AWS SQS and SNS

CS516 – Cloud Computing

Computer Science Department

Maharishi International University

Maharishi International University - Fairfield, Iowa

Content

- Different models of messaging
  - Sync req-res
  - Point-to-point (SQS)
  - Async message bus (SNS and EventBridge)
- SQS
- SNS
- EventBridge
- Event-driven architecture
- Circuit breaker design pattern

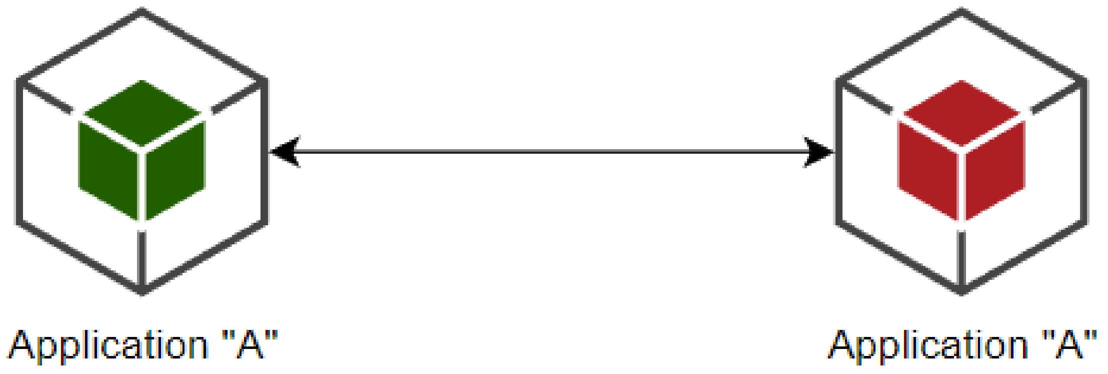
Sync req-res model

- Application “A” calls Application “B” synchronously.
- Advantage:
  - Simple, everyone knows. Straightforward and naïve. It is probably something you did in class or at work such as storing data from the back-end in the DB.
- Disadvantages:
  - If the application “B” is not working, all messages from the app “A” fails. Hence it is not fault-tolerant which costs thousands of dollars in a minute in business.

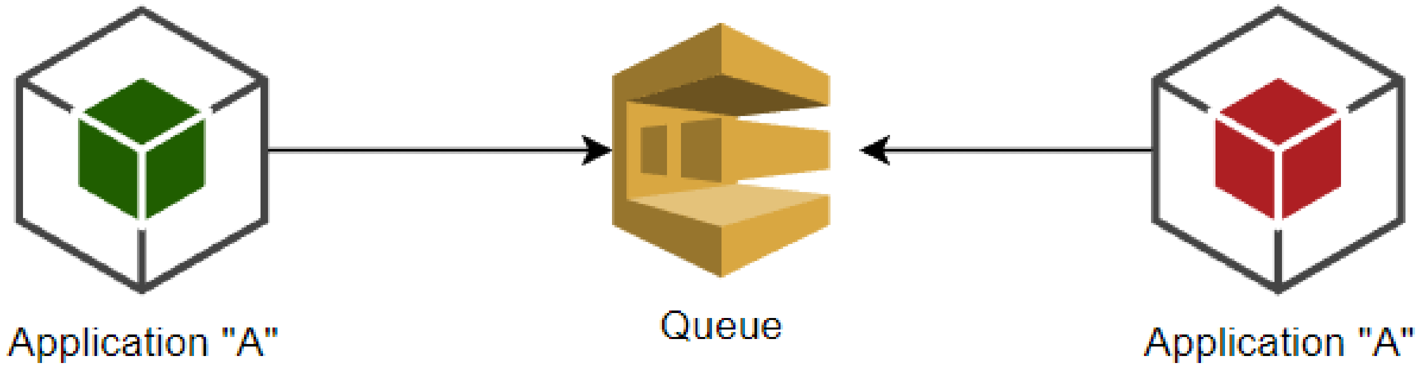
Asynchrony

- In software engineering, effective patterns are derived from real-life patterns in nature!
- Dr. Werner Vogels, Amazon CTO "The world itself is asynchronous".
- What if we make the sync req-res model async? What are the benefits of asynchrony?
  - **Lower latency**
  - **Better fault tolerance**
  - **Throughput management** on the consumer side (the application "B").
- How do we make the sync req-res model async? What technologies do we use?
  - Use integration services such as SNS, SQS. This is known as the **Point-to-point** model.
- Asynchronous leads to **loosely-coupled** (decoupled) applications. Which leads to powerful **evolvable** applications like Amazon S3. Initially, S3 had 8 microservices. Now it has 238+ microservices.

Sync req-res model



Point-to-point model



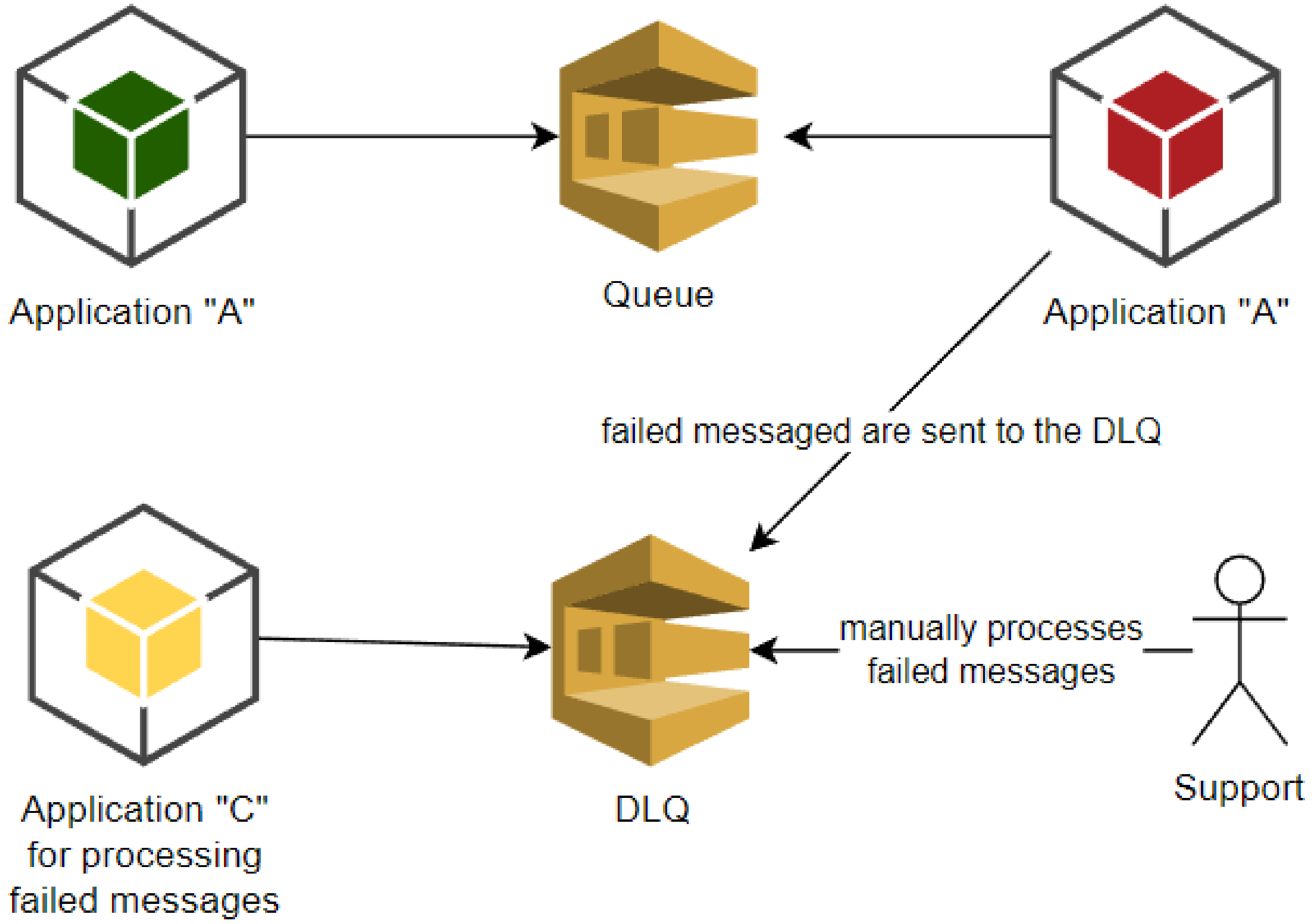
Point-to-point model

- Also known as fire and forget. The producer sends the message to the queue then it returns a success message immediately. The producer doesn’t care how the message is processed. The consumer is responsible for processing the message and returning success/error.
- The application sending the message (in this case the app “A”) to a queue is known as a **Producer**. The other app reading the message from the queue is known as a **Consumer**.
- The consumer **pulls** the messages from the queue at its convenient rate.
- Again, the advantages of this model is lower latency, better fault tolerance, and throughput management on the consumer.

