MODULE 10 · STUDY NOTES

Developing with Messaging Services

How queues, pub/sub messaging, and streams decouple applications — Amazon SQS, Amazon SNS, and Amazon Kinesis Data Streams, end to end

The café's coffee inventory is still updated by hand, and Sofía wants it to update itself — receive supplier messages and process them automatically. Mateo, a café regular and AWS consultant, suggests using Amazon SNS to receive supplier messages and an Amazon SQS queue to hold them until they are processed. That small change is really a shift in architecture: instead of one component calling another and waiting for an answer, a **producer** hands a message to a managed service and moves on while a **consumer** picks it up later. This module builds the vocabulary and the services behind that shift. It contrasts synchronous (tightly coupled) with asynchronous (loosely coupled) designs, then works through the three messaging patterns AWS offers as fully managed services — **message queues** with Amazon SQS, **pub/sub** with Amazon SNS, and **data streams** with Amazon Kinesis Data Streams — covering how to send and receive SQS messages, handle failures and polling, fanout and filter with SNS, secure both services, and stream high-volume data for near-real-time analytics.

LEARNING OBJECTIVES

- Illustrate how messaging services — queues, pub/sub messaging, and streams — support asynchronous processing
- Describe Amazon Simple Queue Service (Amazon SQS)
- Send messages to an SQS queue
- Describe Amazon Simple Notification Service (Amazon SNS)
- Subscribe an SQS queue to an SNS topic
- Describe how Amazon Kinesis can be used for real-time analytics

10.1 Processing requests asynchronously

In a **synchronous** model, the client (producer) sends a request to the consumer and then **waits** until the message is processed and a response comes back — a simple interaction common to create, read, update, and delete (CRUD) API actions. It is like ordering a coffee at the counter and getting the cup before you walk away: the action completes before the next person orders. Diagrammed, the client calls service A, service A calls service B and waits for B to finish, and the response travels back to the client. The strong interdependency between producer and consumer makes this a **tightly coupled** system, and its disadvantage is that it is **not fault tolerant** — if any component fails, the whole system fails. If the consumer fails while processing a message, the producer is forced to wait; and if new consumer instances are launched to recover or to keep up with load, the producer must be **explicitly made aware** of them. The coupling is prone to brittleness.

In an **asynchronous** model, the client sends a request and might get only an **acknowledgement** that the event was received — it does not get a response containing the results. It is like placing an order that is delivered later: you get an order number, the next person in line can order without waiting, and someone brings your order when it is ready. Diagrammed, when the client calls service A, service A surfaces the event to its consumers (service B) and moves on. The trade-off is that service B has no channel to pass results back to service A, so you write your applications to fetch results and handle errors asynchronously. The advantage is real: reduced dependencies on downstream activities

improve responsiveness to the client, and the producer keeps generating messages whether or not the consumer is processing them — and is **not impacted if the consumer fails**. The result is much less interdependency, a more **loosely coupled** system, and greater fault tolerance.

SYNCHRONOUS – TIGHTLY COUPLED	ASYNCHRONOUS – LOOSELY COUPLED
producer waits for the result ▪ Client blocks until it receives a response ▪ Common to CRUD-style API actions ▪ New consumer instances must be made known to the producer ▪ A consumer failure stalls the whole system	*producer hands off and moves on* ▪ Client may get only an acknowledgement, not the results ▪ The producer keeps sending whether or not the consumer keeps up ▪ Reduced downstream dependencies improve responsiveness ▪ Looser coupling, greater fault tolerance

This module covers three types of messaging services used for asynchronous processing: **message queues**, **publish/subscribe (pub/sub) messaging**, and **data streams**. In every case, producers send messages to a service and do not wait for a result, and the service provides an **interim location** from which consumers get messages — an asynchronous, decoupled design between components. AWS offers a fully managed service for each: **Amazon SQS** for queues, **Amazon SNS** for pub/sub, and **Amazon Kinesis Data Streams** for streams. The type you choose depends on the use case.

TABLE 10.1 The three messaging patterns

TYPE	AWS SERVICE	HOW IT WORKS	CONSUMER BEHAVIOR
Message queue	Amazon SQS	A producer adds a message; it waits in the queue until a consumer retrieves it	Consumers poll, process, then delete each message
Pub/sub messaging	Amazon SNS	Publishers send to a topic; messages are pushed out as event notifications	All subscribers are pushed the same message; no polling
Data streams	Kinesis Data Streams	High-rate records held in a temporary repository for near-real-time analysis	Multiple consumers read the same records; none are deleted

A **message queue** is a temporary repository for messages waiting to be processed. Messages are usually small — requests, replies, error messages, or plain information such as customer records, product orders, invoices, or patient records. The workflow: a producer adds a message, which is stored until a consumer retrieves it; consumers **poll** the queue to see whether new messages are available; each consumer processes each new message it retrieves; and if processing succeeds, the consumer **deletes** the message so that no other consumer processes the same one.

