

# **Component-based Software Development**

**Serverless Computing**  
**AWS Lambda,**

**Dr. Vinod Dubey**  
**SWE 645**  
**George Mason University**

# Acknowledgement

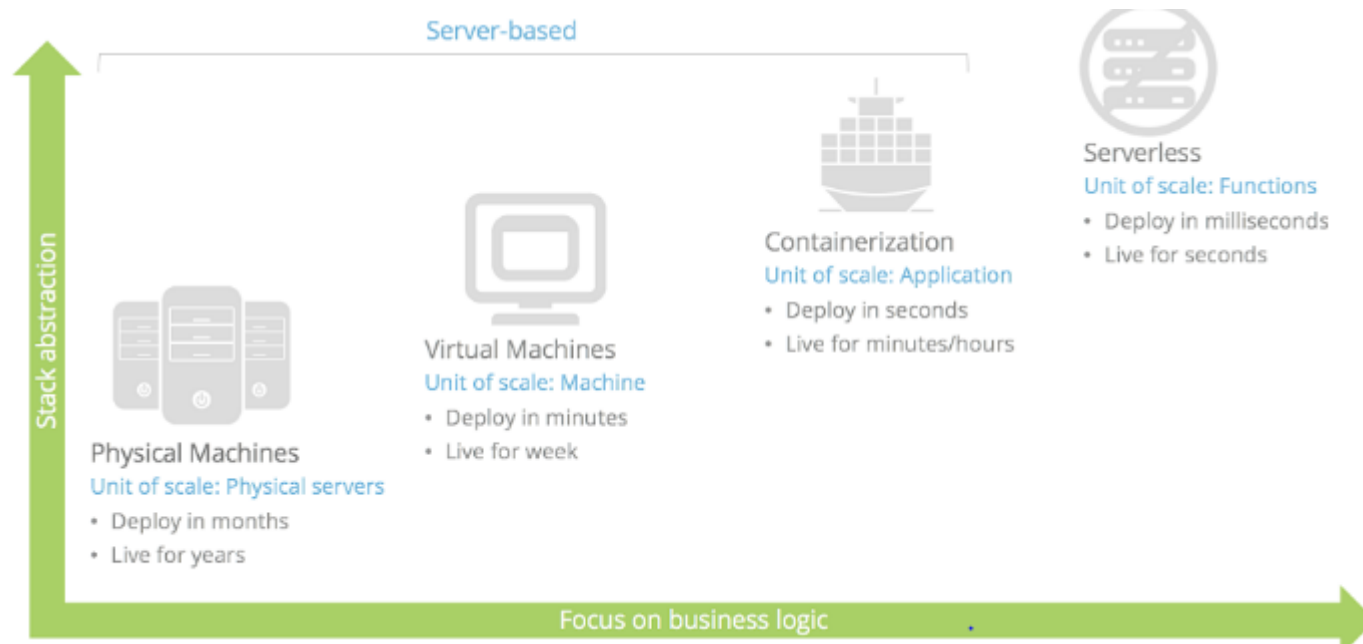
Information on some of the slides in this lecture is adapted from  
<http://aws.amazon.com/lambda>,

# Agenda

- **Evolution of Computing**
- **Serverless Computing**
  - Key characteristics
- **AWS Lambda**
  - Execution model
  - Programming model
  - When to use Lambda
  - Common Use Cases
- **Optional**
  - Amazon SQS
  - Amazon SNS

# Evolution of Computing

- **Serverless computing provides the next level of abstraction in cloud computing**
  - Has a huge potential to transform the way businesses consume cloud services



# Evolution of Computing – a paradigm shift

- **Physical Machines**

- Over the course of last decade, we have seen the evolution that began with the deployment of physical machines in on-premises data center.
- Required guess or very rigorous capacity planning
- Physical machine deployment would take months
- The capital investment was rather expensive
- Physical machines persisted on premises for relatively long time.

# Evolution of Computing – a paradigm shift

- **Virtual Machines**

- Then the introduction of **virtual machines** gave us **hardware independence**
- Cloud computing with virtual machines enabled **faster provisioning (minutes/hours)** as opposed to months
- More **scalable** and **elastic** resources
- This resulted in **speed** and agility and **reduced maintenance**

# Evolution of Computing – a paradigm shift

- **Containerization**

- Eventually Docker gave rise to the popularity of containers
- Containers provided consistent runtime environment
- Easier and faster deployments
- Isolation and sandboxing
- Organizations were able to achieve platform independence
- By being able to bin pack a lot of containers onto the instances allowed them to achieve greater resource utilization
- Containers allowed users to start/deploy their applications faster, literally within seconds

# Evolution of Computing – a paradigm shift

- **Serverless**

- Now we have environments where cloud operators like AWS, Azure, Google offer fully managed compute services
- Services such as AWS Lambda, supports
  - Continuous scaling
  - Built-in fault-tolerance and availability
  - Pay only when code is running
  - No maintenance required on behalf of customers



# Serverless Computing

- “A cloud-computing **execution model** in which the **cloud provider provisions, manages, and runs the server**, and dynamically manages the allocation of machine resources.
- **Pricing is based on the actual amount of resources consumed** by an application,
  - rather than on pre-purchased units of capacity.
- **It’s a true pay-as-you-go model.**
- **It can be a form of utility computing”**

(Source: wiki)

[https://en.wikipedia.org/wiki/Serverless\\_computing](https://en.wikipedia.org/wiki/Serverless_computing)

