# Welcome

Hey there!

Thank you for downloading and reading this guide. The guide contains resources that will help you learn about the most popular AWS services.

The services currently covered in the guide are:

- Amazon Simple Storage Service (S3)
- Amazon Simple Queue Service (SQS)
- AWS Lambda
- Amazon DynamoDB
- Amazon Route 53

This guide is a living a document and I will keep adding more services to it over time.

I hope you find this useful and if you have any feedback, please don't hesitate to reach out to me via Twitter.

Happy reading!

Abhishek

# Amazon Simple Storage Service (S3)

## What is it?

Amazon Simple Storage Service, commonly known as S3, is a fast, scalable, and durable object-storage service. S3 can be used to store and retrieve any type and any amount of data.

## How does it work?

At its core, S3 is an object-storage service. which is different from the traditional file-storage service. Data in S3 is stored as objects. Each object contains a unique identifier, some metadata about the object and the data itself.

### Key Concepts

#### Buckets

An S3 bucket is conceptually similar to a folder in a file-storage system. Objects in S3 are stored within a bucket. An S3 bucket needs to be created before data can be stored in S3.

For e.g. if there is an object with the key `omgcat.png` in the S3 bucket `adorable-cat-photos`, then the addressable path of the object is `s3://adorable-cat-photos/omgcat.png`.

Buckets are important to understand for some of the following reasons:

- S3 bucket names are globally unique across all AWS accounts. For e.g. if a bucket with the name `adorable-cat-photos` already exists, nobody else will be able to create a bucket with this name.
- Access Control can be implemented at bucket level
- AWS billing is based on aggregate bucket sizes

#### Object keys

To create an object in S3, a key must be specified. This key uniquely identifies an object within a bucket. Since S3 is an object-storage service with a flat namespace (no hierarchy), it has no concept of folders.

The following are all valid keys for an object:

```
omgcatphoto.png
catvideos/omgwhatacat.mp4
photos/2020/11/11/photo-of-the-day.png
```

#### Object Metadata

There are two kinds of metadata associated with an object: *system metadata* and *user-defined metadata*. User-defined metadata can be added when an object is created or updated.

Some examples of system metadata are:

- Object creation date
- Storage class for the object
- Object size in bytes

### Storage classes

S3 provides multiple storage classes which are designed for different use-cases.

#### Standard

- Ideal for frequently accessed or performance-critical data
- Most expensive storage class

#### Intelligent-Tiering

- Automatically moves objects between access tiers based on access patterns
- Good for use-cases when access patterns are ambiguous

#### Standard Infrequent-access

- Ideal for long-lived and less frequently access data
- Storage is cheaper than the *Standard* class but there is a retrieval fee for data access

#### Glacier

- Ideal for long-term archiving.
- Configurable retrieval times (minutes to hours).

More information about Storage classes can be found here.

## When to use it?

S3 is a flexible storage service and thus can be used for a variety of use-cases. Some of the common use-cases are:

- **Storing static content and serving it directly to end-users**: Common examples of this are static webpages, images, videos, static web assets such as CSS or Javascript assets. S3 can also be configured with CloudFront (Amazon's CDN) to improve delivery performance for such content.
- **Data Lake**: S3 is ideal for storing raw, unstructured data in any format and thus can be used as the storage layer for building a Data Lake
- **Logs, Backups, and snapshots**: S3's infrequent access tier makes storing logs, backups, and snapshots a good-fit. Some of the services which integrate with S3 are RDS, EBS, and CloudTrail.

## Examples

Some examples of how to use S3 for various use-cases:

- How to create a blog on AWS using S3 in 3 easy steps
- Build Your Data Lake on Amazon S3
- Amazon RDS Snapshot Export to S3

## Getting Started

The following examples will take us through some of the more common operations for S3.

## Creating a bucket

**CLI**

```
aws s3 mb s3://bucket-name
```

**Python (boto3)**

```
s3_client = boto3.client('s3')
s3_client.create_bucket(Bucket=bucket_name)
```

List buckets and objects

**CLI**

```
# List all buckets
aws s3 ls

# List objects within bucket
aws s3 ls s3://bucket-name
```

**Python (boto3)**

```
s3_client = boto3.client('s3')

# List all buckets
s3_client.list_buckets()

# List objects within bucket
s3_client.list_objects(Bucket=bucket_name)
```

## Delete buckets

**CLI**

```
aws s3 rb s3://bucket-name
```

**Python (boto3)**

```
```

```
s3_client = boto3.client('s3')
s3_client.delete_bucket(Bucket=bucket_name)
```

## Delete objects

### CLI

```
aws s3 rm s3://bucket-name/object-key
```

### Python (boto3)

```
s3_client = boto3.client('s3')
s3_client.delete_object(Bucket=bucket_name, Key=object_key)
```

## Copy objects

The following command can be used to move objects from a bucket or a local directory

```
# Copy from one bucket to another
aws s3 cp s3://old-bucket/example s3://new-bucket/

# Copy from local directory to bucket
aws s3 cp /tmp/filename.txt s3://bucket-name
```

### Python (boto3)

```
s3_client = boto3.client('s3')

# Copy object from one bucket to another
s3_client.copy_object(
  Bucket=destination_bucket,
  CopySource={"Bucket": original_bucket, "Key": object_key"}
)

# Copy object from local directory to S3
s3_client.upload_file(
  Filename=local_file_path,  # /tmp/filename.txt
  Bucket=bucket,  # bucket-name
  Key=file_key,  # filename.txt
)
```

Additional documentation for the CLI can be found here whereas documentation for the boto library is available here.

