MODULE 13 · STUDY NOTES

Building Decoupled Architectures

Tight versus loose coupling, and the asynchronous messaging services — Amazon SQS, Amazon SNS, and Amazon MQ — that let application components scale and fail independently

A resilient cloud application is one whose pieces do not drag each other down. This module is about **decoupling** — separating the components of a system so they can operate independently — and the AWS services that make it practical. It opens from a cloud architect's three recurring concerns: addressing performance **bottlenecks** as traffic grows, limiting the **blast radius** of a single component's failure, and enabling **changes** to one component without disturbing the others. Tightly coupled designs fail all three tests; loosely coupled designs insert an intermediary between dependent components so they scale and fail on their own. Loose coupling comes in two flavors — synchronous (Elastic Load Balancing between tiers, microservices within an application) and asynchronous (message based). The heart of the module is the asynchronous set: **Amazon SQS** queues for point-to-point work, **Amazon SNS** topics for publish/subscribe fanout, and **Amazon MQ** managed brokers for hybrid and migrated workloads. Throughout, keep the architect's discipline in mind: start from the business need and work backward to the components that fit it.

LEARNING OBJECTIVES

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

13.1 Decoupling your architecture

Decoupling refers to the separation of components in a system so they can operate independently. A cloud architect approaching this layer of a design carries three concerns, and each is a symptom of coupling: addressing **performance bottlenecks** so the architecture scales as traffic increases; **limiting the impact of failures** so one component's failure does not bring down the whole application; and enabling **changes to one component without affecting the others** so maintenance causes minimal downtime. The architect should always start from the business case and work backward to the components that best serve it, revisiting decisions as needs evolve.

Tight coupling means linked components are so dependent on each other that a change or failure in one forces a change or failure in the others. Consider a three-tier web application moved to the cloud — presentation (web server on Amazon EC2), business logic (application server on EC2), and a database (Amazon RDS). Although the tiers are developed independently, their communication is **synchronous**: a source component sends a request and **waits for a response** before proceeding, and the web server talks directly to the app server. The consequences: if the application server or web server goes down, the whole system is down and returns errors to the client; updating the app server for maintenance or a bug fix requires taking the whole system offline; and scaling out a tier means updating code so components know the IP addresses of new instances and when to route traffic to them.

WHY TIGHT COUPLING RESISTS SCALING

- Because related components connect directly, adding one web server to a tightly coupled tier requires **three** new connections — one from Amazon Route 53 and two to the application servers.
- Adding one application server likewise requires three new connections, one from each web server instance.
- The more components you tightly couple, the more connections you must add or remove to scale out or scale in.
- Failures are handled manually — if one application instance goes down, the performance of **every** directly connected web server is affected.

At the **infrastructure level**, the remedy is to introduce an intermediary between dependent layers. A managed intermediary such as **Elastic Load Balancing (ELB)** loosely couples the layers: place an **Application Load Balancer (ALB)** in front of the web servers to distribute traffic from Route 53 and route only to healthy instances, and place a second ALB between the web tier and the application tier to do the same. The intermediary performs automatic workload management and failover routing. Now scaling out to add a web server needs only **two** connections — one to each load balancer — instead of editing code across every related layer.

Tight coupling also happens **inside** an application. When all business functions are built and deployed as a single unit — a **monolithic** application running in one process — a problem in one function can slow or stop all the others, and changing one function forces the whole application into maintenance. If a three-tier app bundles finance, image-transcoding, and calculation functions into one deployment unit, a slowdown in transcoding can degrade or freeze the entire application, and responses to the web tier gradually slow and eventually stop.

A **microservice architecture** divides application functions into parts that scale and fail independently. Each microservice runs in its own process, maintains its own data, and exposes its function through a **well-defined API**; the services are loosely coupled and can communicate synchronously or asynchronously. Splitting the monolith into finance, transcoding, and calculation microservices lets each scale on its own (finance across three containers, calculations in one, because finance handles more requests) and isolates failure — a transcoding container crashing does not affect the finance or calculation services, or even transcoding running in another instance. Microservices are covered in depth in the *Building Microservices and Serverless Architectures* module.