# Serverless Computing

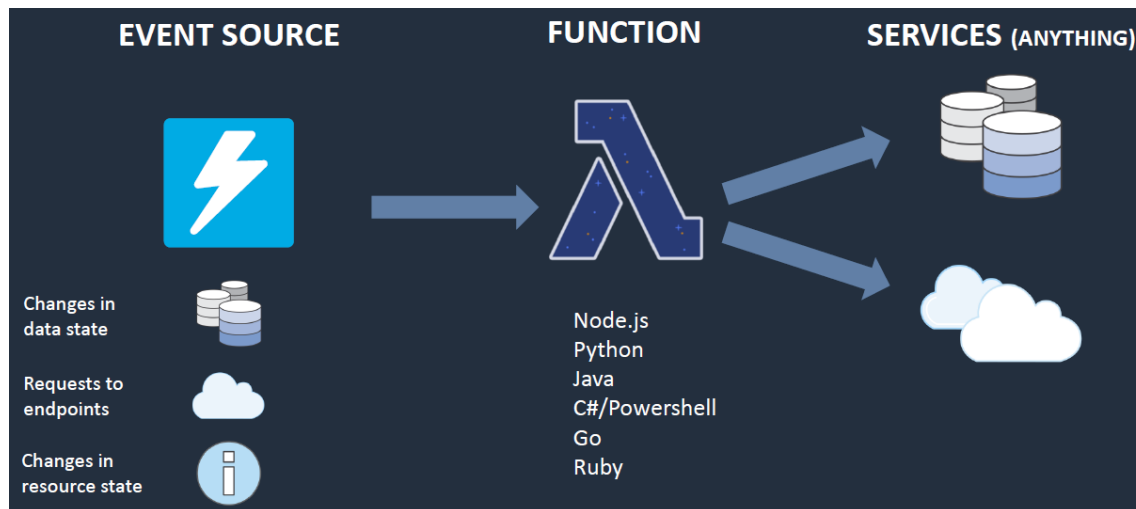
- **Four key characteristics**
  - No server to provision or manage
  - Scales with usage
  - Never pay for idle – only pay for what you use, that is when your code is running
  - Availability and fault tolerance built in
- **Example of serverless services in AWS**
  - AWS Lambda, Amazon API Gateway, Step Functions
  - Amazon S3, DynamoDB, SQS, SNS, AWS Fargate

# Serverless Computing

- **The term “serverless” is a misnomer!**
  - The cloud providers still use servers, but customers have no role in provisioning and maintaining them.
- **Helps customers focus on their core businesses**
  - Leave the worries of maintaining and patching servers to the cloud provider.

# AWS Lambda

- AWS Lambda is an **event-driven, serverless computing platform** on AWS cloud.
- It is a **compute service that runs code in response to events** and automatically manages the computing resources required by that code.
- **Run code without thinking about servers or clusters**
  - Amazon's serverless functions service



# AWS Lambda

- **No infrastructure provisioning**
  - No server to provision or manage
- **Fully managed automated scaling**
  - Scales with usage
  - Supports elasticity
- **High availability, fault-tolerant, & security**
  - Built-in
- **Never pay for idle**
  - Only paying when your function is running

# AWS Lambda

- **Based on event driven architecture**
  - Event-driven implies that a Lambda function is triggered at the occurrence of an event, and the function has the responsibility to process it.
  - Every AWS lambda function is an event consumer in this kind of architecture.
- **Allows you to use many language runtime to run your code/functions**
  - Python, Java, Node.js, and more
- **Natively integrated with different data sources**
  - S3, DynamoDB, RedShift
- **Fully managed infrastructure**
  - AWS provisions and maintains resources

# What can be done with Lambda

- **IT Automation**

- You can configure Lambda to turn off the dev environment at 7:00 PM each day and turning it on again at 7:00 AM.
  - For example, schedule task – where Lambda can suspend instance tagged with DEV env
  - Potentially replacing most of cron jobs by Lambda functions

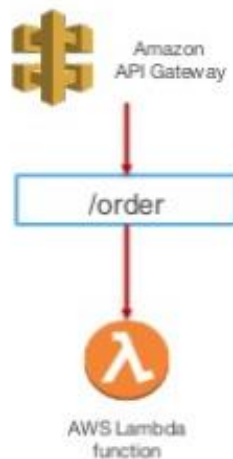
- **Data Processing**

- Lambda can process messages that come through
  - pub-sub architecture such as SNS , or
  - queue based processing with (SQS), or
  - string based processing Kinesis data streams.
- All of these data sources natively integrate with Lambda, where Lambda function doesn't worry about how to read event source – it just focuses on how to handle processing.

# AWS Lambda Execution model

- **Synchronous (push), Asynchronous (event), Stream-based**
  - Lambda service includes a polling capability to check (e.g., a queue) for messages
  - Lambda invokes your code in your function only when there is any message in the queue
  - You don't have to build polling capability in your code

Synchronous (push)



Asynchronous (event)



Stream-based



Source: [amazon.com/lambda](https://amazon.com/lambda)



# Data Source Integration

- **Trigger mechanisms/event sources that integrates with Lambda**
  - Lambda is natively integrated with various data sources including streams, queues, API Gateway, Kinesis, S3, DynamoDB, SNS, SQS, IoT, and others, including some select third party partners.



Source: [amazon.com/lambda](https://aws.amazon.com/lambda)

# Comparison of Shared Operational Responsibility

- **AWS Manages**

- Data source integration

- Data source integration is native to Lambda including streams, queues, backend APIs, and other integrations across the ecosystem
    - For example, if data source is a queue, Lambda service includes a polling capability to check that queue for messages, and when and only when there is any message in the queue, the Lambda service will invoke your code in your function.

- You don't have to build polling capability in your code

- Cluster level scaling and orchestration

- Application-level runtime and updates

- Physical hardware, software, networking, and facilities

- Provisioning

- **Customer Manages**

- Application code

- Security firewall and network configurations

- IAM roles and policies

# Programming Model:

## Key pieces of a Lambda Function

- **Handler() function** – Handler is the function AWS Lambda service calls to start execution of the Lambda function.
  - It takes two objects: **Event** (1st parameter) and **Context** (2<sup>nd</sup> parameter)
  - This encapsulate the **business logic** of the code
- **Event object** – represents the **data** being sent into the handler from the event source or data coming in from API invocation
  - The data sent during Lambda function invocation
  - JSON object
  - The handler processes the incoming event data and may invoke any other functions/methods in the code.
- **Context object**, which has methods available to interact with runtime information (e.g., log group, request IDs etc.)
  - For example, using the context object, your code can find the **execution time remaining** before AWS Lambda terminates your Lambda function.

```
import json

def lambda_handler(event, context):
    # TODO implement
    return {
        'statusCode': 200,
        'body': json.dumps('Hello world!')
    }
```

# Programming Model

- **Logging** – Your Lambda function can contain logging statements.
  - Lambda writes these logs to [CloudWatch Logs](#).
- **Exceptions** – If there is an issue/error processing an event, you can return an exception
  - For synchronous invocations, AWS Lambda forwards the result back to the client.
  - For asynchronous invocation, Lambda retries on errors automatically.