With **pub/sub messaging**, publishers (producers) send messages to **topics**, which provide a lightweight mechanism to broadcast asynchronous event notifications and expose endpoints that software components connect to. Unlike queue consumers, subscribers do **not** check for messages — messages published to a topic are **pushed out to all subscribers**. This makes it easy to notify large numbers of subscribers about events: publish a message to an `alerts` topic when a certain error occurs, and multiple consumers can each take a different action on it.

Data streams resemble queues — a temporary repository that consumers poll for new messages — but the message rate is typically much higher, and rather than processing each message individually, streams are used to **analyze high volumes of data in near-real time**. Another distinction: multiple consumers can process the same messages and do entirely different things with them (all viewing the same data, as if looking through the same window), and consumers

do not delete messages from a stream. Streams suit rapid, continuous data intake and aggregation — IT infrastructure logs, social media data, market data feeds, or web clickstream data.

SECTION 2 KEY TAKEAWAYS

- An asynchronous design reduces interdependencies and can improve responsiveness to the client.
- Messaging services are good for asynchronous designs and include queues, pub/sub messaging, and streams.
- AWS provides fully managed messaging services: Amazon SQS, Amazon SNS, and Kinesis Data Streams.

10.2 Introducing Amazon SQS

Amazon Simple Queue Service (Amazon SQS) is a fully managed message queuing service. With it you can decouple and scale microservices, distributed systems, and serverless applications, because the service **eliminates the complexity and overhead** of managing and operating message-oriented middleware so you can focus on business logic. You can transmit any volume of data at any level of throughput **without losing messages** or requiring other services to be available; decoupled components run and fail independently, which increases overall fault tolerance. Multiple copies of every message are stored **redundantly across multiple Availability Zones**, and the service scales elastically with demand, so there is no capacity planning or pre-provisioning — and no limit on the number of messages per queue.

AMAZON SQS AT A GLANCE

- **Fully managed** — removes the complexity and overhead of running message-oriented middleware
- **Durable** — stores multiple copies of every message redundantly across multiple Availability Zones
- **Elastic** — scales dynamically with demand; no capacity planning, and no limit on messages per queue
- **Secure** — server-side encryption (SSE) via AWS KMS, with key usage logged to AWS CloudTrail
- **Two queue types** — standard (default) and First-In-First-Out (FIFO)

For sensitive data, Amazon SQS offers **server-side encryption (SSE)** that encrypts each message body. SSE integrates with **AWS Key Management Service (AWS KMS)** so you can centrally manage the keys that protect SQS messages alongside the keys that protect your other AWS resources, and AWS KMS logs every use of your encryption keys to **AWS CloudTrail** to help meet regulatory and compliance needs. Amazon SQS offers both **standard** and **FIFO** queues to support different processing priorities.

TABLE 10.2 Standard and FIFO queue types, with use cases

PROPERTY	STANDARD QUEUE (DEFAULT)	FIFO QUEUE
Ordering	Best-effort ordering — generally in order, some out of order	Message order is preserved
Delivery	At-least-once delivery (a copy may arrive more than once)	Exactly-once processing (no duplicates)
Throughput	Nearly unlimited transactions per second (TPS)	More limited TPS
Example use cases	Upload media while resizing/encoding it; process a high number of credit-card validations; schedule non-order-dependent database entries	Record a deposit before a withdrawal; process a credit-card transaction only once; require account registration before course enrollment

DECOUPLING COMPONENTS WITH SQS

A common architecture uses a queue to decouple an image-processing pipeline. A user uploads a photo to an Amazon S3 bucket, which invokes an AWS Lambda function; the Lambda function sends a message about the image to an SQS queue. A processing server in an Amazon EC2 Auto Scaling group **polls** the queue, processes the photo, and sends the result back to S3. The Auto Scaling group scales on the **size of the queue** — servers keep working until the queue is empty, then wait for new messages. If the EC2 instances become unavailable, the Lambda function keeps putting messages on the queue; when the Auto Scaling group is ready again, it resumes polling. Neither side has to know the state of the other — that is the point of decoupling.