To push resiliency further, make component interactions **asynchronous** wherever an immediate response is not required and an acknowledgment that the request was registered is enough. Asynchronous messaging pairs a **producer** that generates events with a **consumer** that processes them; the two never interact directly but communicate through an **intermediate durable storage layer — a queue or a topic**. In the transcoding example, the microservice becomes a producer: for each request it stores the image in Amazon S3, puts a message on an SQS queue, and immediately acknowledges the user. A separate consumer application polls the queue, transcodes each image (scaling out as needed), and on completion posts to an SNS topic that emails the user. The decoupled components can be traditional functions or microservices.

TABLE 13.1 Categories of loose coupling solutions

CATEGORY	LEVEL	SOLUTION
Synchronous	Infrastructure level	Elastic Load Balancing (ELB) between dependent layers
Synchronous	Application level	Microservice architecture
Asynchronous	Queue based	Amazon SQS
Asynchronous	Topic based	Amazon SNS
Asynchronous	Queue and topic based	Amazon MQ (supports both)

COMMON PITFALL

Loose coupling is not only asynchronous messaging. It solves the same tight-integration problems through **synchronous** solutions (an ELB load balancer between tiers, or a microservice architecture within an application) as well as **asynchronous** ones (Amazon SQS, SNS, and MQ). This module focuses on the asynchronous solutions but you should recognize ELB and microservices as loose coupling too.

13.2 Decoupling applications with Amazon SQS

Point-to-point messaging is an asynchronous model in which the sending application knows the receiver and sends each message to **only one** consumer. The sender is the **producer** — it generates messages and puts them in a **message queue** (a temporary repository for messages waiting to be processed). The receiver is the **consumer** — it retrieves messages from the queue and processes them using a **pull mechanism**: it *polls* the queue periodically and, if a message is available, retrieves and processes it. Messages are usually small (customer records, product orders, invoices, patient records). The flow is simply: the producer puts a message in the queue → the queue stores it until retrieved → the consumer gets it from the queue.

Amazon Simple Queue Service (Amazon SQS) is a **fully managed** message-queuing service that decouples application components so they run and fail independently, acting as a buffer between senders and receivers. It provides highly available, secure, and durable queuing, works at massive scale (billions of messages per day), and stores all queues and messages within a **single AWS Region across multiple redundant Availability Zones** — protecting messages from any single computer, network, or Availability Zone failure. You reach it through the AWS Management Console or programmatically through the AWS SDK APIs.

AMAZON SQS BENEFITS

- **Fully managed** — no messaging software to run or infrastructure to maintain; SQS provisions the compute, storage, and networking automatically, with little configuration.
- **Reliability** — stores messages on multiple servers to deliver large volumes of data without losing messages.
- **Security** — control who can send and receive, and encrypt messages with server-side encryption (SSE) using AWS KMS keys; SQS decrypts a message only for an authorized consumer.
- **Scalability** — messages can be sent and read simultaneously by multiple producers and consumers, scaling elastically with usage and no capacity pre-provisioning.

KEY TERMS

Message
Up to **256 KB** (XML, JSON, or unformatted text); use the SQS Extended Client Library for Java for payloads up to 2 GB. A message stays in the queue until the consumer deletes it or the retention period expires — default **4 days**, configurable up to **14 days**.

Queue
Comes in two types: **standard** and **first-in-first-out (FIFO)**. Configurable parameters include the message retention period, the visibility timeout, and the receive-message wait time (short vs. long polling).

Dead-letter queue (DLQ)
Associated with a source queue; after a message exceeds the maximum number of processing attempts, SQS automatically moves it to the DLQ. A standard queue's DLQ must be standard; a FIFO queue's DLQ must be FIFO.