# Programming Model

- **Concurrency** – When your function is invoked more quickly than a single instance of your function can process events, Lambda scales up by running additional instances.
  - There is a default concurrency quota on your account per region.
    - Need to request a limit increase if you do go above that limit.
  - Each instance of the function handles only one request at a time
    - No need to worry about synchronizing threads or processes.
  - Can use asynchronous language features to process batches of events in parallel
- **Scaling is managed automatically**, from 0 to N invocations per second. It's based on usage.
  - Under the hood, it makes use of containers to allocate resources quickly and efficiently.

# Programming Model

- **Stateless Execution** - Your Lambda function **code must be written in a stateless style**, and have no affinity with the underlying compute infrastructure.
  - Persistent state should be stored in Amazon S3, Amazon DynamoDB, or another cloud storage service.
  - Requiring functions to be stateless **enables AWS Lambda to scale**
    - Enables Lambda to launch as many copies of a function as needed to scale to the incoming rate of events and requests.
  - These functions may not always run on the same compute instance from request to request
  - A given instance of a Lambda function may be used more than once by AWS Lambda
- **Layers** - You can configure a Lambda function **to pull in additional code** and content in the form of layers.
  - A layer is a **ZIP archive that contains libraries**, a custom runtime, or other dependencies.

# Security and Access Control

- **Focuses essentially on two aspects**
  - **Who can invoke the Lambda function**
    - based on function invocation policy
  - **What can the function do**
    - Based on function execution role – an IAM role with a trust policy to access other AWS services
- Lambda functions may be **attached to a specific VPC** for additional security
- Lambda code is **stored in a Lambda managed S3** bucket and is encrypted at rest



# Performance

- **Based on simplified computing environment in terms of configuration.**
  - Only one knob to dial related to the memory allocation of the function.
  - RAM range: AWS Lambda now supports up to 10 GB of memory and 6 vCPU cores for Lambda Functions.
  - Amazon allocates CPU and network resources proportionately to RAM.
- **Longest run time that your code can execute is 15 minutes.**
  - You can now configure your AWS Lambda functions to run up to 15 minutes per execution.
  - If things could go beyond 15 minutes, consider
    - Maybe a distributed workload with AWS Step Function
    - Consider asynchronous patterns with callback when appropriate



# Cost Optimization

- **Pricing is based on two dimensions, and it is all on-demand price.**
  - Number of invocation requests
  - Duration of compute time used
- **Lambda Power Tuning can be used to finetune and right size Lambda function, and to determine the optimal memory settings**
- **CloudWatch logs show the memory used, which plays a role determining the compute time**

# Lambda Pricing

- **Invocations**

- First 1M invocations are free
- \$0.20/1M invocations thereafter

- **Compute Duration**

- First 400 GB-seconds are free
- \$0.0000166667/GB-sec thereafter (\$0.20/12,000 GB-seconds)
- Priced in 100 millisecond increments of usage
- CPU Memory Provisioned (MB) \* Execution Time (milliseconds) => GB-Seconds

# Monitoring

- **Logging** – Lambda is **natively integrated** with **CloudWatch**
  - Anything your function writes to stdout/stderr ends up in CloudWatch logs
- **Metrics** – CloudWatch metrics (e.g., error or throttles) **can be configured for alarms** or kickoffs of other events.
- **Tracing** – Ability to integrate with services such as **AWS X-Ray**
  - for tracing a given interaction/data flowing end-to-end throughout an architecture

# Testing and CI/CD Integration

- **AWS CodeCommit/CodeDeploy/CodePipeline can be used to enable CI/CD pipeline for Lambda functions**
  - **automatic rollback, gradual rollout**
- **Integration with other tools such as Serverless Application Model (SAM).**

# When to use Lambda - Decision factors

- **Level of management**
  - Whether you want to be involved managing functionality at the application level (or at the platform-level runtime).
  - Ability to focus on business logic and to take advantage of serverless capability
- **Short Running tasks**
  - Lambda function should run **15 minutes or less**
  - If it's part of workflow (e.g., using AWS Step Function), does each component of the workflow should run 15 mins or less?
- **Memory Limit**
  - Lambda runs on commodity hardware, with a max of **10 GB RAM**
- **Stateless Environment**
  - Be conscious that Lambda is a stateless environment
  - An ephemeral temp storage is available to read/write to throughout the function's lifecycle.
    - Anything that you write to the temp store will go away.
  - **\*\***You can also read/write function's state to persistent storage, such as S3 or DynamoDB
- **Scaling and burst limit**
  - Be conscious of Lambda scaling behavior and Lambda burst limit.
  - There is a burst limit per region which documents how far out you can go concurrency wise in a given region.
  - If you need to exceed that, you may want to pre-warm