10.3 Working with Amazon SQS messages

The lifecycle of an SQS message has three steps. A **producer** sends a message to the queue, and Amazon SQS redundantly stores it across multiple SQS servers. A **consumer** retrieves the message, and Amazon SQS starts the **visibility timeout** — during which no other consumer can pick the message up. The consumer processes the message and then **deletes** it during the visibility timeout. If the message is not processed and deleted before the timeout expires, another consumer might pick up and process the same message. These actions map to the `SendMessage`, `ReceiveMessage`, and `DeleteMessage` API calls, used programmatically or from the console.

TABLE 10.3 SQS message lifecycle operations

API OPERATION	WHAT IT DOES	KEY DETAIL
SendMessage	Delivers a message to a specific queue	MessageBody (max 256 KB) and QueueUrl required; response returns an MD5 digest and a MessageId
ReceiveMessage	Retrieves up to 10 messages	MaxNumberOfMessages sets the count; each message returns a receipt handle and starts the visibility timeout
DeleteMessage	Removes a processed message from the queue	Uses the most recent receipt handle; DeleteMessageBatch removes up to 10 at once

THE SENDMESSAGE OPERATION

An AWS CLI call looks like `aws sqs send-message --queue-url <url> --message-body "My first message"`. The required parameters are `MessageBody` (maximum string size 256 KB) and `QueueUrl`. Optionally you can add up to **10** `MessageAttribute` entries — structured metadata such as timestamps, geospatial data, signatures, and identifiers, each a name, type, and value, sent alongside but separate from the body — and `DelaySeconds` to delay a specific message. The response returns an `MD5OfMessageBody` digest, which you can use to verify the message was received correctly, and a system-assigned `MessageId` for identifying the message.

The `ReceiveMessage` operation retrieves one or more messages (up to 10, via `MaxNumberOfMessages`). For each message returned, Amazon SQS includes a **receipt handle**, which is associated with the **act of receiving** the message — not with the message itself. If you receive the same message more than once, you get a **different receipt handle each time**. You need the most recent receipt handle to delete the message or change its visibility on the queue.

Immediately after a message is received, Amazon SQS sets a **visibility timeout** — a period during which other consumers cannot receive and process that message. Before it expires, the consumer is expected to **process AND delete** the message. Amazon SQS does not auto-delete it because, as a distributed system, there is no guarantee the consumer actually received and handled it (a connectivity issue, or a bug in the consumer). The visibility timeout **defaults to 30**

seconds, with a minimum of 0 seconds and a maximum of 12 hours; use the `VisibilityTimeout` parameter to change it for a `ReceiveMessage` request. If the consumer does not delete the message in time, it becomes visible again, and Amazon SQS increments a **receive counter** each time — by default keeping the message available until its retention period expires, unless you change that behavior with a dead-letter queue.

> **COMMON PITFALL**
>
> Receiving a message is not the same as deleting it. `ReceiveMessage` only hides the message for the duration of the visibility timeout; the consumer must call `DeleteMessage` (with the latest receipt handle) to remove it. Miss the timeout window and the message reappears and can be processed a second time — which is exactly why standard-queue consumers must be built to tolerate duplicates.

For messages that cannot be processed and deleted, configure a **dead-letter queue (DLQ)**. A DLQ helps you troubleshoot faulty transmissions and stops a queue from endlessly retrying a corrupted message. The DLQ receives messages from the source queue after the **maxReceiveCount** — set in the source queue's **redrive policy** — is reached. A DLQ is an ordinary SQS queue: you send and receive from it like any other, and create it from the API or console. To configure it, create the queue to serve as the DLQ, then on the source queue configure a redrive policy that selects the DLQ and sets the maximum receives. For example, with `maxReceiveCount` = 2, message A moves to the DLQ after the second consumer fails, rather than becoming visible for a third time — and it is no longer available on the source queue.

SHORT POLLING (DEFAULT)	LONG POLLING
WaitTimeSeconds = 0	*WaitTimeSeconds = 1–20*
▪ Samples a subset of SQS servers (weighted random distribution) ▪ Returns immediately with only the messages on the sampled servers ▪ With few messages you may get fewer than requested — or none on a given call ▪ Repeated calls eventually sample every server	▪ Queries all SQS servers and waits until a message is available ▪ Eliminates empty responses and false empty responses ▪ Reduces the cost of using Amazon SQS ▪ Enable by setting a non-zero WaitTimeSeconds

> **A LONG-POLLING REQUEST**
>
> A `ReceiveMessage` query against `testQueue` with `WaitTimeSeconds=10` enables **long polling** with a 10-second wait; `MaxNumberOfMessages=5` returns up to five messages per call; and `VisibilityTimeout=15` hides those messages from other consumers for 15 seconds while they are processed.