TABLE 13.2 Standard vs. FIFO queues

	STANDARD QUEUE	FIFO QUEUE
Delivery	At-least-once (a message may arrive more than once)	Exactly-once processing
Ordering	Best-effort — may deliver out of order	First-in-first-out — exact order preserved
Throughput	Nearly unlimited API calls per second per operation	Up to 300 API calls/s per operation, or 3,000 when batching 10 messages
Use when	The consumer tolerates duplicates and out-of-order arrival	The order of messages must be preserved

The **receive-message wait time** parameter controls the polling type. By default a queue uses **short polling** (a wait time of 0): SQS queries only a subset of its servers (by weighted random distribution) and responds immediately even if it finds no messages — fast, but with many empty responses that increase cost. Any nonzero value (up to a maximum of **20 seconds**) enables **long polling**: SQS queries all of the servers and waits until a message is available or the wait time expires. Long polling gives fewer responses but reduces cost by cutting empty receives, and it is preferable in almost all cases. The exceptions: an application that needs an immediate response to each poll (for example, one consuming real-time stock prices), or a single thread polling multiple queues.

The **visibility timeout** is the period during which SQS prevents *other* consumers from receiving and processing a message that one consumer has already received — it keeps the same message from being processed twice. During the timeout, the consumer should process the message and then delete it; if it fails to delete before the timeout expires, the message becomes visible again and might be processed again. Set the timeout to the maximum time your application needs to process and delete a message. The **default is 30 seconds** and the **maximum is 12 hours**; the value can also be overridden at the message level.

> **WORKED EXAMPLE – THE LIFE OF AN SQS MESSAGE**
>
> A producer sends a message to a queue, and SQS distributes it redundantly across the queue's servers. When a consumer is ready, it retrieves the message and the **visibility timeout** starts — the message remains in the queue but is invisible to other consumers. After processing, the consumer **deletes** the message during the timeout window; SQS never deletes it automatically, because in a distributed system there is no guarantee the consumer actually received it (a connectivity or application issue could intervene). If the visibility timeout expires and the consumer never deleted the message, the message becomes visible again to be consumed.

WORKED EXAMPLE – DECOUPLING ORDER CAPTURE FROM FULFILLMENT

A web application captures customer orders and fulfills them. In a monolith both functions live in one unit; introducing an SQS **customer order queue** isolates fulfillment into its own process. The **order capture** app is the producer — three instances behind an ALB all send orders to the queue. The **order fulfillment** app is the consumer — two instances poll the queue, process orders, and delete each message on success; a message that cannot be processed goes to the DLQ. Benefits: the queue **buffers traffic spikes** so the two functions scale independently, work proceeds only as fast as needed to manage cost, and failed orders can be retried or redirected to the DLQ for later reprocessing.

AMAZON SQS USE CASES (AND WHEN NOT TO USE A QUEUE)

- **Work queues** — decouple components that process work at different rates; workers in an Auto Scaling group pull from the queue and scale on workload and latency requirements.
- **Buffering and batch operations** — absorb temporary volume spikes without losing messages, and batch work for bulk processing at a later time.
- **Request offloading** — move slow operations off interactive request paths so the user gets an immediate response (e.g., a bill payment processed in the background).
- **Trigger Amazon EC2 Auto Scaling** — use queue depth as a load signal; a CloudWatch alarm on the queue's `NumberOfMessagesSent` metric can add an instance when messages back up.
- **Not a fit** for selectively receiving specific messages by attribute, or for very large messages — store large payloads in Amazon S3 and pass a reference in the message.

SAMPLE EXAM QUESTION, WORKED

A company implements Amazon SQS for asynchronous processing and wants to keep the number of empty responses from polling requests to a minimum. What should a solutions architect do? Key phrases: **SQS queue**, **empty responses kept to a minimum**. Increasing the message retention period only changes how long messages are kept, not whether polls come back empty. Increasing the redrive-policy maximum receives changes retry behavior on failures, not polling. Increasing the visibility timeout prevents double-processing, not empty responses. The answer is to **increase the receive-message wait time** — that is, enable **long polling** — which lengthens the interval between requests so a message is more likely to have arrived by the next poll.

COMMON PITFALL

Keep the SQS lifecycle straight: a consumer **receiving** a message does not remove it from the queue. The message is merely hidden for the duration of the visibility timeout; the consumer must **delete** it explicitly, or it becomes visible again and may be processed a second time.