Disadvantages of the point-to-point model

- Response correlation is required. You probably need a correlation id then. From now on, responses don't always have to send a result back, it can send an acknowledge. That improves speed drastically. You can have a different flow to return a result later asynchronously.
- The producer code gets harder to maintain over time when there are new consumer applications such as apps “C”, “D” and so on. That is when the **async message bus** model comes into the picture. In AWS, SNS and EventBridge provide an async event bus.
- Before hopping onto the async message bus model, how does the point-to-point model handle the failed messages?
  - Put the failed messages into another queue known as a Dead Letter Queue (DLQ). Then you can write another application to process those messages or the support can process the messages manually.

Point-to-point model with a DLQ



Async event bus model

Point-to-point model (SQS)	Async event bus model (SNS and EventBridge)
Consumers **pull** messages from the queue.	An event bus **pushes** messages to the consumers.
Producer could end up having complex code for sending message to multiple types of consumers.	Producer is simple. Just send the messages to the bus. Consumer is responsible for reading messages from the bus that they are interested in with **rules** or **filters**.
One message is processed by one consumer as its name suggests.	One message is sent to multiple consumers at the same time. This is also known as **Fanout** in messaging**.**

Simple Queue Service - SQS

SQS offers a secure, durable, and available hosted queue that lets you **integrate** and **decouple** distributed software systems and components (microservices).

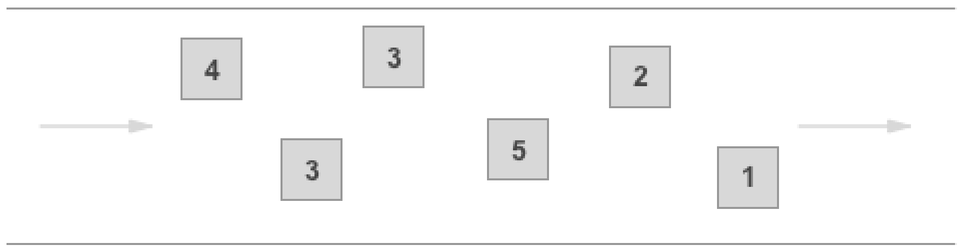
SQS and SNS provides nearly **unlimited scalability**. There are 2 types of queues, standard and FIFO.

Check out this video "AWS SQS Overview For Beginners" by the best channel on serverless technologies.

Standard queue

Use it when the throughput is important. Standard queues provide:

- **Unlimited throughput** - Standard queues support a nearly unlimited number of API calls per second. API calls include SendMessage, ReceiveMessage, DeleteMessage.
- **At-least-once delivery** - A message is delivered at least once, but occasionally more than one copy of a message is delivered.
- **Best-effort ordering** - Occasionally, messages are delivered in an order different from which they were sent.

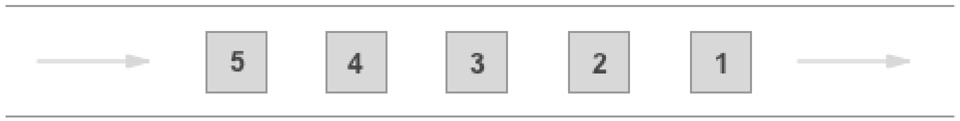


FIFO queue

Use it when the order of events is important. FIFO queues guarantee consistency and strict message ordering.

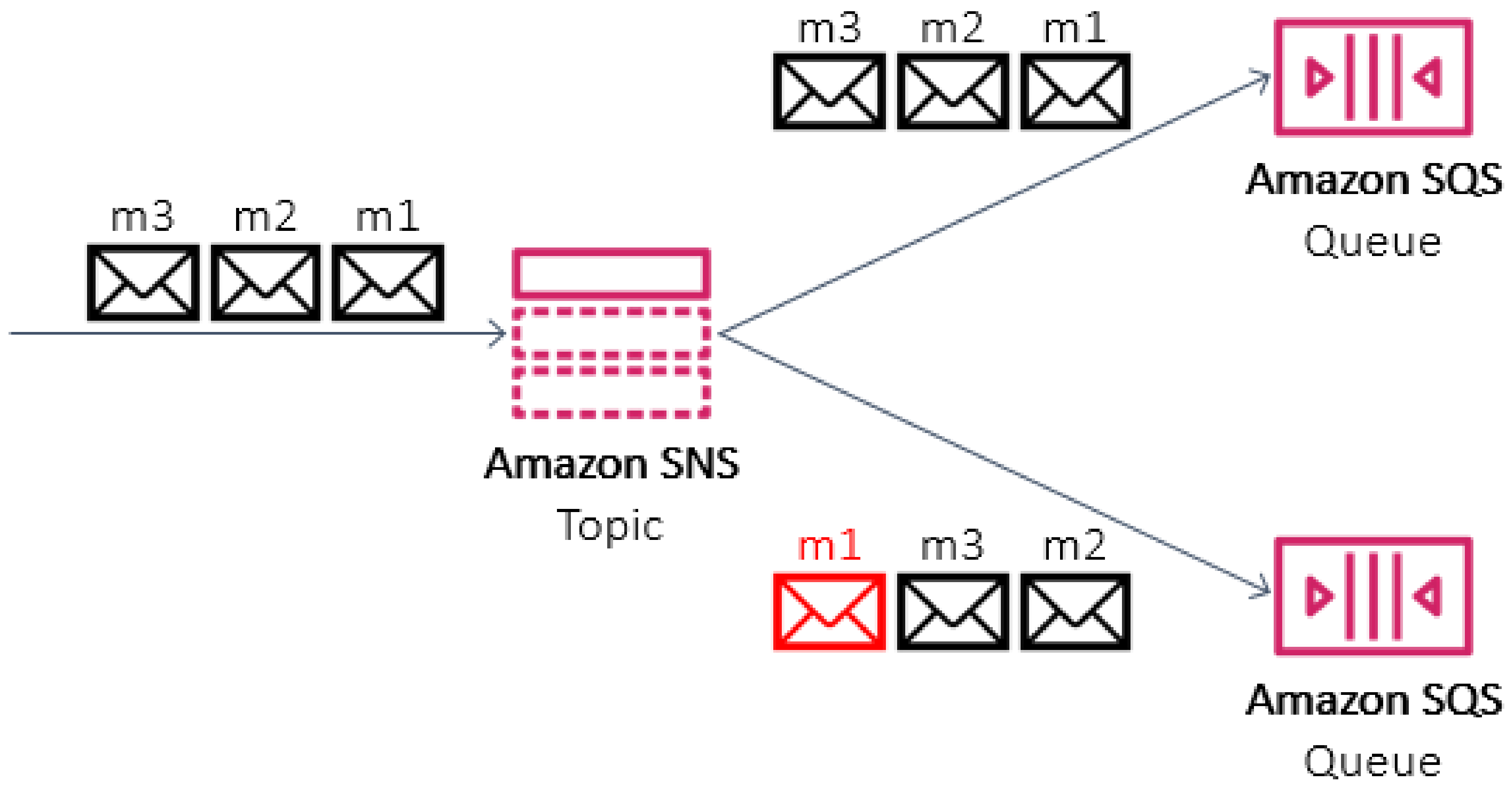
- **High** (Not Unlimited) **Throughput** - If you use **batching**, FIFO queues support up to 3,000 messages per second (300 API calls, 10 messages per call).
- **Exactly-Once processing** - A message is delivered once and remains available until a consumer processes and deletes it. Duplicates aren't introduced into the queue.
- **First-In-First-Out delivery** - The order in which messages are sent and received is strictly preserved.

Message ordering and deduplication concepts also apply to SNS.