# Common Use Cases

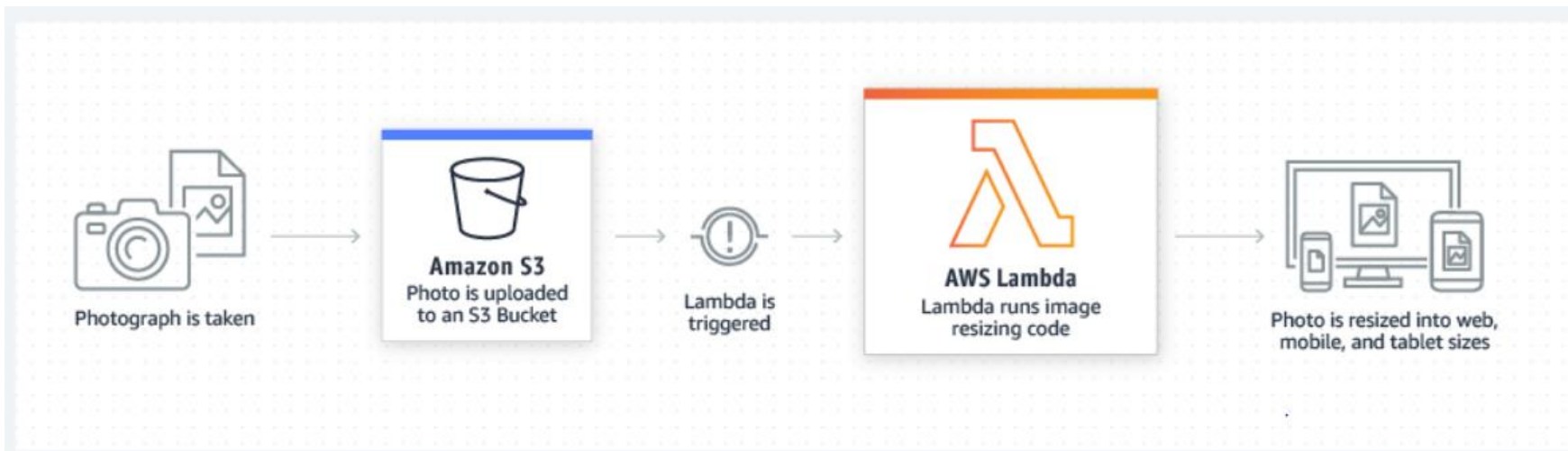
- **Web Applications**

- By combining AWS Lambda with other AWS services, you can build powerful web applications that automatically scale up and down and run in a highly available configuration across multiple data centers – with zero administrative effort required for scalability, backups or multi-data center redundancy.
- Reference Sample Code:
  - <https://github.com/aws-samples/lambda-refarch-webapp>



# Common Use Cases

- **Near Real-Time File Processing**
  - You can use Amazon S3 to trigger AWS Lambda to process data immediately after an upload.
  - For example, you can use Lambda to thumbnail images, transcode videos, index files, process logs, validate content, and aggregate and filter data in real-time.
  - Reference Sample Code:
    - <https://github.com/aws-samples/lambda-refarch-fileprocessing>





# Common Use Cases

- **Near Real-Time Stream Processing**

- You can use **AWS Lambda** and **Amazon Kinesis** to process **real-time streaming data** for application activity tracking, transaction order processing, click stream analysis, data cleansing, metrics generation, log filtering, indexing, social media analysis, and IoT device data telemetry and metering.
- Reference Sample Code:
  - <https://github.com/aws-samples/lambda-refarch-streamprocessing>





# Common Use Cases

- **Extract-Transform-Load**

- AWS Lambda can be used to perform data validation, filtering, sorting, or other transformations for every data change in a DynamoDB table and load the transformed data to another datastore.



# Backup

# **Messaging Support on AWS**

**Amazon SQS & Amazon SNS**

# Queues are everywhere!



Source: AWS reinvent

# Queues are everywhere!



Source: AWS reinvent

**This is one way to solve the problem of serving more people (eventually) than the actual capacity!**

# Message Queue

- A Message Queue is a repository that is used to store messages while waiting to be processed by applications
  - Messages are generated by one component of the application and consumed by another
  - Acts as a message buffer (temporary repository) between message producer(s) and consumer(s)
  - Allows producers to produce and store messages faster than the consumers can process
- **Decouple data producer and consumer**
  - So that components run independently



# Message-Oriented Middleware

- **Message-Oriented Middleware (MOM)** allows **sending** and **receiving** messages between distributed **systems**
- **Java Message Service (JMS)** is a **MOM** that provides a way **for Java programs** to **create, send, receive, and read** messages **asynchronously**



# Messaging Service

- Provides **communication** between applications through “**messages**”
  - one way communication
- **Message clients**
  - Any **application** or component that **uses** a **messaging service**
- **Decouple message clients**
  - **Senders** and **receivers** do **not** need to **know** each other