To stop a message from being received again after the visibility timeout, the consumer must **delete** it. `DeleteMessage` removes one message (selected by its receipt handle), and `DeleteMessageBatch` removes up to 10 (given a list of receipt handles); you must supply the **most recently received** receipt handle. Separately, Amazon SQS **automatically deletes** messages older than the queue's **message retention period** — the default is **4 days**, and you can set it from **60 seconds to 1,209,600 seconds (14 days)** with `SetQueueAttributes`.

SECTION 4 KEY TAKEAWAYS

- Developers use SendMessage, ReceiveMessage, and DeleteMessage to add, retrieve, and delete queue messages.
- Amazon SQS makes a retrieved message invisible for the duration of the visibility timeout.
- Amazon SQS can send failed records to a dead-letter queue for separate processing.
- Short polling samples Amazon SQS servers for messages, while long polling queries all servers.

10.4 Configuring Amazon SQS queues

Standard queues are the default type and support a nearly unlimited number of transactions per second per action. Because of the highly distributed architecture that enables that throughput, you must account for two conditions. First, **ordering is not guaranteed** — standard queues provide best-effort ordering, so if a target application needs messages in the order received, add logic in the consumer to reorder them (for example, by each message's timestamp). Second, **more than one copy of a message might be delivered** — standard queues support at-least-once, not exactly-once, delivery, so your code must check for duplicates and process a message only the first time. To verify uniqueness of the record on the queue, use the SQS-generated `MessageID`; to verify uniqueness of the **payload** (an upstream producer sending the same record twice yields two different MessageIDs but the same content), use `md5OfBody`, a hash of the payload.

To avoid writing that ordering and deduplication code where order is critical and duplicates cannot be tolerated, use a **FIFO queue**. FIFO queues provide **exactly-once processing** (avoiding sending duplicates) and **guarantee message ordering**. They also support **message groups**, which allow multiple ordered message groups within a single queue. FIFO queues have more limited TPS throughput than standard queues, so choose a FIFO queue **only if you need its features**.

SAMPLE EXAM QUESTION, WORKED

A developer wants an asynchronous connection that processes individual banking transactions — deposits and withdrawals — that must be handled in the order they arrive. The key phrases are *individual transaction processing* and *the order*. Processing **individual messages** points to a queue rather than a Kinesis stream (which derives value from aggregating many messages), and configuring Amazon ECS on EC2 is not a messaging service at all. Between the two queue choices, only a **FIFO** queue guarantees in-order processing — so the answer is a **FIFO SQS queue**.

TABLE 10.4 Queue management API operations

API OPERATION	DESCRIPTION
CreateQueue	Creates a queue and its attributes; any attribute you don't provide takes its default value
GetQueueAttributes	Reads queue attributes — e.g., approximate messages available versus currently invisible — to estimate the resources needed to process them
SetQueueAttributes	Sets attributes on an existing queue (for example, the retention period, or WaitTimeSeconds on the queue itself)
GetQueueUrl	Retrieves the URL of a queue
ListQueues	Retrieves a list of your queues
DeleteQueue	Deletes a queue whether or not it is empty; its messages are then no longer available

CREATING A QUEUE FROM THE CLI

`aws sqs create-queue --queue-name MyQueue --attributes file://create-queue.json` creates a queue from an attributes file. In that file, a `RedrivePolicy` sends failed records to a dead-letter queue (MyDeadLetterQueue) after 1,000 attempts (`maxReceiveCount`), and `MessageRetentionPeriod` is set to 259200 seconds (3 days). If a message remains on the source queue for more than 3 days, it is discarded **regardless of** the maxReceiveCount.

There are **three ways to secure** Amazon SQS resources. For **identity and access management**, access requires credentials with permissions to SQS queues and messages; you use **IAM policies** and **Amazon SQS policies** (SQS has a resource-based permission system written in the same language as IAM, so you can achieve similar results and combine them) — with one crucial difference: you **must use SQS policies to grant permissions to other AWS accounts**. For **data encryption**, SSE protects message bodies with keys managed by AWS KMS; note that encryption applies at a point in time — if messages 1–10 are already on the queue when you enable SSE, message 11 is encrypted but 1–10 are not. For **internetwork traffic privacy**, a VPC endpoint lets you send messages to your queues without crossing the public internet.