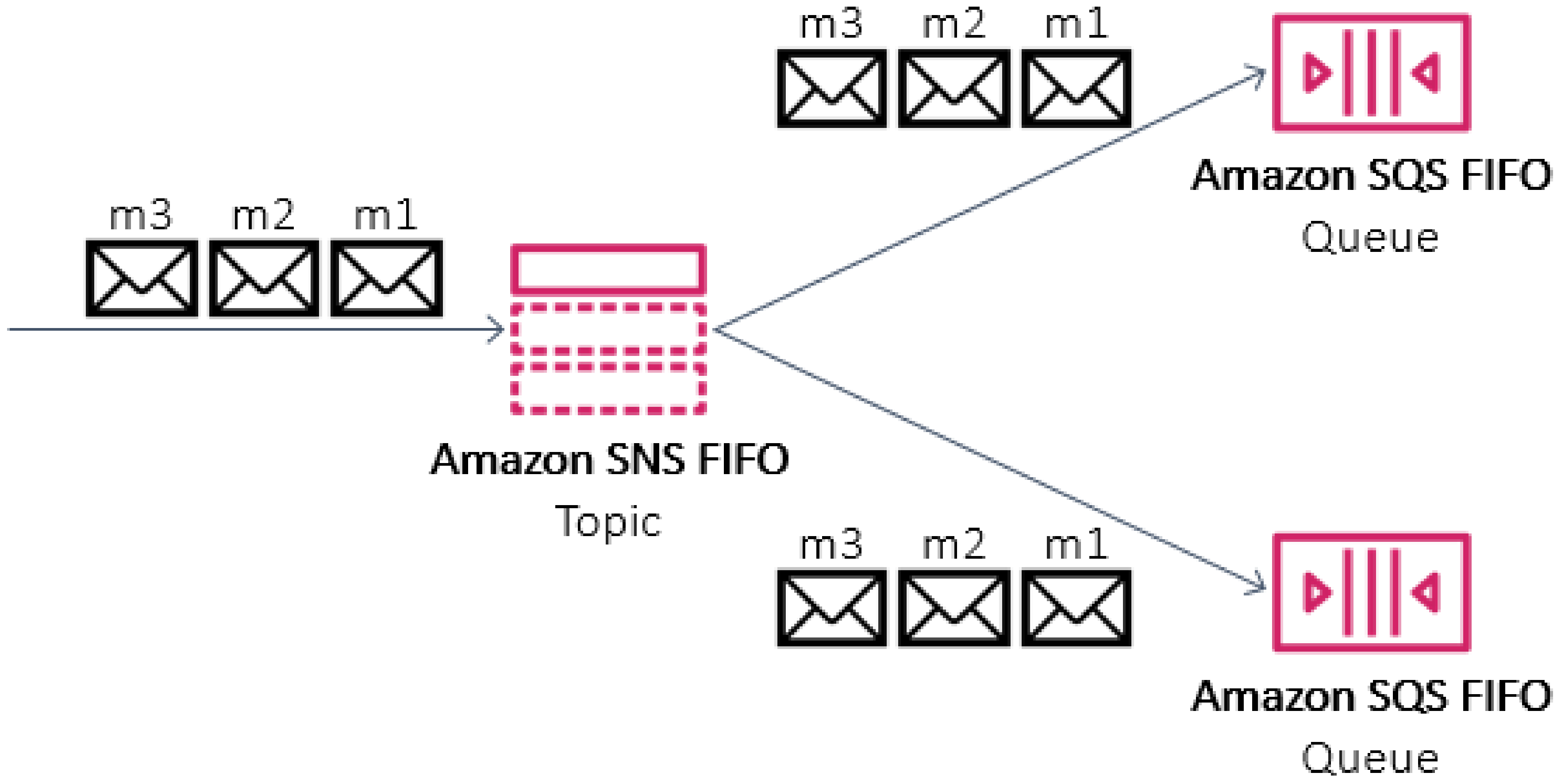
Best-effort message ordering

Standard SNS and SQS provide best-effort message ordering with rarely out of order delivery.



Strict message ordering

SNS FIFO and SQS FIFO provide strict message ordering.



Message deduplication ID

You ordered an item online. But you were charged twice. Because the app processed the message twice. That is when deduplication id (or idempotency key) comes into the picture.

The message deduplication ID is the token used for deduplication of sent messages. If a message with a particular message deduplication ID is sent successfully, any messages sent with the same message deduplication ID are accepted successfully but aren't delivered during the 5-minute **deduplication interval**.

If content-based deduplication for the queue is enabled. The producer can omit the message deduplication ID.

Deduplication id can also be used for:

- Different contents but must be treated as duplicates.
- Identical content but different attributes must be treated as unique.

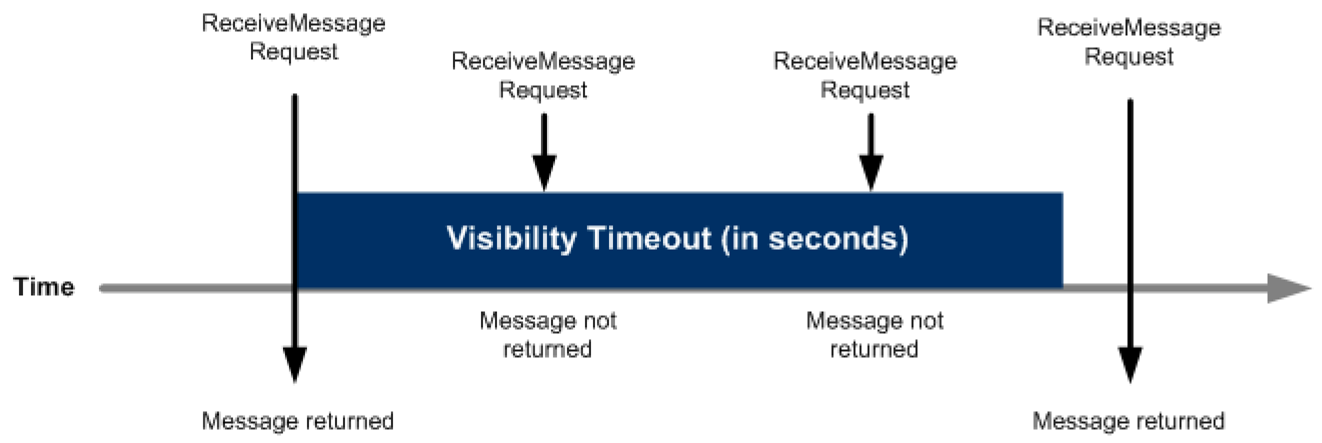
Message group ID

The message group ID is the tag that specifies that a message belongs to a specific message group. Messages that belong to the same message group are always processed one by one.

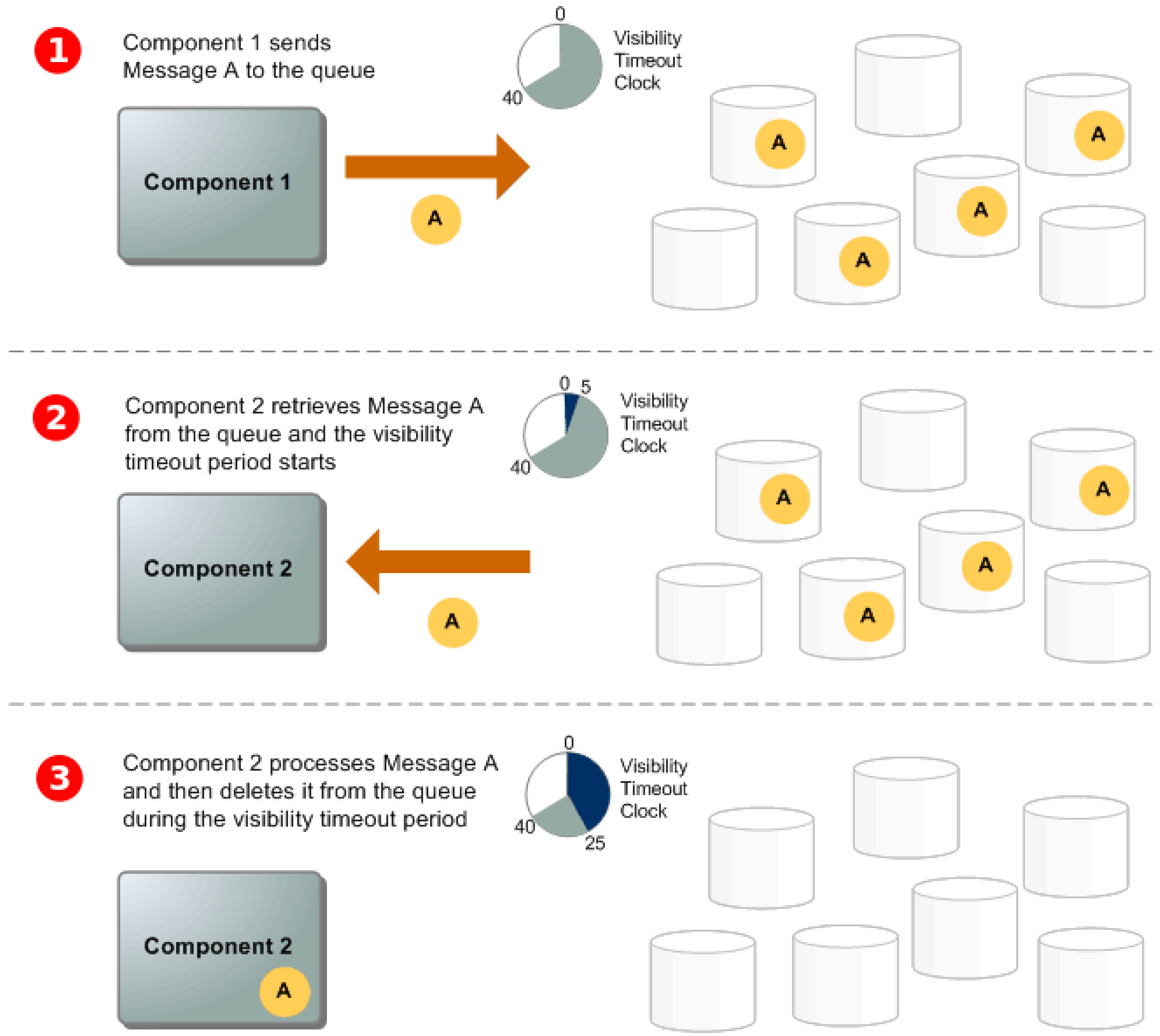
- Max number of messages in the queue is 20,000. Use group ID to avoid it. With groups, you can process messages parallelly.
- There could be bottleneck on the group of messages if the current message is not being processed properly by a consumer. Use group ID to prevent it.