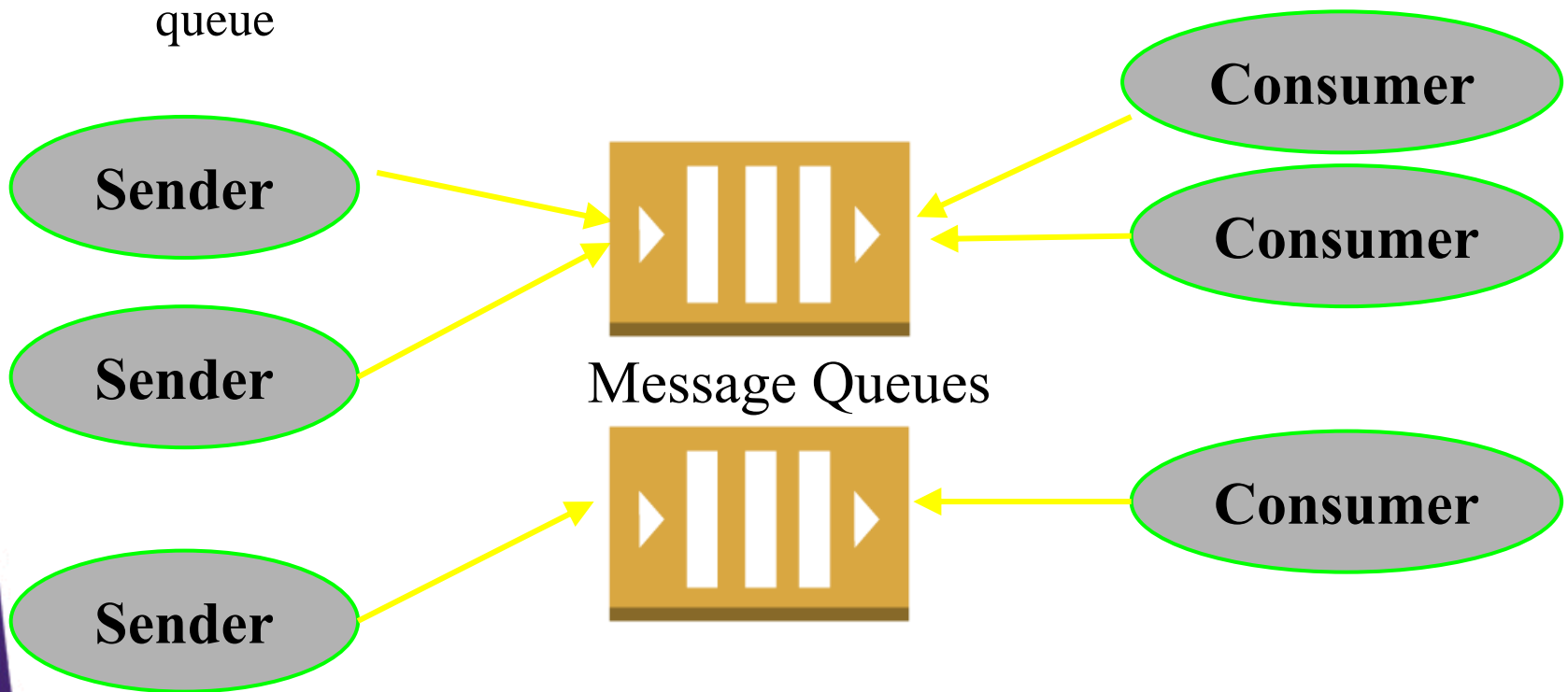


# Point-to-Point Messaging Model

- A publisher sends a message to a specific destination, called a queue
- Multiple publishers can send messages to the queue, but each message is delivered to and consumed by one consumer only.
- Queues retain all messages sent to them until the messages are consumed or expire.

# Point-to-Point Messaging Model

- In Point to Point model, there is one, and **only one**, **message consumer for each message**
  - although there maybe more (than one) consumers listening to the same queue



# Publish-Subscribe Messaging Model

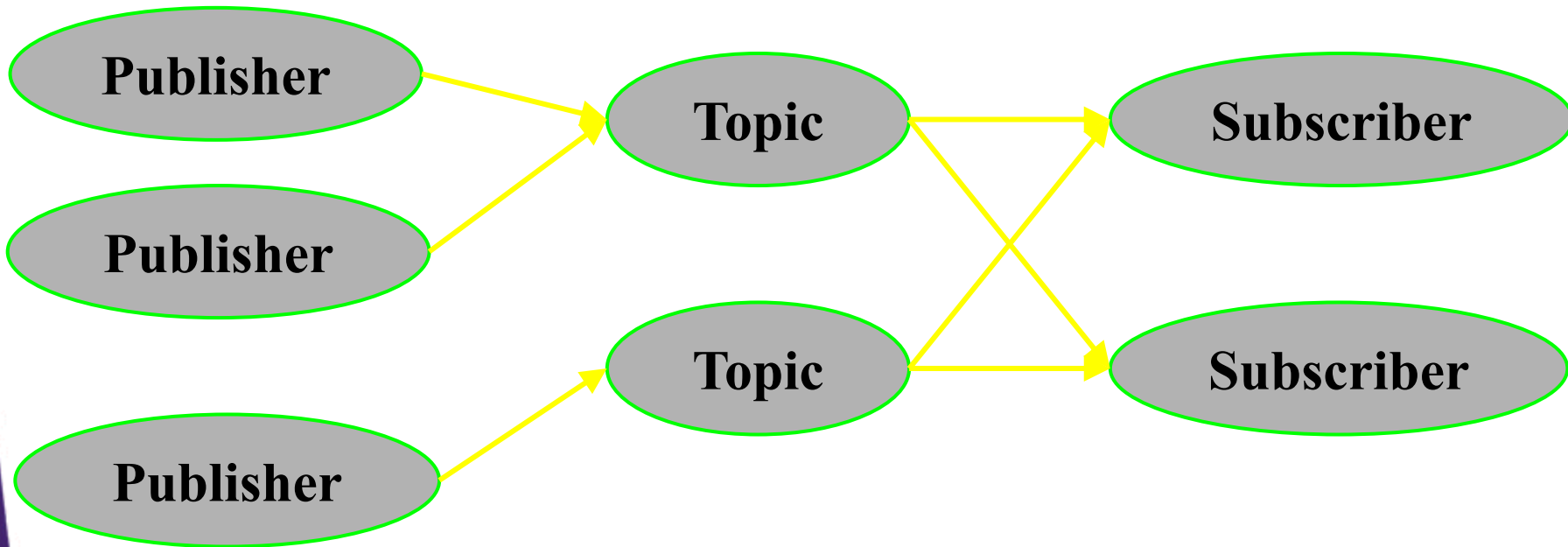
- In the Publish-Subscribe model, a **publisher** publishes a message to a particular **destination**, called a **topic**
- A **subscriber** registers interest by **subscribing** to that **topic**
- **Multiple publishers** can **publish** messages to the topic, and
- **Multiple subscribers** can subscribe to the topic and consume from the topic.