IAM POLICY	AMAZON SQS POLICY
attached to the identity (user Bob) ▪ Grants Bob actions such as ReceiveMessage and SendMessage on MyQueue ▪ Restricts access only to users within your own AWS account	*attached to the resource (the queue)* ▪ Names Bob and grants the same ReceiveMessage/ SendMessage actions on MyQueue ▪ Resource-based — the only way to grant access to other AWS accounts

SECTION 5 KEY TAKEAWAYS

- Standard queues provide nearly unlimited throughput but don't preserve order and might include duplicates.
- FIFO queues preserve order and provide exactly-once delivery but have more limited throughput.
- Queue security covers managing access to resources, protecting stored data, and limiting internet exposure.

10.5 Introducing Amazon SNS

Amazon Simple Notification Service (Amazon SNS) is a highly available, durable, secure, fully managed **pub/sub** messaging service for many-to-many messaging between distributed systems, microservices, and event-driven applications. Its topics provide **high-throughput, push-based, many-to-many** messaging, and it decouples the pieces of your system: SNS **offloads message filtering logic from your subscribers** and **message routing logic from your publishers**. It includes substantial **retry policies** to limit delivery failures and a **dead-letter queue** option for messages that keep failing, scales dynamically, and offers a **FIFO topics** option that works with FIFO SQS queues to preserve message order.

AMAZON SNS AT A GLANCE

- **Pub/sub** — topics provide high-throughput, push-based, many-to-many messaging
- **Offloads logic** — message filtering away from subscribers, message routing away from publishers
- **Reliable** — substantial retry policies, plus a dead-letter queue option for messages that keep failing
- **FIFO topics** — pair with FIFO SQS queues to preserve order without custom ordering/dedup code

An **SNS topic** is a logical access point that acts as a communication channel. Instead of putting a specific destination address in each message, a **publisher sends the message to the topic**; Amazon SNS matches the topic to its list of subscribers and delivers the message to each one. Each topic has a **unique name** identifying the SNS endpoint where publishers post and subscribers register. A publisher can send to topics it created or has permission to publish to, and when you create a topic you control access with policies that determine which publishers and subscribers can communicate with it. To receive messages, you **subscribe an endpoint** to the topic; all subscribers to a topic receive the **same messages**, and SNS formats each message based on the endpoint type.

TABLE 10.5 Subscriber endpoint types

MESSAGING TYPE	SUBSCRIBER ENDPOINTS
Application-to-application (A2A)	Lambda functions, SQS queues, Kinesis Data Firehose delivery streams, and HTTP(S)
Application-to-person (A2P)	Email, text messaging (SMS), and mobile push notifications

> **THE FANOUT PATTERN**
>
> When a message must be processed by **more than one consumer**, combine pub/sub with queues in a **fanout** pattern: an SNS topic receives a message that is then pushed to multiple subscribers for parallel processing. Several SQS queues subscribe to a `new-order` topic; each queue gets an **identical** copy but does its own asynchronous work — one hands messages to an order-fulfillment application, another to a data-warehousing application. Subscribers can mix endpoint types (HTTP endpoints, email addresses). Fanout can also replicate production data to development: subscribe a DEV queue to the same topic and route it through a Lambda function that **anonymizes** records before writing them to a development Amazon DynamoDB table.

> **IMAGE-PROCESSING FANOUT**
>
> A user uploads images to an S3 bucket; a message about the event is **published to an SNS topic**. Multiple SQS queues are subscribed to that topic, and when each receives the message it invokes a Lambda function with the published payload. The Lambda functions process the images differently — generate thumbnails, size for mobile, size for web — and send the results to another S3 bucket. One event, several independent outcomes, all in parallel.

10.6 Developing with Amazon SNS

When a publisher sends a message to a topic, Amazon SNS returns a **message ID** and then attempts to deliver the message to **all** subscriber endpoints. SNS defines a **delivery policy** for each delivery protocol, which specifies how it **retries** delivery when server-side errors occur (the host of a subscribed endpoint becomes unavailable). When the delivery policy is exhausted, SNS stops retrying and **discards the message** — unless a **dead-letter queue** is attached to the subscription, in which case the failed message is captured there.