13.3 Decoupling applications with Amazon SNS

Publish/subscribe (pub/sub) messaging decouples applications asynchronously when the sender broadcasts to **multiple** receivers about which it knows little or nothing. The sender is the **publisher**; it posts messages to an interim destination called a **topic**. The receivers are **subscribers**; they express interest by subscribing to the topic and then automatically receive every message published to it. Delivery uses a **push mechanism** — messages are pushed out immediately, eliminating any need to poll. Unlike a queue, which stores messages until they are retrieved and deleted, a topic transfers messages with little or no queuing and pushes them to all subscribers, which often perform different functions in parallel. The flow: subscribers subscribe to the topic → the publisher publishes a message → the topic pushes it to all subscribers.

Amazon Simple Notification Service (Amazon SNS) is a fully managed pub/sub messaging service for setting up, operating, and sending notifications from the cloud. You create a **topic** and set policies for who may publish or subscribe; a publisher posts to topics it owns or has permission to use, and SNS delivers each message to every subscriber of that topic. Each topic has a unique name identifying the SNS endpoint. Publishers and subscribers can use **TLS** to secure the channel. SNS provides *durable storage while it processes a message* — it stores multiple copies across Availability Zones before acknowledging the publisher — but **deletes the message after publishing it** to subscribers; unlike Amazon SQS, **SNS does not persist messages**. Access is through the console or the SDK APIs.

AMAZON SNS SUBSCRIBER (ENDPOINT) TYPES

- **Email** — sent to a registered address as text or a JSON object.
- **Mobile text (SMS)** — sent to a registered phone number as an SMS text message.
- **Mobile push** — a native push notification to a mobile device endpoint.
- **HTTP or HTTPS endpoint** — delivered to a URL via an HTTP POST request.
- **AWS Lambda function** — invokes custom business logic to run.
- **Amazon SQS queue** — placed in a queue for a receiver application to process later.
- **Amazon Kinesis Data Firehose delivery stream** — forwarded to storage and analytics endpoints.

SNS suits event notification, workflow systems, mobile applications, and any application that generates or consumes notifications. Common use cases are **application and system alerts** (immediate notification when an event occurs, such as a change to an Auto Scaling group), **push email and text messaging** (targeted news headlines pushed to subscribers by email or SMS), and **mobile push notifications** (for example, telling an application that an update is available, with a link to download and install it).

WORKED EXAMPLE – FANOUT WITH SNS AND SQS

In a **fanout** scenario, one message published to an SNS topic is replicated and pushed to multiple SQS queues, HTTP endpoints, or email addresses for parallel asynchronous processing. A mobile client uploads an image to an S3 bucket to be resized into three variants (thumbnail, mobile, and web). Amazon S3 **Event Notifications** invoke a publish of the object's URL to an SNS topic. The topic **fans out** the message simultaneously to three SQS queues — generate-thumbnail, size-for-mobile, and size-for-web — each observed by its own application running in its own Auto Scaling group. Each application polls its queue, performs its resize independently of the others, and stores the result in a converted-images S3 bucket.

AMAZON SNS CONSIDERATIONS

- Each notification carries a **single** published message, and once a message is delivered there is **no way to recall** it.
- SNS offers **standard** and **FIFO** topics: use a standard topic when delivery order does not matter; use a **FIFO topic** when exact ordering is required (a standard topic can arrive out of order because of network issues).
- A **delivery policy** controls the retry pattern when a subscribed endpoint becomes unavailable; when retries are exhausted, SNS discards the message unless a **dead-letter queue** is attached to the subscription.
- You cannot modify SNS delivery policies except for **HTTP/HTTPS** endpoints, whose retry behavior you can customize to match your server's capacity.

