off), maximum and minimum retry delays, and other parameters.

To prevent messages from being lost, all messages published to Amazon SNS are stored redundantly across multiple servers and data centers.

Amazon SNS is designed to meet the needs of the largest and most demanding applications, allowing applications to publish an unlimited number of messages at any time.

Amazon SNS allows applications and end-users on different devices to receive notifications via Mobile Push notification (Apple, Google and Kindle Fire Devices), HTTP/HTTPS, Email/Email-JSON, SMS or Amazon Simple Queue Service (SQS) queues, or AWS Lambda functions.

Amazon SNS provides access control mechanisms to ensure that topics and messages are secured against unauthorized access. Topic owners can set policies for a topic that restrict who can publish or subscribe to a topic. Additionally, topic owners can ensure that notifications are encrypted by specifying that the delivery mechanism must be HTTPS.



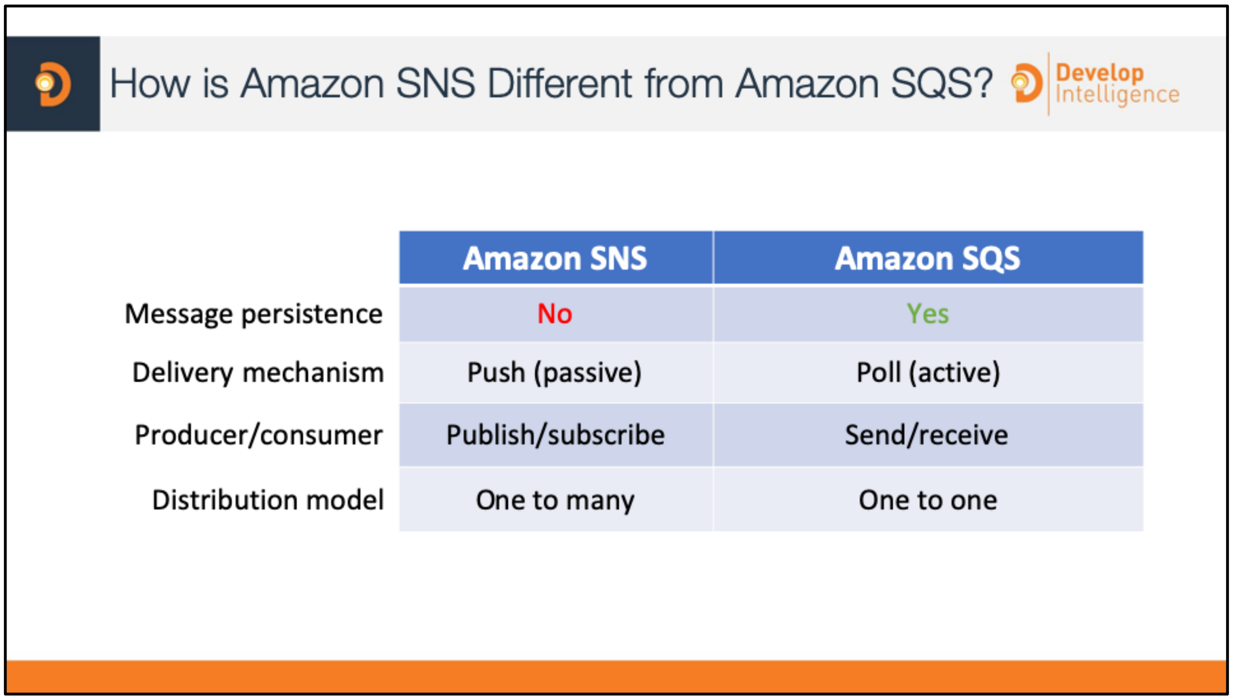
Achieving Loose Coupling with Amazon SNS and Amazon S3

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

SNS can be used to send messages within a single account or to resources in different accounts to create administrative isolation.

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 alternative scenario, uploading the image to Amazon S3 triggers an event notification, which sends the message to the SNS topic automatically.



How is Amazon SNS Different from Amazon SQS?

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

- Amazon SNS allows applications to send time-critical messages to multiple subscribers through a push mechanism.
- Amazon SQS exchanges messages through a polling model: sending and receiving components are decoupled.
- Amazon SQS provides flexibility for distributed components of applications to send and receive messages without requiring each component to be concurrently available.