TABLE 10.6 Common Amazon SNS API operations

API OPERATION	PURPOSE	INPUT → OUTPUT
CreateTopic	Creates a topic where notifications can be published	Topic name → topic ARN
Subscribe	Prepares a subscription and sends a confirmation to the endpoint	Endpoint, protocol, topic ARN → subscription ARN (or pending)
ConfirmSubscription	Confirms the endpoint owner's intent to receive messages	Token from the confirmation message → subscription ARN
Publish	Sends a message to all of a topic's subscribed endpoints	Message (+ optional attributes, subject) and topic ARN → message ID
DeleteTopic	Deletes a topic and all of its subscriptions	Topic ARN → (none)

Subscription is a two-step handshake: `Subscribe` sends a confirmation message to the endpoint, and the endpoint owner must call `ConfirmSubscription` with the **token** from that message to actually create the subscription. Once confirmed, `Publish` saves the message (returning its ID) and SNS attempts delivery to every subscriber.

By default, a subscriber receives **every** message published to a topic. To receive only a subset, a subscriber assigns a **filter policy** to its subscription — a simple JSON object of attributes that define which messages it should receive. When you publish a message, Amazon SNS compares the **message attributes** to the attributes in each subscription's filter policy; **if any match**, SNS delivers the message, otherwise it skips that subscriber. A subscription with **no** filter policy receives every message. Filtering lets one topic serve many subscribers that each care about different events, without the publisher having to route them.

> **A FILTER POLICY IN ACTION**
>
> An online buyer visits a page, cancels one order, and places another; all three events publish to a `Shopping Events` topic. The **Payment** SQS queue's subscription carries a filter policy for the order messages, so it ignores the page visit; a **Lambda** subscription filters on `{"event_type": ["product_page_visited"]}`. When a message publishes carrying the attribute `event_type = product_page_visited`, only the Lambda subscription matches — the Lambda function is notified and the Payment queue is not.

As with Amazon SQS, there are **three ways to secure** Amazon SNS. For **identity and access management**, SNS integrates with IAM so you can specify which SNS actions a user may perform and on which topics (for example, granting the `Publish` action on specific topics) — but an IAM policy restricts access only to users **within your AWS account**, whereas an **Amazon SNS policy** on a topic can grant access to **other AWS accounts** or to your own users, and controls who can publish to and subscribe to that topic. For **data encryption**, SSE protects the contents of messages in topics with keys managed in AWS KMS. For **internetwork traffic privacy**, a VPC endpoint gives you a private connection so you can publish to topics without sending messages over the public internet.

> **SECTIONS 6 AND 7 KEY TAKEAWAYS**
>
> - Developers publish messages to SNS topics, and every subscriber to that topic gets a copy of all published messages.
> - With attribute filtering, subscribers receive only relevant messages.
> - Subscriber endpoints include both application-to-application (A2A) and application-to-person (A2P) types.
> - SNS security covers managing access to resources, protecting data sent to topics, and limiting internet exposure.

10.7 Introducing Kinesis Data Streams

Amazon Kinesis Data Streams is a fully managed, massively scalable, durable **real-time data streaming** service you use to **ingest, buffer, and process** streaming data in real time. It can continuously capture **gigabytes of data per second** from hundreds of thousands of sources — website clickstreams, database event streams, financial transactions, social media feeds, IT logs, and location-tracking events — and makes the data available in **milliseconds** for real-time analytics. It reduces the overhead of building a streaming application with tools such as the AWS SDK, the Kinesis Client Library (KCL), connectors, and agents, and offers built-in integrations with **AWS Lambda**, **Kinesis Data Analytics**, **Kinesis Data Firehose**, and the **AWS Glue Schema Registry**.

KINESIS DATA STREAMS USE CASES

- **Log and event data collection** — process data, generate metrics, power live dashboards, emit aggregated data to stores like Amazon S3
- **Real-time analytics** — run analytics on high-frequency event data such as sensor data
- **Mobile data capture** — push application data from hundreds of thousands of devices as soon as it is produced
- **Gaming data feed** — continuously collect player-game interactions and feed them into your platform

A **producer** collects data and puts it on the stream — a web server sending log data is a producer — built with the Kinesis Producer Library (KPL) or the AWS SDK for Java. Kinesis stores a unit of data as a **data record** (a partition key, a sequence number, and a data blob), and a **data stream** is a group of data records distributed into **shards**. The **partition key** groups records by shard; the **sequence number** is unique per partition key within its shard and is assigned by Kinesis. A shard's capacity is fixed: for **writes**, up to **1,000 records per second or 1 MB/s**; for **reads**, up to **5 transactions per second or 2 MB/s**. You specify the number of shards when you create a stream, the total stream capacity is the **sum of its shards**, and the service adds shards to scale horizontally. A record stays available for a per-stream **retention period**. A **consumer** polls the stream and processes records; you can have **multiple consumers** that share the stream's read throughput, and an **enhanced fanout** option gives each consumer its own read-throughput allotment. Crucially, consumers **do not delete** records — each maintains a pointer to where it is on the stream.