AMAZON SNS	AMAZON SQS
publish/subscribe — push, one-to-many ▪ Model: publisher–subscriber; the publisher need not know anything about the subscribers ▪ One published message reaches many subscribers, which must be available at publication time to receive it ▪ Delivery is **push**; messages are **not persistent** and are deleted after publishing	*point-to-point — pull, one-to-one* ▪ Model: producer–consumer; the producer knows the single consumer ▪ A message goes to one consumer, which need not be available when the message is sent ▪ Delivery is **pull**; messages **persist** until the consumer deletes them or the retention limit is reached

13.4 Decoupling a hybrid application with Amazon MQ

Amazon MQ is a fully managed **message broker** service for **Apache ActiveMQ** and **RabbitMQ**, two popular open-source brokers. A message broker lets different software systems — often written in different programming languages on different platforms — communicate and exchange information. As a managed service, Amazon MQ reduces operational responsibility by handling the provisioning, setup, and maintenance of the broker, and it provides a **queue- and topic-based** solution for loosely coupling applications. It connects to existing applications through industry-standard messaging APIs and protocols — **JMS, NMS, AMQP, STOMP, MQTT, and WebSocket** — so you can migrate from any broker that uses these standards **without rewriting messaging code**; in most cases you simply update your applications' endpoints to connect to Amazon MQ.

Many enterprises rely on message brokers as the backbone connecting distributed systems, and some applications — mainframes, for instance — are too costly to migrate yet must still interact with cloud components. **Amazon MQ enables hybrid environments** by passing messages between on-premises and cloud applications; you can even invoke a **Lambda** function from MQ-managed queues and topics to integrate legacy systems with serverless architectures. In a typical setup, an administrator creates a single **Amazon MQ for ActiveMQ** broker in an Availability Zone and configures it to store messages in an **Amazon EBS** volume optimized for low latency and high throughput (Amazon MQ can also use **Amazon EFS**). An on-premises producer then sends messages to the broker, which propagates them to a consumer running in the AWS Cloud.

TABLE 13.3 Choosing the right decoupling service

FACTOR	AMAZON SQS	AMAZON SNS	AMAZON MQ
Applicability	Cloud-centered apps	Cloud-native apps	Hybrid apps; broker migration
Messaging model	Producer–consumer	Publisher–subscriber	Producer–consumer and publisher–subscriber
Programming API	Amazon SQS API	Amazon SNS API	Industry-standard broker APIs
Pricing model	Pay per request	Pay per request	Pay per hour + pay per GB

The guidance is straightforward. If you need to **integrate on-premises applications with cloud applications**, or you want to **move existing message brokers to the cloud**, choose **Amazon MQ** — it supports open standards such as JMS and AMQP, reduces broker maintenance and licensing costs, improves broker stability, and lets on-premises apps communicate with cloud apps without rewriting messaging code. If you are **building new applications in the cloud**, choose **Amazon SQS and Amazon SNS** — queue and topic services that use APIs and need no broker setup. SQS bills on monthly API requests plus data transfer; SNS bills on monthly API requests, deliveries to endpoints, and data transfer; Amazon MQ bills on broker instance runtime, monthly storage, and data transfer.

COMMON PITFALL

Do not reach for Amazon MQ just because a workload uses messaging. For **new, cloud-native** applications, Amazon SQS and SNS are the recommended, lower-overhead choice. Amazon MQ earns its place specifically when you must bridge **on-premises and cloud** environments, or migrate an **existing** ActiveMQ or RabbitMQ broker to the cloud without rewriting messaging code.