SQS visibility timeout

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



Message lifecycle


1
Component 1 sends Message A to the queue
0
40
Visibility Timeout Clock
Component 1
A
A
A
A
A
2
Component 2 retrieves Message A from the queue and the visibility timeout period starts
0
5
40
Visibility Timeout Clock
Component 2
A
A
A
A
A
3
Component 2 processes Message A and then deletes it from the queue during the visibility timeout period
0
40
25
Visibility Timeout Clock
Component 2
A

SQS short and long polling

Amazon SQS provides short polling and long polling to receive messages from a queue. By default, queues use short polling.

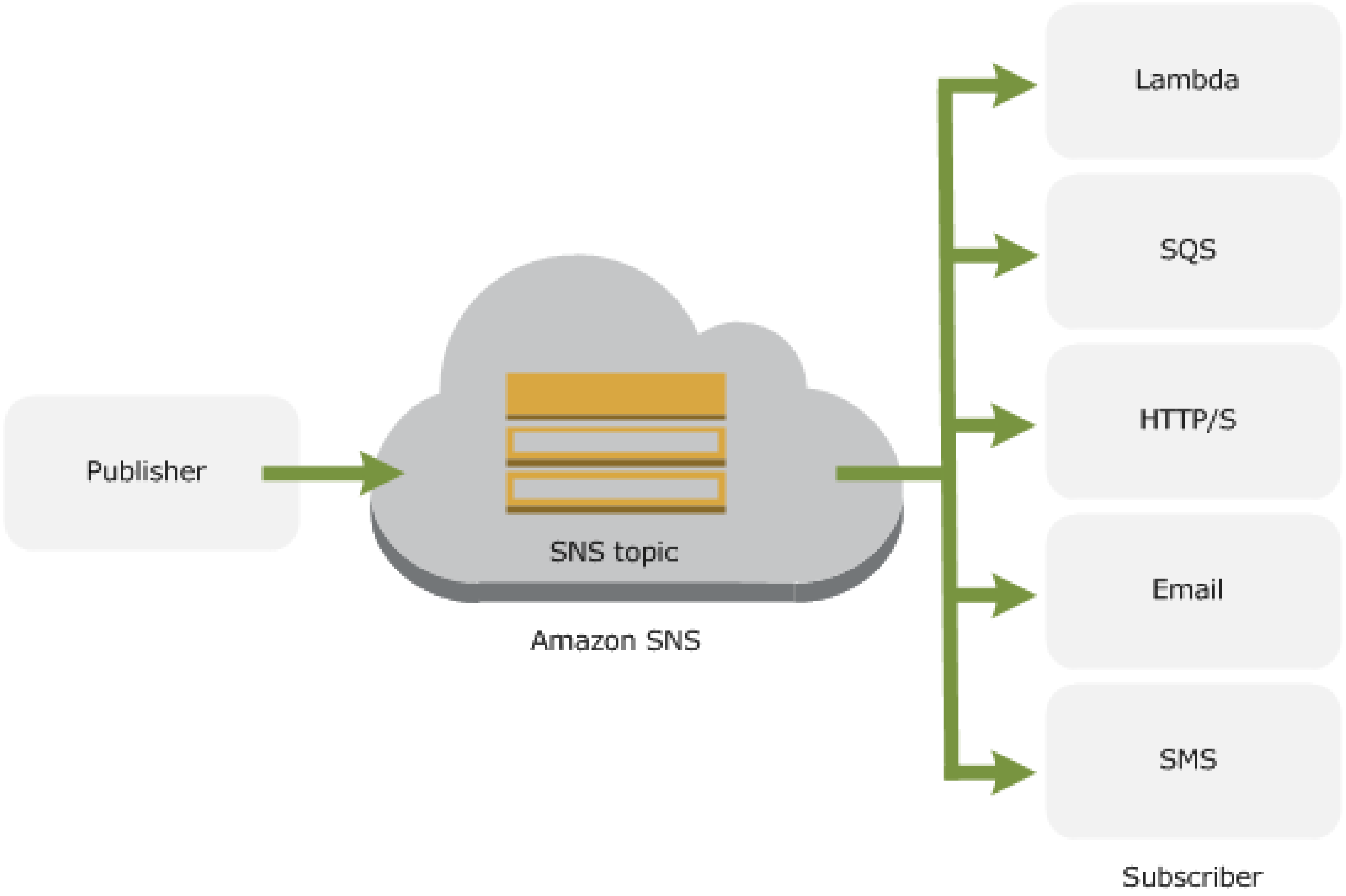
- With **short polling**, the ReceiveMessage request queries only a **subset of the servers** to find messages that are available to include in the response. Amazon SQS sends the response right away, even if the query found **no messages**.
- With **long polling**, the server keeps the client connection open and the ReceiveMessage request queries **all of the servers** for messages. Amazon SQS sends a response after it collects **at least one available message**, up to the maximum number of messages specified in the request. Amazon SQS sends an empty response only if the polling wait time expires.

Simple Notification Service - SNS

Simple Notification Service (SNS) is a fast, flexible, fully-managed **push** notification service that sends messages to subscribers from the message publisher.

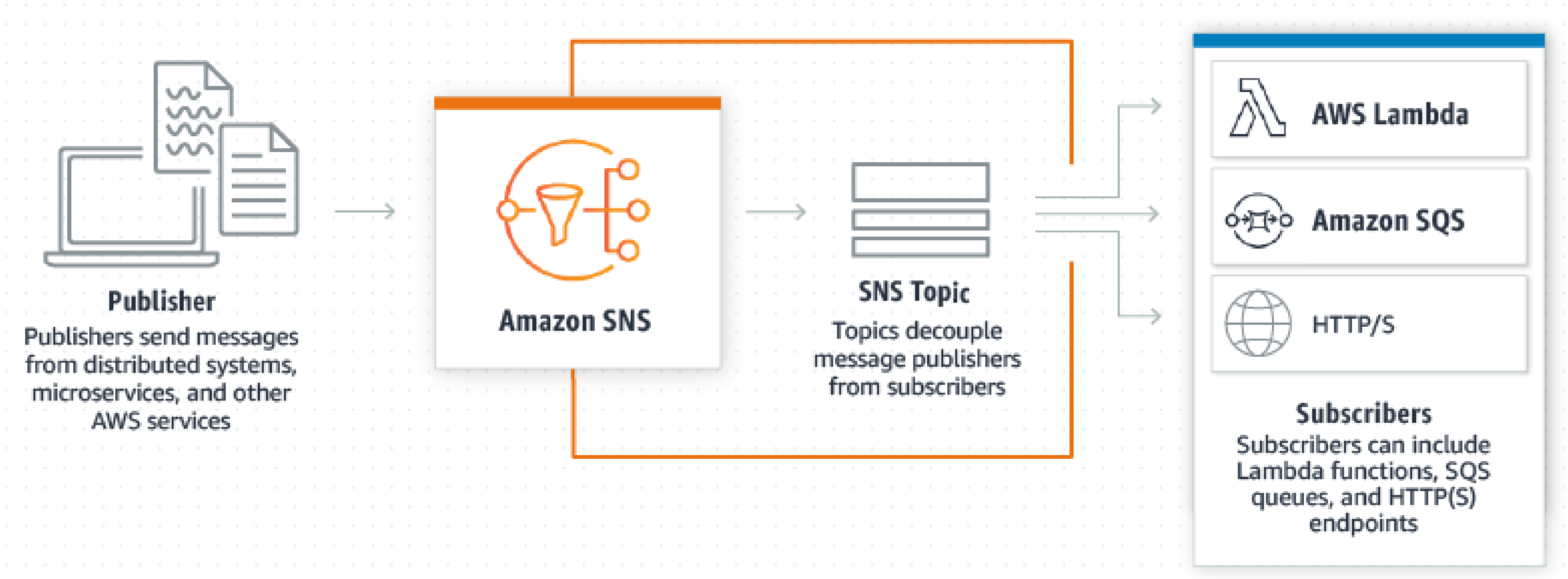
You can use SNS to send emails from your application to the users and send emails to yourself when events occur in your AWS account in conjunction with CloudWatch and so on.