COMMON PITFALL

Unlike a queue, a stream is not drained. Kinesis stream consumers never delete records; each keeps its own pointer and reads independently, so several consumers can process the very same records for different purposes. Records leave the stream only when the retention period elapses — not because a consumer processed them.

WAYS TO BUILD A STREAM CONSUMER

- Build your own application with the Kinesis Client Library (KCL) and deploy it to Amazon EC2
- Configure a Lambda function as a serverless consumer and let Lambda handle stream consumption
- Use Kinesis Data Firehose to deliver records to destinations like Amazon S3, Amazon Redshift, Amazon ES, and Splunk
- Send records to a Kinesis Data Analytics application to process and analyze with SQL, Java, or Scala

TABLE 10.7 Other Kinesis data streaming services

SERVICE	WHAT IT DOES	REMEMBER
Kinesis Data Firehose	Delivers streaming data to data stores (S3, Redshift, Amazon ES, Splunk, HTTP endpoints) with no consumer app to write and no shards to manage	Auto-converts to open formats like Apache Parquet and ORC
Kinesis Data Analytics	Runs real-time analysis on the stream with SQL before persisting; aggregates over a sliding window; can preprocess with Lambda and detect anomalies with ML	Apache Flink, Java, Scala, or Python

A STREAMING ANALYTICS PIPELINE

Customer-interaction logs from applications on Amazon EC2, Amazon ECS, and Lambda all act as **producers** writing to one Kinesis data stream. **Kinesis Data Firehose** consumes the stream and delivers it to Amazon ES (operational insights) and Amazon S3 (queryable with Amazon Athena, with no ETL processor to build). At the same time, **Kinesis Data Analytics** consumes the same stream to compute metrics, percentiles, and derived values, which are emitted to **Amazon CloudWatch** as part of an overall monitoring solution.

TABLE 10.8 Selected Kinesis Data Streams API operations

API OPERATION	DESCRIPTION
CreateStream / DeleteStream	Creates a stream (requires ShardCount and StreamName) or deletes a named stream and its shards
DescribeStreamSummary	Provides summary information about a stream — for example, whether it is active
UpdateShardCount / MergeShards / SplitShards	Change stream capacity by adjusting, merging, or splitting shards
PutRecord / PutRecords	Producers write a record or records; a partition key determines the target shard
GetShardIterator / GetRecords	Consumers get a shard position, then read records starting from it

TABLE 10.9 Choosing among KPL, KCL, and the AWS SDK

CAPABILITY	KPL	KCL	SDK
Abstracts stream management so you focus on application logic	Yes	Yes	No
Producer tasks: automatic retries; aggregating records to optimize throughput	Yes	No	No
Consumer subtasks: load balancing and checkpointing processed records	No	Yes	No
Direct interaction with the Kinesis API to write producers or consumers	No	No	Yes
Requires your application code to handle stream management	No	No	Yes

TABLE 10.10 Queues versus streams

DIMENSION	QUEUES (AMAZON SQS)	STREAMS (KINESIS)
Where the value is	Processing each individual message	Aggregating many messages into actionable data
Message rate	Variable	Continuous and high volume
Message processing	Deleted after one consumer succeeds	Read in parallel by many consumers; each keeps a pointer, none delete
Example use cases	Financial transactions, product orders	Clickstream data, application logs

SECTION 8 KEY TAKEAWAYS

- With Kinesis Data Streams, you can ingest, buffer, and process streaming data in real time.
- Kinesis Data Firehose and Kinesis Data Analytics simplify common data streaming use cases.
- Producers put records on a stream (stored in shards), and consumers get records off for processing.
- The Kinesis Producer Library (KPL) and Kinesis Client Library (KCL) abstract stream interactions for developers.