# Amazon Simple Queue Service (SQS)

## What is it?

Amazon Simple Queue Service (SQS) is a managed, message-queue service that enables us to build scalable and reliable systems. Queues allow services to be decoupled. They can communicate with each other asynchronously and are especially useful when the throughput of the producing service is different from the throughput of the consuming service.

## How does it work?


SQS provides a message-queue service. To use SQS, you need the following components:

- **Producer(s)**: Producers are responsible for sending messages to a particular queue. Messages are stored in the queue when they are sent by the producer.
- **Consumer(s)**: Consumers are responsible for retrieving and processing the messages from a particular queue. Messages must be deleted by the consumer after processing to ensure they aren't processed by any other consumers.

### Key Concepts

- **SQS Visibility Timeout**: Configurable period of time when a message received by one consumer is protected from other consumers. The default timeout is 30 seconds.
- **Standard vs FIFO queue**: SQS provides two types of queues: Standard and FIFO. Standard queues provide best-effort ordering whereas FIFO queues provide first-in first-out delivery.
- **Dead-letter Queues**: Dead-letter Queues (DLQ) are used to store messages that couldn't be processed successfully by the consumer. DLQs provide multiple benefits. They can be used to debug any issues with the processing of problematic messages. Also, they allow applications to continue processing the rest of the messages which don't have any issues.

### Key Limits

- **Message Retention**: Message retention is configurable between 1 minute to 14 days. The default is 4 days.
- **Message Limit**: A single SQS queue can contain an unlimited number of messages. However, there are limits to the number of inflight messages (received by a consumer but not yet deleted).
- **Maximum size of one message**: Maximum allowed size of a single message is 256 KB.
- **Message format**: Messages can include text data, including XML, JSON and unformatted text.

## When to use it?

SQS is a great fit for building reliable and scalable systems. Some use-cases where SQS fits in well:

**Event-based architectures**

An event-driven architecture uses event to trigger and communicate between decoupled services. The key component of any event-driven architecture is a queue.

**Microservices-based architectures**

Asynchronous communication is becoming more common in microservices-based architectures where different microservices are decoupled and independent of each other.

## How much does it cost?

SQS' pricing model is based on how much resources are used. The screenshots below show the pricing as of Nov 2020.

### How are Amazon SQS requests priced?

The first 1 million monthly requests are free. After that, the pricing is as follows for all regions:

Region: US East (Ohio)

| | Pricing per 1 million Requests after Free tier (Monthly) |
|---|---|
| Standard Queue | $0.40 ($0.0000004 per request) |
| FIFO Queue | $0.50 ($0.0000005 per request) |

### How are Amazon SQS charges metered?

| | |
|---|---|
| **API Actions** | Every Amazon SQS action counts as a request. |
| **FIFO Requests** | API actions for sending, receiving, deleting, and changing visibility of messages from FIFO queues are charged at FIFO rates. All other API requests are charged at standard rates. |
| **Contents of Requests** | A single request can have from 1 to 10 messages, up to a maximum total payload of 256 KB. |
| **Size of Payloads** | Each 64 KB chunk of a payload is billed as 1 request (for example, an API action with a 256 KB payload is billed as 4 requests). |
| **Interaction with Amazon S3** | When using the Amazon SQS Extended Client Library to send payloads using Amazon S3, you incur Amazon S3 charges for any Amazon S3 storage you use to send message payloads. |
| **Interaction with AWS KMS** | When using the AWS Key Management Service to manage keys for SQS server-side encryption, you incur charges for calls from Amazon SQS to AWS KMS. For more information see KMS pricing and How Do I Estimate My AWS KMS Usage Costs? in the *Amazon SQS Developer Guide*. |

SQS Pricing Model — Tips & Alternatives provides some useful tips on working with SQS.