Read more about Amazon SNS


Publisher
SNS topic
Amazon SNS
Lambda
SQS
HTTP/S
Email
SMS
Subscriber

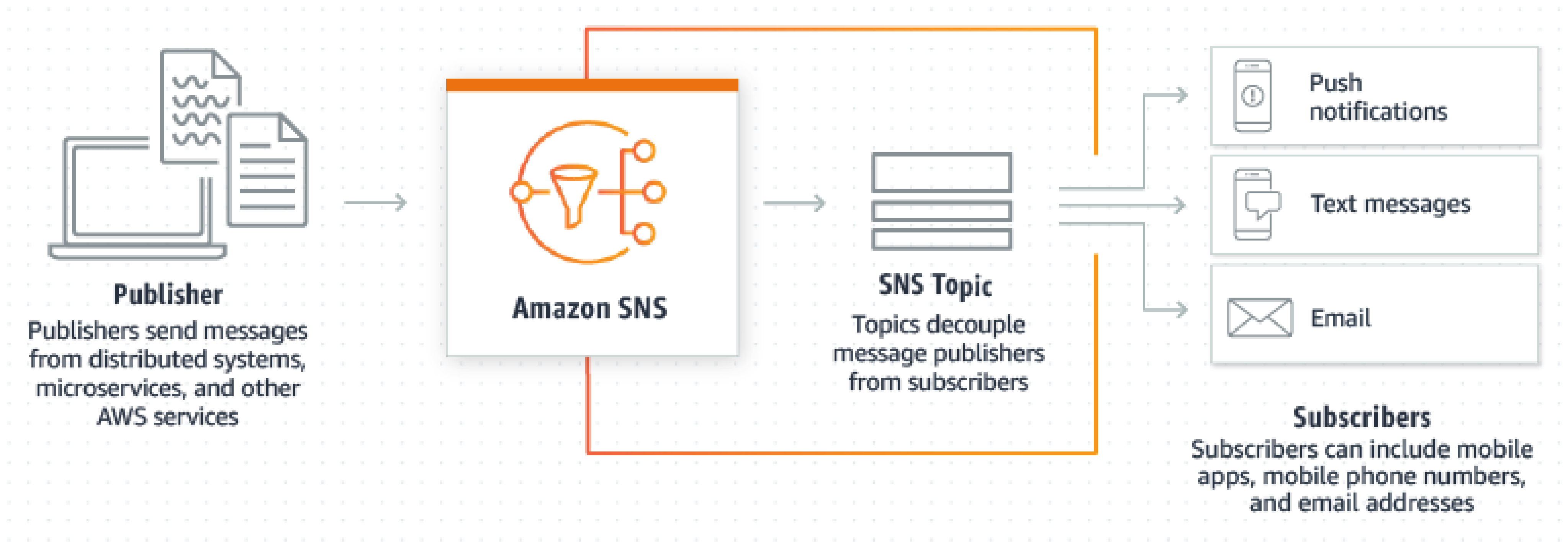
Application-to-application (A2A)

Amazon SNS lets you decouple publishers from subscribers. This is useful for application-to-application messaging for microservices, distributed systems, and serverless applications.


Publisher
Publishers send messages from distributed systems, microservices, and other AWS services
Amazon SNS
SNS Topic
Topics decouple message publishers from subscribers
AWS Lambda
Amazon SQS
HTTP/S
Subscribers
Subscribers can include Lambda functions, SQS queues, and HTTP(S) endpoints

Application-to-person (A2P)

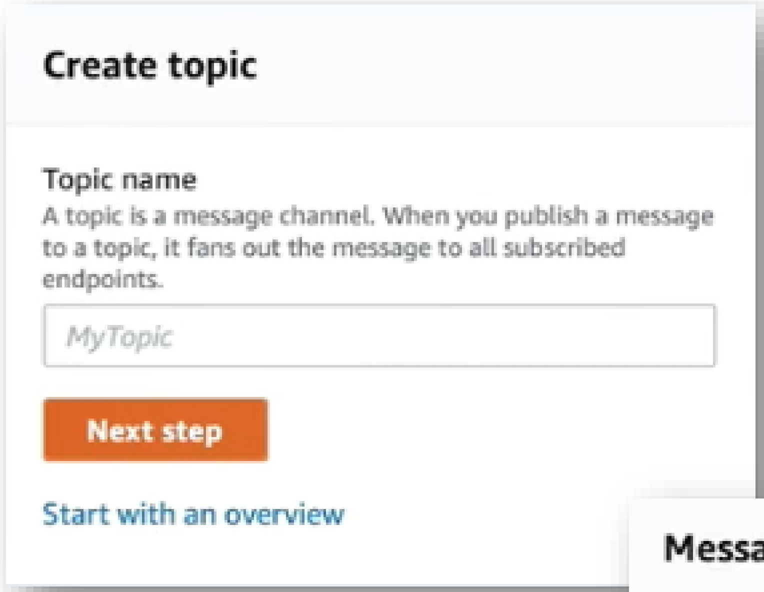
Amazon SNS lets you send push notifications to mobile apps, text messages to mobile phone numbers, and plain-text emails to email addresses. You can fan out messages with a topic, or publish to mobile endpoints directly.


Publisher
Publishers send messages from distributed systems, microservices, and other AWS services
Amazon SNS
SNS Topic
Topics decouple message publishers from subscribers
Push notifications
Text messages
Email
Subscribers
Subscribers can include mobile apps, mobile phone numbers, and email addresses

SNS Basic Components

1. **Publishers** communicate **asynchronously** with subscribers by producing and sending a message to a topic.
2. **Topic** is a logical access point and communication channel.
3. **Subscribers** consume or receive the message or notification over one of the supported protocols when they are subscribed to the topic.

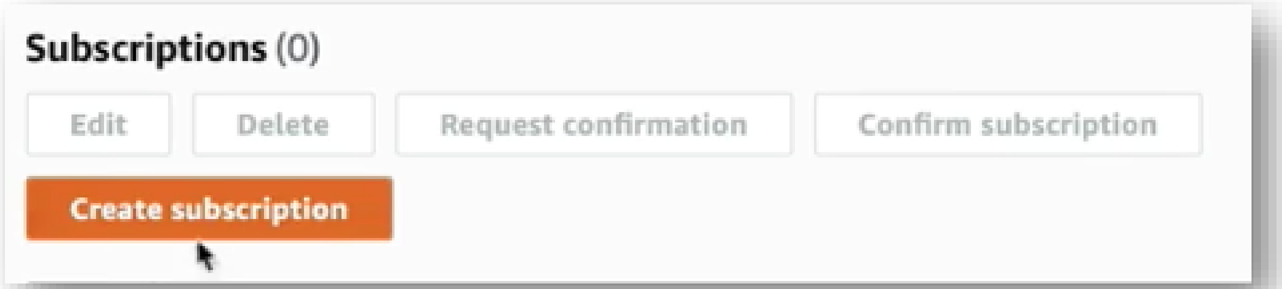
Create a Topic, Subscription, Publish

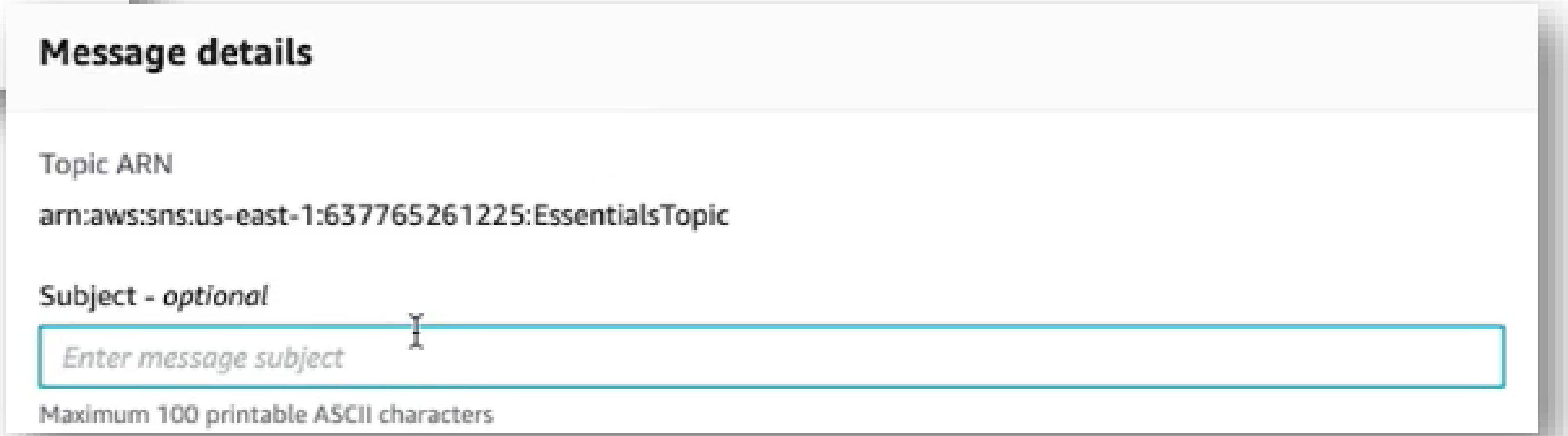


Create topic

Topic name

A topic is a message channel. When you publish a message to a topic, it fans out the message to all subscribed endpoints.