# Publish-Subscribe Messaging Model

- **By default, a subscriber will receive messages only when it is active.**
  - However, a subscriber **may establish a durable connection**, so that any messages published while the subscriber is not active are **redistributed** whenever it reconnects.
- **The publisher and subscriber are loosely coupled from each other**
  - They have no knowledge of each other's existence. They only need to **know** the **destination** and the **message format**.
- **Different levels of quality-of-service can be configured, such as**
  - Missed, or duplicate messages, or deliver-once

# Communication/Messaging Models

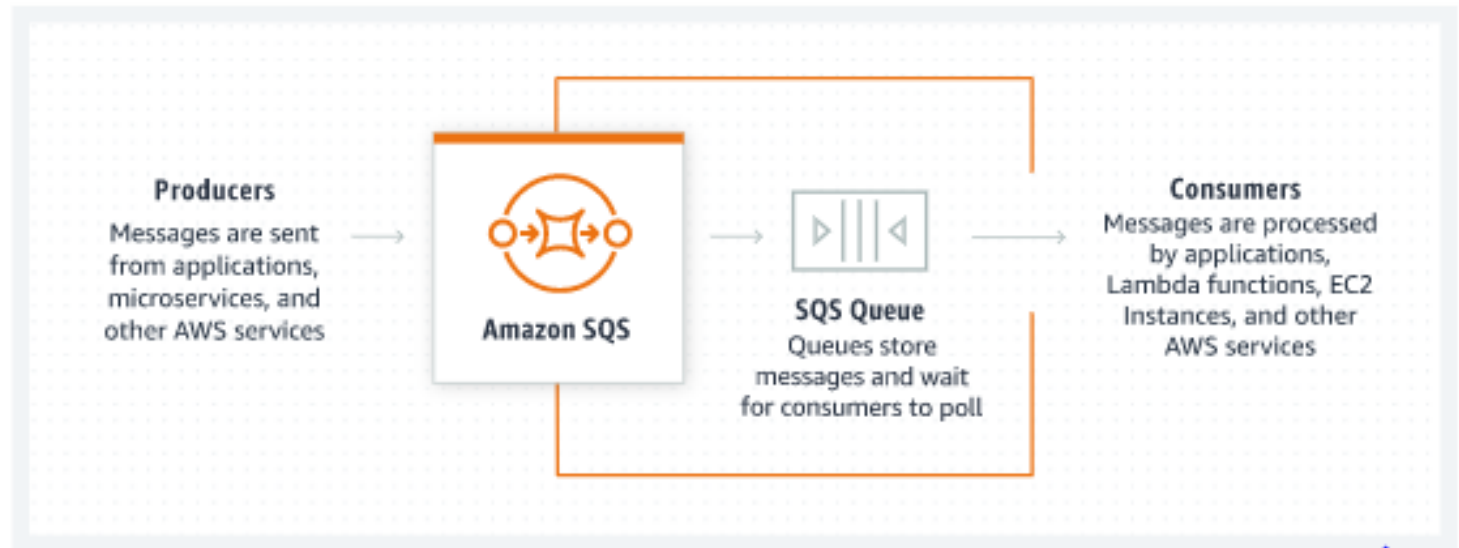
- **Publish/Subscribe model**
  - A message can be delivered to more than one client



# Messaging Support on AWS

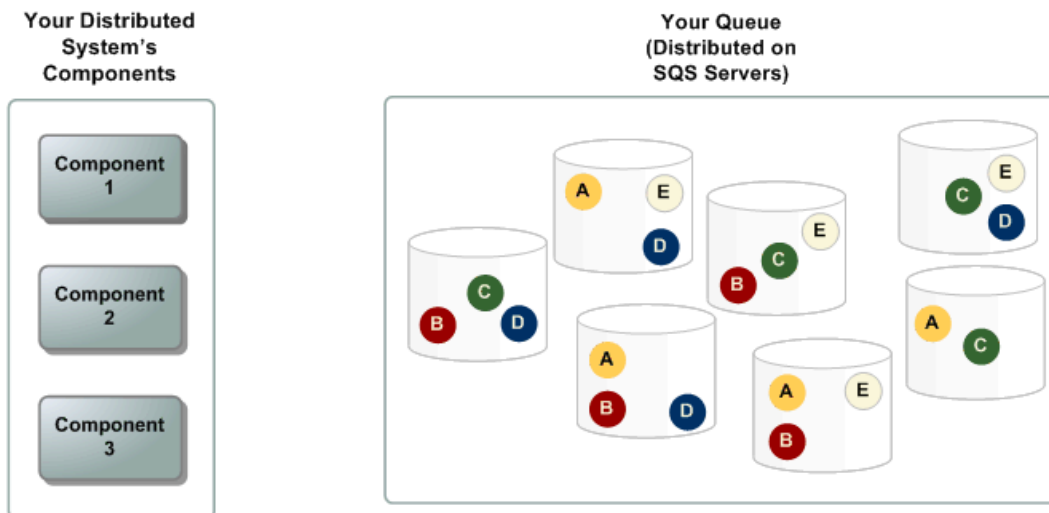
- **Amazon SQS**

- A fast, reliable, scalable, fully managed message queuing service
- Amazon SQS can be used to transmit relatively large volume of data
- Messages in a single queue are redundantly saved across multiple Amazon SQS servers