Module summary

- Decoupling separates components so they operate independently; tightly coupled designs bottleneck, share single points of failure, and force system-wide downtime for changes, while loosely coupled designs insert an intermediary so components scale and fail on their own.
- Loose coupling is synchronous or asynchronous: synchronous uses ELB between tiers (infrastructure level) or microservices (application level); asynchronous uses messages through queues or topics — Amazon SQS, SNS, and MQ.
- Amazon SQS is fully managed point-to-point queuing: a producer enqueues messages, and a consumer polls (pull), processes, then explicitly deletes each message. Messages persist (256 KB max; retention 4–14 days) within one Region across multiple Availability Zones.
- SQS queues are standard (at-least-once, best-effort order, near-unlimited throughput) or FIFO (exact order, exactly-once, 300 or 3,000 batched API calls/s). The visibility timeout (default 30 s, max 12 h) hides an in-flight message, and unprocessable messages move to a dead-letter queue.
- Long polling (receive-message wait time up to 20 s) queries all servers and waits, cutting empty responses and cost; short polling is the default but returns immediately, often empty.
- Amazon SNS is fully managed pub/sub: a publisher posts to a topic, and SNS pushes the message to all subscribers (email, SMS, mobile push, HTTP/HTTPS, Lambda, SQS, Kinesis Data Firehose). SNS does not persist messages.
- A fanout pattern sends one message to an SNS topic that replicates it to multiple SQS queues (or endpoints) for parallel processing; SNS also offers standard and FIFO topics and configurable delivery policies with optional DLQs.
- SNS versus SQS: publisher–subscriber vs. producer–consumer, one-to-many vs. one-to-one, push vs. pull, and no persistence vs. persistence.
- Amazon MQ is a managed Apache ActiveMQ or RabbitMQ broker supporting queues and topics and open protocols (JMS, NMS, AMQP, STOMP, MQTT, WebSocket); it loosely couples on-premises and cloud apps and migrates existing brokers without rewriting code.
- Choose SQS and SNS for new cloud-native applications (pay per request); choose Amazon MQ for hybrid integration or broker migration (pay per hour plus per GB).

Review questions

1. What distinguishes a tightly coupled architecture from a loosely coupled one, and what does loose coupling add between components?

Answer: In a tightly coupled system, a change or failure in one component forces changes or failures in others, and scaling requires code updates. Loose coupling reduces those dependencies by inserting an intermediary (ELB, queue, topic, or broker) between components so they scale and fail independently; the intermediary handles failover and scaling automatically.

2. Name the two categories of loose coupling and give an AWS example of each.

Answer: Synchronous — Elastic Load Balancing between tiers (infrastructure level) and a microservice architecture (application level). Asynchronous — message based: Amazon SQS (queues), Amazon SNS (topics), and Amazon MQ (both).

3. In Amazon SQS, trace a message from producer to deletion, and explain the role of the visibility timeout.

Answer: The producer puts a message in the queue, where SQS stores it redundantly. A consumer polls and retrieves it, starting the visibility timeout that hides the message from other consumers. The consumer processes and then explicitly deletes it within the timeout; SQS does not delete automatically. If the timeout expires before deletion, the message becomes visible again and may be processed twice.

4. When should you use a FIFO queue instead of a standard queue, and what does each guarantee?

Answer: Use a standard queue when the consumer tolerates occasional duplicates and out-of-order delivery (at-least-once, best-effort ordering, near-unlimited throughput). Use a FIFO queue when order must be preserved — it guarantees first-in-first-out ordering and exactly-once processing, at up to 300 API calls/s (3,000 when batching 10 messages).

5. A queue produces many empty polling responses and rising cost. What do you change, and how does it help?

Answer: Increase the receive-message wait time above zero (up to 20 seconds) to enable long polling. Long polling queries all of the queue's servers and waits until a message is available or the wait time expires, reducing the number of empty responses and lowering cost.

6. How does Amazon SNS differ from Amazon SQS in messaging model, delivery, distribution, and persistence?

Answer: SNS is publisher–subscriber, pushes messages, distributes one-to-many, and does not persist messages (deleted after publishing). SQS is producer–consumer, is pulled by polling, distributes one-to-one, and persists messages until the consumer deletes them or the retention limit is reached.

7. What is a fanout scenario, and which services implement it?

Answer: A message published to an Amazon SNS topic is replicated and pushed to multiple subscribers — typically several Amazon SQS queues (or HTTP endpoints or email addresses) — so multiple applications process it in parallel. SNS provides the fanout topic and SQS provides the per-consumer queues.

8. When does AWS recommend Amazon MQ over Amazon SQS and SNS?

Answer: Use Amazon MQ to integrate on-premises applications with cloud applications (hybrid environments) or to migrate an existing Apache ActiveMQ or RabbitMQ broker to the cloud without rewriting messaging code. For new cloud-native applications, AWS recommends SQS and SNS, which need no broker setup.