Message details

Topic ARN

arn:aws:sns:us-east-1:637765261225:EssentialsTopic

Subject - *optional*

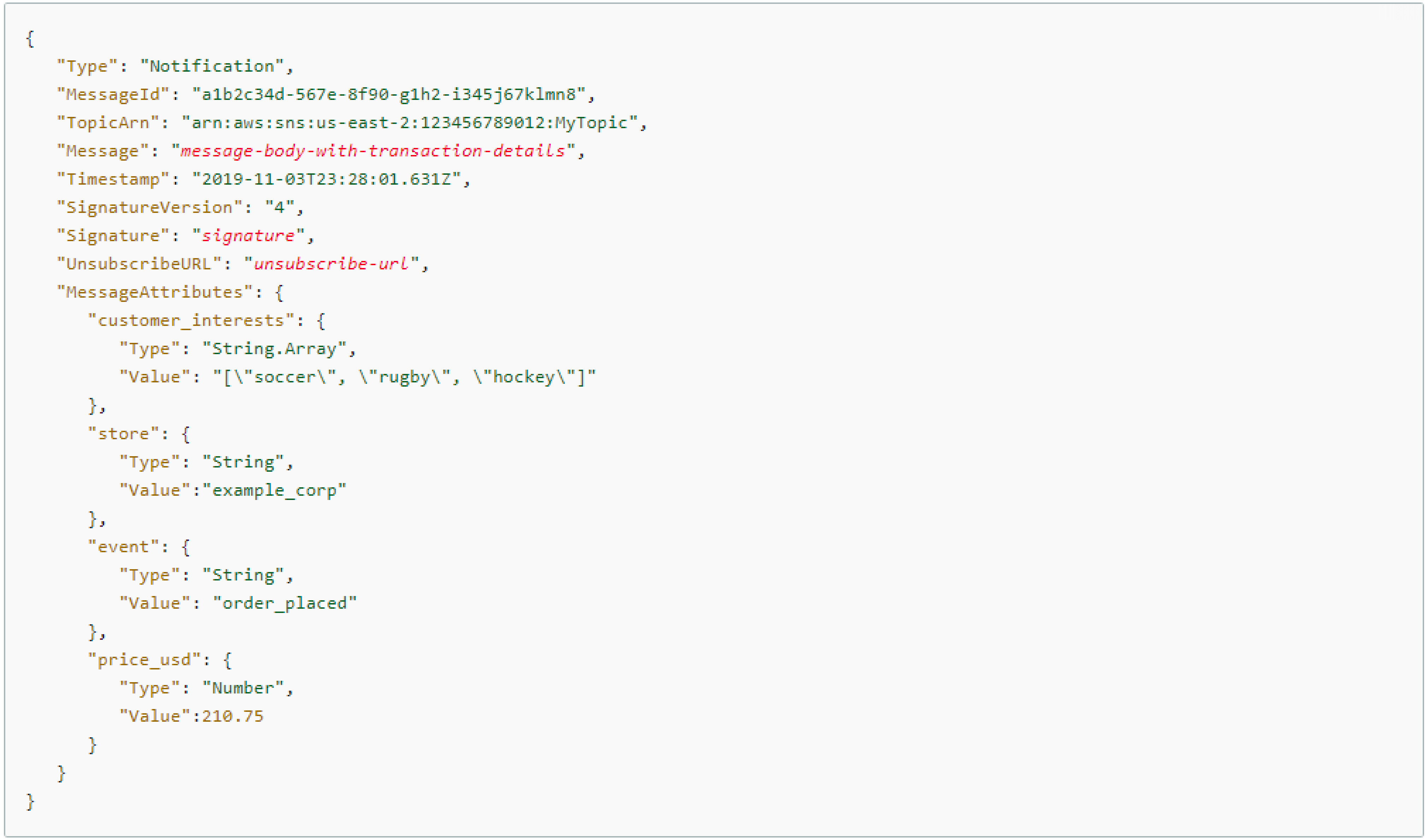
Maximum 100 printable ASCII characters

Message attributes

Message attributes let you provide any **arbitrary metadata** (such as timestamps, geospatial data, signatures, and identifiers) about the message.

You can use message attributes to make your messages filterable using subscription filter policies. You can apply **filter** policies to topic subscriptions. When a filter policy is applied, a subscription receives only those messages that have attributes that the policy accepts.

Sample SNS payload



```
{
   "Type": "Notification",
   "MessageId": "a1b2c34d-567e-8f90-g1h2-i345j67klmn8",
   "TopicArn": "arn:aws:sns:us-east-2:123456789012:MyTopic",
   "Message": "message-body-with-transaction-details",
   "Timestamp": "2019-11-03T23:28:01.631Z",
   "SignatureVersion": "4",
   "Signature": "signature",
   "UnsubscribeURL": "unsubscribe-url",
   "MessageAttributes": {
      "customer_interests": {
         "Type": "String.Array",
         "Value": "[\"soccer\", \"rugby\", \"hockey\"]"
      },
      "store": {
         "Type": "String",
         "Value":"example_corp"
      },
      "event": {
         "Type": "String",
         "Value": "order_placed"
      },
      "price_usd": {
         "Type": "Number",
         "Value":210.75
      }
   }
}
```

Amazon SNS message filtering

By default, an Amazon SNS topic subscriber receives every message published to the topic. To receive a subset of the messages, a subscriber must assign a filter policy to the topic subscription.

You need to pass the **message attributes** (MessageAttributes) parameter which is a map consists of a key and value.

A policy that accepts messages

If any single attribute in this policy doesn't match an attribute assigned to the message, the policy rejects the message.

```
"MessageAttributes": {
   "customer_interests": {
      "Type": "String.Array",
      "Value": "[\"soccer\", \"rugby\", \"hockey\"]"
   },
   "store": {
      "Type": "String",
      "Value":"example_corp"
   },
   "event": {
      "Type": "String",
      "Value": "order_placed"
   },
   "price_usd": {
      "Type": "Number",
      "Value":210.75
   }
}
```

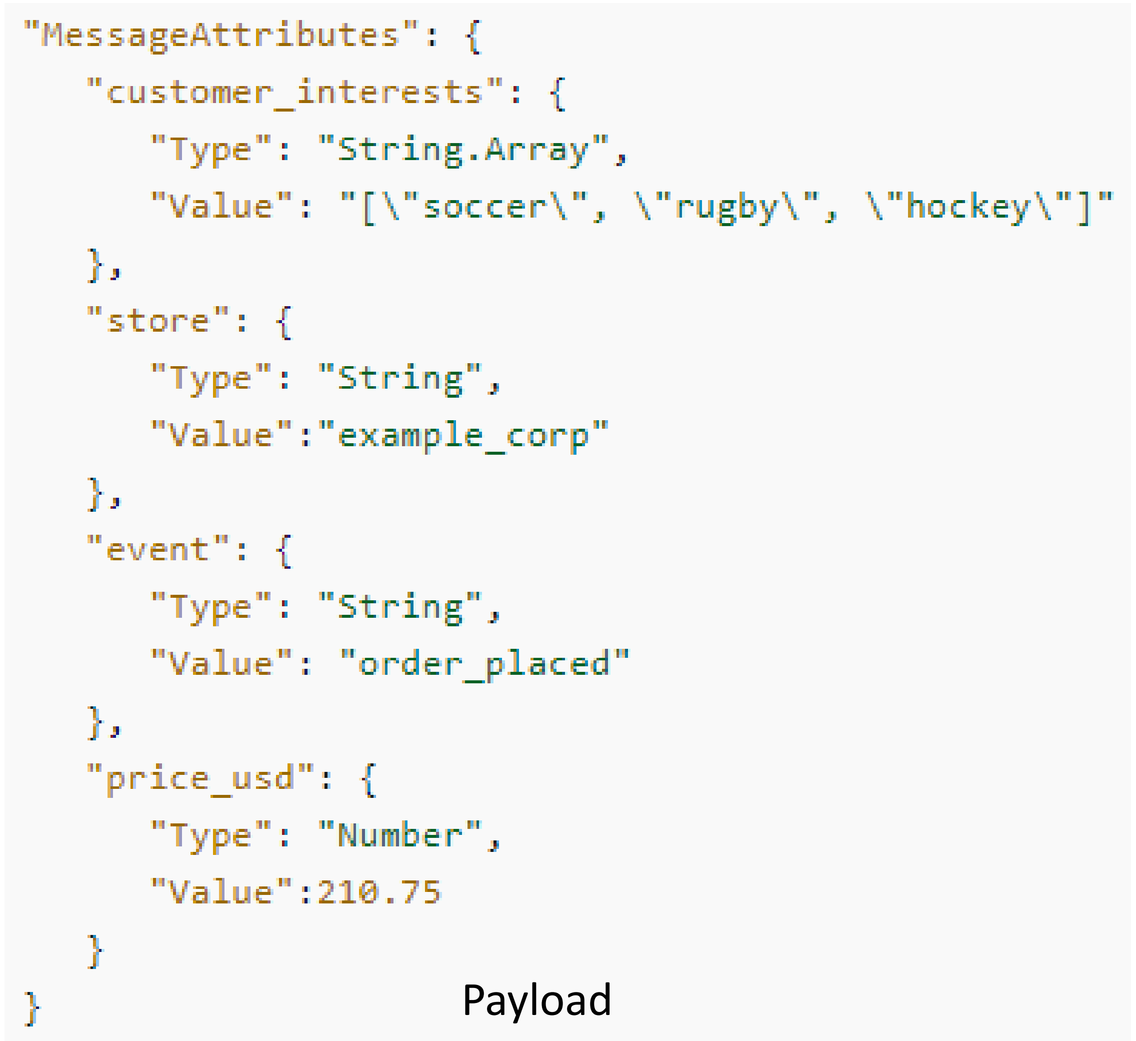
Payload

```
{
   "store": ["example_corp"],
   "event": [{"anything-but": "order_cancelled"}],
   "customer_interests": [
      "rugby",
      "football",
      "baseball"
   ],
   "price_usd": [{"numeric": [">=", 100]}]
}
```