# Basic SQS Architecture: Distributed Queue

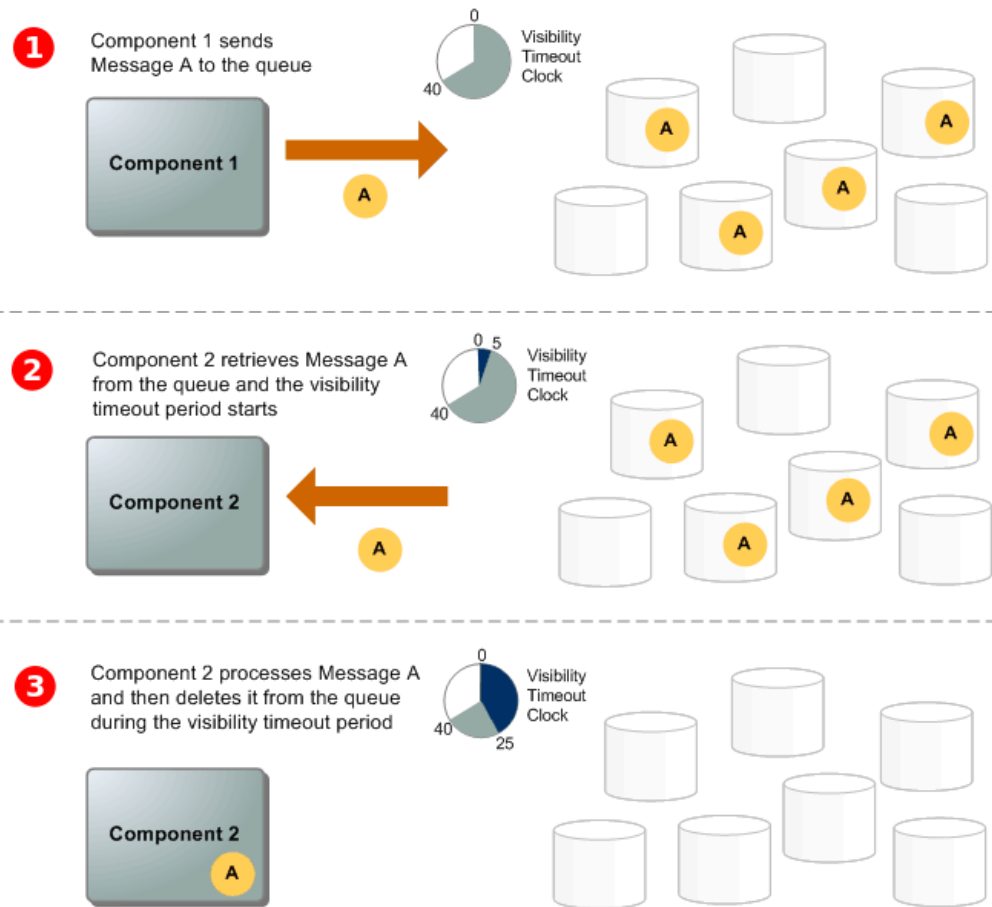
- **There are three main parts in a distributed messaging system:**
  - the components of your distributed system,
    - producers (components that send messages to the queue)
    - consumers (components that receive messages from the queue).
  - your queue (distributed on Amazon SQS servers), and
  - the messages in the queue.
- **In the following scenario, the system has several producers and consumers**
- **The queue () redundantly stores the messages across multiple Amazon SQS servers.**
  - which holds messages A through E



# Basic SQS Architecture: Message Life Cycle

The following scenario describes the lifecycle of an Amazon SQS message in a queue, from creation to deletion.

- A producer (component 1) sends message A to a queue, and the message is distributed across the Amazon SQS servers redundantly.
- When a consumer (component 2) is ready to process messages, it consumes messages from the queue, and message A is returned. While message A is being processed, it remains in the queue and isn't returned to subsequent receive requests for the duration of the visibility timeout.
- The consumer (component 2) deletes message A from the queue to prevent the message from being received and processed again when the visibility timeout expires.
- Amazon SQS automatically deletes messages that have been in a queue for more than the maximum message retention period. The default message retention period is 4 days. However, you can set the message retention period to a value from 60 seconds to 1,209,600 seconds (14 days) using the `SetQueueAttributes` action.





# Messaging Support on AWS

- **SQS Benefits and Features**

## Highly scalable Standard and FIFO queues

Queues scale elastically with your application. Nearly unlimited throughput and no limit to the number of messages per queue in Standard queues. First-In-First-Out delivery and exactly once processing in FIFO queues.

## Durability and availability

Your queues are distributed on multiple servers. Redundant infrastructure provides highly concurrent access to messages.

## Security

Protection in transit and at rest. Transmit sensitive data in encrypted queues. Send messages in a Virtual Private Cloud.

## Batching

Send, receive, or delete messages in batches of up to 10 messages or 256KB to save costs.

Source: <https://console.aws.amazon.com/sqs/>

# Messaging Support on AWS

- **Common SQS Features**

- **Reliable**

- It runs within Amazon's high-availability data centers

- **Simple**

- Only **three APIs** to start : SendMessage, ReceiveMessage, and DeleteMessage

- **Flexible**

- **Standard queues** for high throughput and **FIFO queues** for strict ordering

- **Scalable**

- **Automatically scales** and support for unlimited queue and messages available

- **Secure**

# Messaging Support on AWS

- **Additional SQS Features**

- Designed to provide high durability.
- Hold messages until you explicitly delete them
- Unlimited backlog up to 14 days
- Amazon CloudWatch metrics and alerts for queue depth, message rate and more.
- Payload size up to 256KB and more
- Message batching for higher throughput and reduced cost
- Support long polling for reduced cost and latency
- Cross-origin resource sharing support

# Messaging Support on AWS

## ● Create a Queue

Use the AmazonSQS client's `createQueue` method, providing a `CreateQueueRequest` object that describes the queue parameters.

### Imports

```
import com.amazonaws.services.sqs.AmazonSQS;  
import com.amazonaws.services.sqs.AmazonSQSClientBuilder;  
import com.amazonaws.services.sqs.model.AmazonSQSException;  
import com.amazonaws.services.sqs.model.CreateQueueRequest;
```

### Code

```
AmazonSQS sqs = AmazonSQSClientBuilder.defaultClient();  
CreateQueueRequest create_request = new CreateQueueRequest(QUEUE_NAME)  
    .addAttributesEntry("DelaySeconds", "60")  
    .addAttributesEntry("MessageRetentionPeriod", "86400");  
  
try {  
    sqs.createQueue(create_request);  
} catch (AmazonSQSException e) {  
    if (!e.getErrorCode().equals("QueueAlreadyExists")) {  
        throw e;  
    }  
}
```

You can use the simplified form of `createQueue`, which needs only a queue name, to create a standard queue.

```
sqs.createQueue("MyQueue" + new Date().getTime());
```

# Messaging Support on AWS

- **Sending a messages to Queue**

## Imports

```
import com.amazonaws.services.sqs.AmazonSQS;  
import com.amazonaws.services.sqs.AmazonSQSClientBuilder;  
import com.amazonaws.services.sqs.model.SendMessageRequest;
```

## Code

```
SendMessageRequest send_msg_request = new SendMessageRequest()  
    .withQueueUrl(queueUrl)  
    .withMessageBody("hello world")  
    .withDelaySeconds(5);  
sqs.sendMessage(send_msg_request);
```

- **Receive a messages to Queue**

## Imports

```
import com.amazonaws.services.sqs.AmazonSQSClientBuilder;  
import com.amazonaws.services.sqs.model.AmazonSQSException;  
import com.amazonaws.services.sqs.model.SendMessageBatchRequest;
```

## Code

```
List<Message> messages = sqs.receiveMessage(queueUrl).getMessages();
```

# Messaging Support on AWS

- Types of Queues

Standard Queue	FIFO Queues
Unlimited number of transactions per second	First-in-first-out delivery, limited to 300 transactions per second
Guarantee that a message is delivered at least once	Exactly-once processing
Ex: Schedule multiple entries to be added to a database.	Ex: Prevent a student from enrolling in a course before registering for an account.