Module summary

- An asynchronous design decouples producer from consumer: the producer hands off a message and moves on, so a consumer failure doesn't stall the system — more fault tolerant and responsive than a tightly coupled synchronous call chain.
- Three patterns map to three fully managed services: message queues (Amazon SQS), pub/sub (Amazon SNS), and data streams (Kinesis Data Streams). Queues delete each processed message; SNS pushes copies to all subscribers; streams let many consumers read the same records without deleting.
- Amazon SQS stores messages redundantly across Availability Zones, scales elastically with no capacity planning, and secures data with SSE and AWS KMS (key usage logged to CloudTrail).
- Standard queues give best-effort ordering, at-least-once delivery, and nearly unlimited throughput; FIFO queues give strict ordering and exactly-once processing at lower throughput — choose FIFO only when order or deduplication is critical.
- The SQS lifecycle is SendMessage → ReceiveMessage (which starts the visibility timeout, default 30s, range 0s–12h) → DeleteMessage; a message not deleted before the timeout reappears for another consumer, and default message retention is 4 days (60s–14 days).
- A dead-letter queue is an ordinary queue named in the source queue's redrive policy; messages move there after maxReceiveCount failed attempts. Short polling samples some servers; long polling (WaitTimeSeconds 1–20) queries all of them to cut empty responses.
- Amazon SNS is pub/sub: publishers post to a topic and SNS pushes to every subscriber — A2A (SQS, Lambda, Firehose, HTTP/S) and A2P (email, SMS, push). Subscribe SQS queues to a topic to fanout one event to parallel consumers.
- SNS filter policies (JSON attributes on a subscription) deliver only matching messages; SNS retries failed deliveries per a delivery policy and can attach a DLQ to a subscription. Subscribing requires a Subscribe → ConfirmSubscription handshake.

- Both SQS and SNS secure resources three ways: IAM plus resource-based policies (required for cross-account access), SSE with AWS KMS, and VPC endpoints for private traffic.
- Kinesis Data Streams ingests gigabytes/second into shards keyed by a partition key (writes 1,000 records/s or 1 MB/s; reads 5 tx/s or 2 MB/s); consumers keep a pointer and never delete records. Kinesis Data Firehose delivers to data stores and Kinesis Data Analytics runs SQL on the stream.
- Use a queue when the value is in each individual message; use a stream when the value comes from aggregating a continuous, high-volume feed. Build stream producers and consumers with the KPL and KCL (or the AWS SDK).

Review questions

1. **Why is a synchronous, tightly coupled system not fault tolerant?**

   Answer: The producer and consumer are strongly interdependent, so if any component fails the whole system fails. The producer must wait for the message to be processed, and it must be explicitly made aware of any new consumer instances.

2. **Match each messaging pattern to its AWS service, and say how the consumer gets messages.**

   Answer: Message queue → Amazon SQS (consumers poll, process, then delete each message); pub/sub → Amazon SNS (messages are pushed to all subscribers, no polling); data streams → Kinesis Data Streams (multiple consumers poll and read the same records without deleting them).

3. **A consumer received a message but crashed before deleting it. What happens, and what governs it?**

   Answer: When the visibility timeout expires (default 30 seconds; range 0s–12h) the message becomes visible again and another consumer can process it. If it keeps failing, a redrive policy moves it to a dead-letter queue after maxReceiveCount attempts.

4. **When should you choose a FIFO queue over a standard queue, and what do you trade away?**

   Answer: Choose FIFO when order is critical or duplicates can't be tolerated (a deposit before a withdrawal, no double charge). FIFO provides strict ordering and exactly-once processing but has more limited throughput than a standard queue's nearly unlimited TPS.

5. **How does the SNS fanout pattern let several systems process one event in parallel?**

   Answer: Publish the event to an SNS topic and subscribe multiple SQS queues (or other endpoints) to it. SNS pushes an identical copy to each queue, and each consumer performs its own asynchronous processing.

6. **A subscriber should receive only some of a topic's messages. How do you arrange that?**

   Answer: Attach a filter policy — a JSON object of attributes — to the subscription. On publish, SNS compares the message attributes to each subscription's filter policy and delivers only when they match; a subscription with no filter policy receives every message.

7. **In Kinesis Data Streams, what is a shard, what decides which shard a record lands in, and what are a shard's limits?**

   Answer: A shard is a stream's capacity unit; the partition key groups records by shard. A shard supports writes up to 1,000 records/s or 1 MB/s and reads up to 5 transactions/s or 2 MB/s, and total stream capacity is the sum of its shards.

8. The exam scenario: an asynchronous connection processing individual banking transactions (deposits and withdrawals) that must be handled in order — standard SQS, Kinesis, FIFO SQS, or ECS on EC2?

Answer: A FIFO SQS queue. Processing individual transactions points to a queue rather than a stream (which aggregates), and in-order, no-duplicate handling requires FIFO rather than standard.