Policy

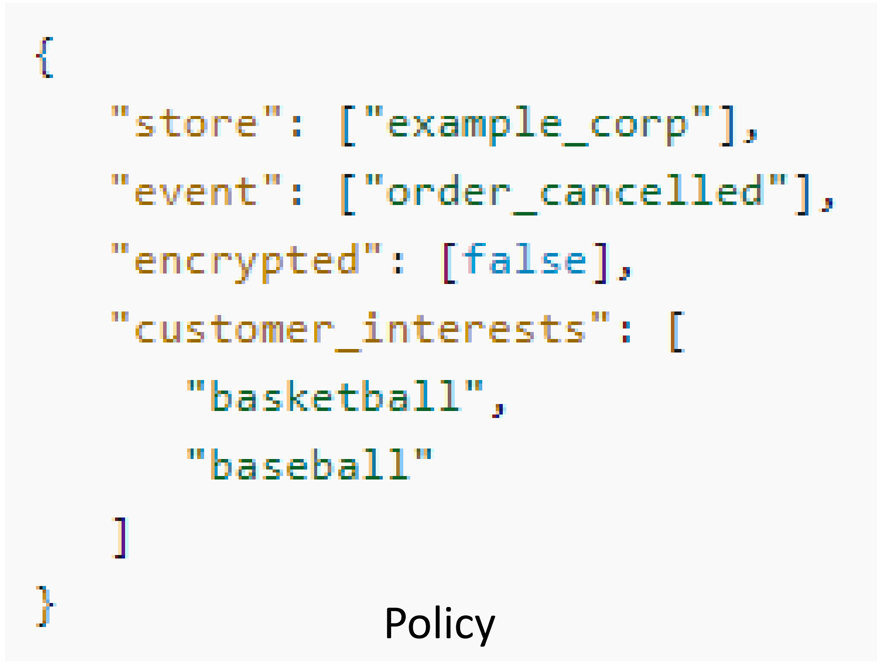
A policy that rejects messages

If any mismatches occur, the policy rejects the message. The encrypted attribute isn't present in the message, this causes the message to be rejected regardless of the value assigned to it.



```
"MessageAttributes": {
   "customer_interests": {
      "Type": "String.Array",
      "Value": "[\"soccer\", \"rugby\", \"hockey\"]"
   },
   "store": {
      "Type": "String",
      "Value":"example_corp"
   },
   "event": {
      "Type": "String",
      "Value": "order_placed"
   },
   "price_usd": {
      "Type": "Number",
      "Value":210.75
   }
}
```

Payload



```
{
   "store": ["example_corp"],
   "event": ["order_cancelled"],
   "encrypted": [false],
   "customer_interests": [
      "basketball",
      "baseball"
   ]
}
```

Policy

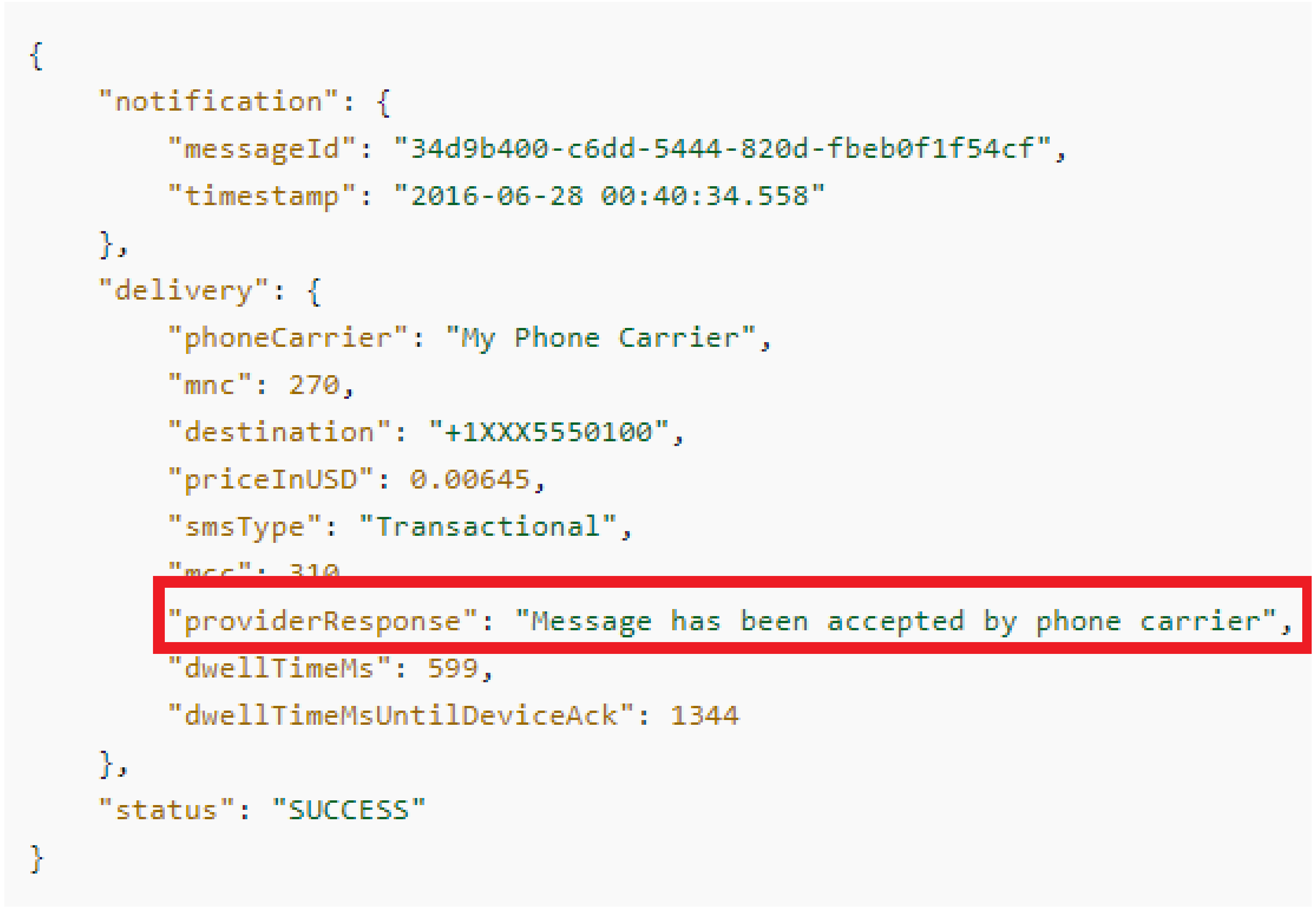
Message delivery status logging

Amazon SNS supports logging the delivery status of notification messages using CloudWatch. Amazon SNS sends the following metrics to CloudWatch:

- NumberOfMessagesPublished
- NumberOfNotificationsDelivered
- NumberOfNotificationsFailed
- NumberOfNotificationsFilteredOut
- PublishSize – Message size etc.