# Messaging Support on AWS

- Creating Queues using Console


Amazon SQS > Queues > Create queue

## Create queue

### Details

#### Type

Choose the queue type for your application or cloud infrastructure.

 You can't change the queue type after you create a queue.

#### ☒ Standard [Info](#)

At-least-once delivery, message ordering isn't preserved

- At-least once delivery
- Best-effort ordering

#### ☐ FIFO [Info](#)

First-in-first-out delivery, message ordering is preserved

- First-in-first-out delivery
- Exactly-once processing

#### Name

MyQueue

A queue name is case-sensitive and can have up to 80 characters. You can use alphanumeric characters, hyphens (-), and underscores (\_).

### Configuration

Set the maximum message size, visibility to other consumers, and message retention. [Info](#)

#### Visibility timeout [Info](#)

30

Seconds ▼

Should be between 0 seconds and 12 hours.

#### Delivery delay [Info](#)

0

Seconds ▼

Should be between 0 seconds and 15 minutes.

#### Message retention period [Info](#)

4

Days ▼

Should be between 1 minute and 14 days.

#### Maximum message size [Info](#)

256

KB

Should be between 1 KB and 256 KB.

#### Receive message wait time [Info](#)

0

Seconds ▼

Should be between 0 seconds and 30 seconds.

# Messaging Support on AWS

- Creating Queues using Console (contd.)

### Access policy

Define who can access your queue. [Info](#)

Choose method

☒ Basic  
Use simple criteria to define a basic access policy.

☐ Advanced  
Use a JSON object to define an advanced access policy.

Define who can send messages to the queue

☒ Only the queue owner  
Only the owner of the queue can send messages to the queue.

☐ Only the specified AWS accounts, IAM users and roles  
Only the specified AWS account IDs, IAM users and roles can send messages to the queue.

Define who can receive messages from the queue

☒ Only the queue owner  
Only the owner of the queue can receive messages from the queue.

☐ Only the specified AWS accounts, IAM users and roles  
Only the specified AWS account IDs, IAM users and roles can receive messages from the queue.

JSON (read-only)

```
{
  "Version": "2008-10-17",
  "Id": "__default_policy_ID",
  "Statement": [
    {
      "Sid": "__owner_statement",
      "Effect": "Allow",
      "Principal": {
        "AWS": "492835594071"
      },
      "Action": [
        "SQS:*"
      ],
      "Resource": "arn:aws:sqs:us-east-1:492835594071:"
    }
  ]
}
```

► **Encryption - Optional**

Amazon SQS provides in-transit encryption by default. To add at-rest encryption to your queue, enable server-side encryption. [Info](#)

► **Dead-letter queue - Optional**

Send undeliverable messages to a dead-letter queue. [Info](#)

► **Tags - Optional**

A tag is a label assigned to an AWS resource. Use tags to search and filter your resources or track your AWS costs. [Learn more](#) [↗](#)

Cancel

Create queue



# Messaging Support on AWS

- **Amazon SNS**

“Amazon Simple Notification Service (Amazon SNS) is a fast, flexible, fully managed push notification service that lets you send individual messages or to fan-out messages to large numbers of recipients”

Amazon SNS makes it simple and cost effective to send push notifications to mobile device users, email recipients or even send messages to other distributed services.



# Messaging Support on AWS

- **Amazon SNS: Features and Benefits**

## Reliably deliver messages with durability

Amazon SNS uses cross availability zone message storage to provide high message durability. Amazon SNS reliably delivers messages to valid AWS endpoints, such as Amazon SQS queues and AWS Lambda functions.

## Automatically scale your workload

Amazon SNS leverages the proven AWS cloud to dynamically scale with your application. Amazon SNS is a fully managed service, taking care of the heavy lifting related to capacity planning, provisioning, monitoring, and patching.

## Simplify your architecture with Message Filtering

Amazon SNS helps you simplify your pub/sub messaging architecture by offloading the message filtering logic from your subscriber systems, and message routing logic from your publisher systems.

## Keep messages private and secure

Amazon SNS topic owners can set topic policies that restrict who can publish and subscribe to a topic. Amazon SNS also ensures that data is encrypted in transit and at rest, and provides VPC endpoints for message privacy.

.

# Messaging Support on AWS

- **Amazon SNS**

Application-to-application (A2A)

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



# Messaging Support on AWS

- **Amazon SNS**

## 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.



# Messaging Support on AWS

- **Amazon SNS**

- Deliver messages instantly to applications or users and eliminate polling in your apps.
- Scale as you needs grow
- Send messages to individual devices or broadcast to multiple destinations at once.
- SDK available in iOS, Android, Java, Python, PHP, Node.js or .NET.
- Send notification to Apple, Android and other mobile devices and also destinations like SQS, Lambda and email address SMS and http endpoints.
- You can get delivery status information via cloud watch success and failure rates.

Developers can get started with Amazon SNS by using just three APIs:

CreateTopic,  
Subscribe, and  
Publish.

Additional APIs are available, which provide more advanced functionality.

Supported Transport protocols for subscribers :

- "HTTP", "HTTPS" (POST)
- "Email", "Email-JSON"
- "SQS" – Note that FIFO queues are not currently supported.
- "SMS Text Messages"