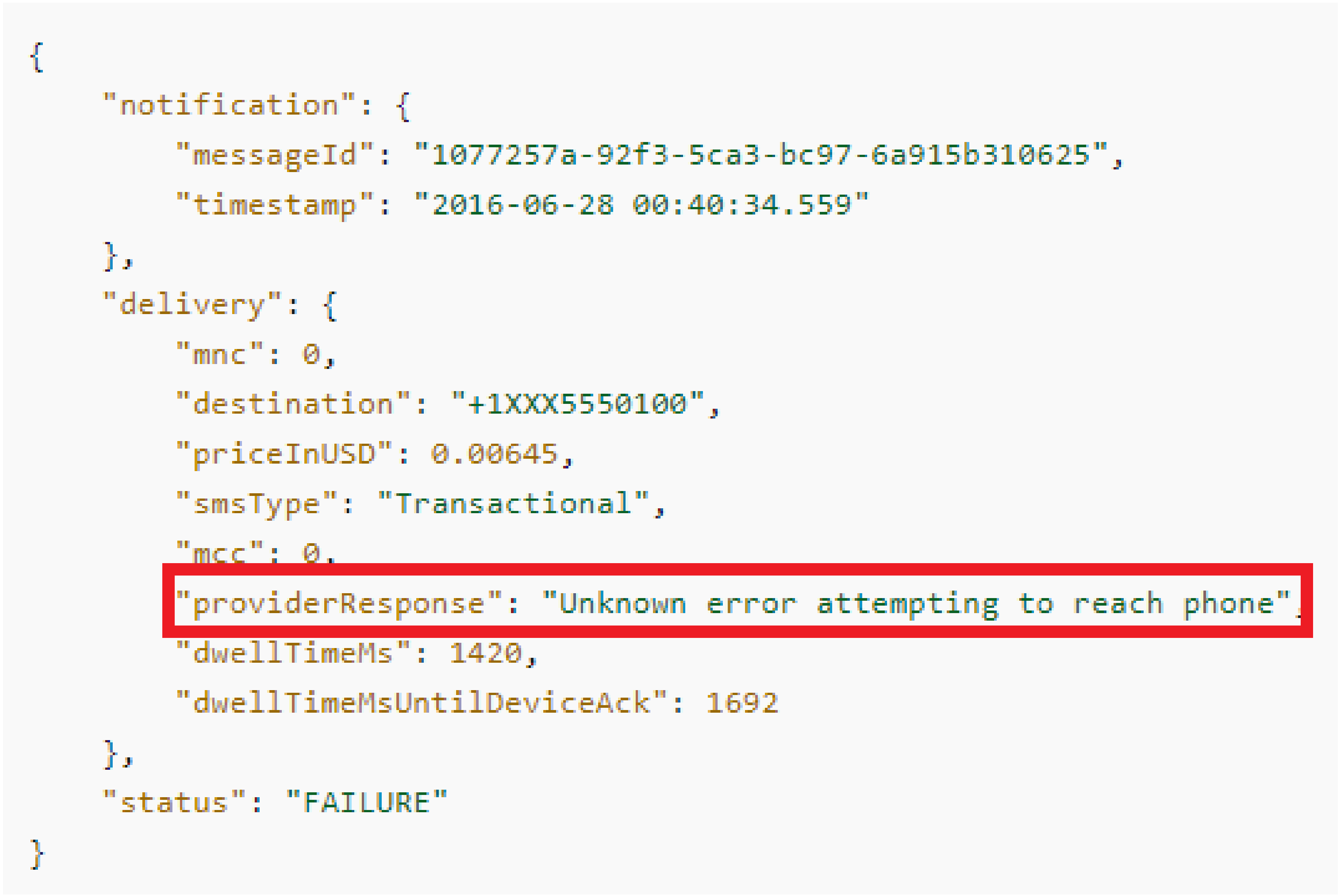
Based on these metrics, you can set a policy for ASG.

Example log for successful SMS delivery



```
{
    "notification": {
        "messageId": "34d9b400-c6dd-5444-820d-fbeb0f1f54cf",
        "timestamp": "2016-06-28 00:40:34.558"
    },
    "delivery": {
        "phoneCarrier": "My Phone Carrier",
        "mnc": 270,
        "destination": "+1XXX5550100",
        "priceInUSD": 0.00645,
        "smsType": "Transactional",
        "mcc": 310
        "providerResponse": "Message has been accepted by phone carrier",
        "dwellTimeMs": 599,
        "dwellTimeMsUntilDeviceAck": 1344
    },
    "status": "SUCCESS"
}
```

Example log for failed SMS delivery



```
{
    "notification": {
        "messageId": "1077257a-92f3-5ca3-bc97-6a915b310625",
        "timestamp": "2016-06-28 00:40:34.559"
    },
    "delivery": {
        "mnc": 0,
        "destination": "+1XXX5550100",
        "priceInUSD": 0.00645,
        "smsType": "Transactional",
        "mcc": 0,
        "providerResponse": "Unknown error attempting to reach phone",
        "dwellTimeMs": 1420,
        "dwellTimeMsUntilDeviceAck": 1692
    },
    "status": "FAILURE"
}
```

Unsuccessful message delivery

SNS retries if message is not delivered. Policies can be:

- 3 times immediately in the no-delay phase
- 2 times (1 second apart) in the pre-backoff phase
- 10 times (with exponential backoff from 1 second to 60 seconds)
- 35 times (60 seconds apart) in the post-backoff phase

If retry is not successful, the message is sent to the **dead-letter queue** further analysis or manual reprocessing.

You can have a dead-letter queue for SQS as well.

EventBridge

- EventBridge is similar to SNS that fans out messages or sends an event (message) to multiple consumers at the same time. It has more features such as schema (even payload validation), third-party API support (DataDog, Shopify, etc), and retry/replay.
- EventBridge pipe is newer feature announced in re:Invent 2022 that allows developers to connect 2 different services without writing any code (or any ETL).
- It has a scheduler that you can run a task using CRON expression such as triggering a lambda everyday at 8 am.

Event-driven architecture

- An **event** is a signal that a system’s state has changed such as a new order is placed. It is just a payload that can have metadata about the event and its type.
- A **rule** (similar to filter) matches incoming events and sends them to targets for processing. A single rule can send an event to multiple targets, which then run in parallel (fan-out).
- Events must be **immutable** (so you can replay)!
- Event-driven architecture is to use serverless services such as Lambda, SQS, SNS, EventBridge, Step Functions in your application. Meaning you already gained experience in event-driven architecture without knowing it in this class.

SNS and SQS pricing

These services are almost free. You pay tiny and achieve a lot.

- SNS (Ohio)
  - Email – The first 1,000 is free, $2 per 100,000. (SES is much cheaper).
  - HTTP – 100,000 is free. $0.60 per million.
  - Mobile – The first 1 million requests are free. Then $0.50 per million.
- SQS
  - First 1M is free.
  - Then $0.40 per million.

Circuit breaker design pattern

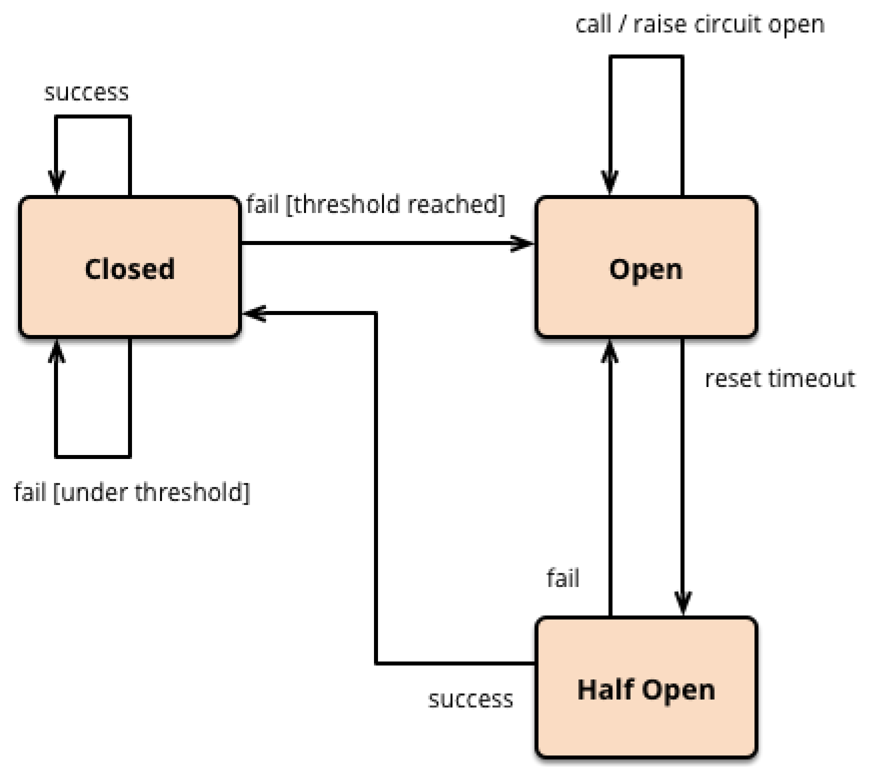
It is used to detect failures and encapsulates the logic of preventing a failure from constantly recurring, during maintenance, temporary external system failure or unexpected system difficulties.

For example, if the database is not available, you can’t keep inserting records into it. Instead, keep it in the queue and send data when the database is back.

Circuit breaker design pattern

Circuit states:

- **Closed** – Underlying system is working. All transactions are success.
- **Open** – The underlying system is **not** working. All transactions failed.
- **Half open** – Sending some requests to see whether the underlying system is available or not.



Popular circuit breaking library: resilience4j