# Examples

The following resources provide examples of applications which have been built using SQS:

- Scalable serverless event-driven applications using Amazon SQS & Lambda
- Trax Retail: An Innovative Approach to Per-Second Scaling for SNS/SQS Message Processing
- Decouple and Scale Applications Using Amazon SQS and Amazon SNS - 2017 AWS Online Tech Talks

# Getting Started

## Creating a queue

### CLI

```
aws sqs create-queue --queue-name your-queue-name
```

### Python (boto3)

```
sqs_client = boto3.client('sqs')
sqs_client.create_quque(QueueName="your-queue-name")
```

List queues

### CLI

```
# List all queues
aws sqs list-queues
```

### Python (boto3)

```
sqs_client = boto3.client('sqs')
sqs_client.list_queues()
```

## Delete queue

### CLI

```
aws sqs delete-queue --queue-url https://sqs.us-east-
1.amazonaws.com/myaccountid/your-queue-name
```

### Python (boto3)

```
sqs_client = boto3.client('sqs')
sqs_client.delete_queue(QueueUrl='https://sqs.us-east-
1.amazonaws.com/myaccountid/your-queue-name')
```

### Send message

**CLI**

```
aws sqs send-message --queue-url https://sqs.us-east-
1.amazonaws.com/myaccountid/your-queue-name --message-body "Sending a
message on the queue."
```

**Python (boto3)**

```
sqs_client = boto3.client('sqs')
sqs_client.send_message(
  QueueUrl='https://sqs.us-east-1.amazonaws.com/myaccountid/your-queue-
name',
  MessageBody='Sending a message on the queue.',
)
```

### Receive message

**CLI**

```
# Receive 1 message
aws sqs receive-message --queue-url https://sqs.us-east-
1.amazonaws.com/myaccountid/your-queue-name
```

**Python (boto3)**

```
sqs_client = boto3.client('sqs')
messages = sqs_client.receive_message(QueueUrl='https://sqs.us-east-
1.amazonaws.com/myaccountid/your-queue-name')
```

### Delete message

A message can only be deleted after it has been received.

**CLI**

```
```

```
# receipt-handle is sent when the message is received

aws sqs delete-message --queue-url https://sqs.us-east-
1.amazonaws.com/myaccountid/your-queue-name --receipt-handle
AQEBRXTo...q2doVA==
```

**Python (boto3)**

```
sqs_client = boto3.client('sqs')
sqs_client.delete_message(
    QueueUrl='https://sqs.us-east-1.amazonaws.com/myaccountid/your-queue-
name',
    ReceiptHandle='AQEBRXTo...q2doVA=='
)
```

CLI documentation is available **here**.

Boto3 documentation is available **here**.

---

# AWS Lambda

## What is it?

AWS Lambda is a service that lets developers run their code in the cloud without having to host any servers. AWS Lambda is a foundational service in the serverless paradigm where code is only run when needed and developers only pay for the resources used.

Some of the reasons why AWS Lambda is one of the most popular AWS services:

- No need to manage any servers
- Your application scales automatically to handle the size of the workload

## How does it work?

AWS Lambda provides Functions as a service (FaaS) which developers leverage to deploy functions that are run in response to various events.

### Key concepts

#### Function

A function is code provided by the developer that runs in AWS Lambda. A function processes the invocation events sent by AWS Lambda. The function takes two arguments:

- **Event object**: contains details about the invocation event.
- **Context object:** contains information about the Lambda runtime, such as the function name, memory limit etc.

#### Execution environment

Lambda provides a secure and isolated runtime environment where your function is invoked. The execution environment manages the resources required to run the function.

Additional information is available here.

#### Runtime

AWS Lambda supports functions in multiple languages through the use of runtimes. A runtime is chosen when a function is created.

Additional information about Lambda runtimes is available here.

#### Trigger

Lambda functions are invoked as a response to certain actions. These actions are called triggers. Lambda functions can be triggered by other AWS services or your own applications. For e.g. you could trigger a function for a new object in S3.

#### Concurrency

Concurrency is the number of requests that a Lambda function is serving at any given time.

**Cold Starts**

A cold start happens when a Lambda function is invoked after not being used for a while. Cold starts generally result in increased latency.

**Provisioned Concurrency**

Ability to keep lambda functions initialized and ready to respond to request. Reduces the cold-start problem.

## Resource Limits

AWS Lambda has some resource limits which are useful to know about. Some of the more common resource limits are:

- Maximum execution time is 15 minutes
- Maximum memory allowed is 3008 MB
- Maximum deployment package size is 50 MB (zipped)

All the resource limits (as of Nov 2020) are shown below:

| Resource | Quota |
| --- | --- |
| Function memory allocation | 128 MB to 3,008 MB, in 64 MB increments. |
| Function timeout | 900 seconds (15 minutes) |
| Function environment variables | 4 KB |
| Function resource-based policy | 20 KB |
| Function layers | 5 layers |
| Function burst concurrency | 500 - 3000 (varies per region) |
| Invocation payload (request and response) | 6 MB (synchronous)<br>256 KB (asynchronous) |
| Deployment package size | 50 MB (zipped, for direct upload)<br>250 MB (unzipped, including layers)<br>3 MB (console editor) |
| Test events (console editor) | 10 |
| `/tmp` directory storage | 512 MB |
| File descriptors | 1,024 |
| Execution processes/threads | 1,024 |

## When to use it?

AWS Lambda forms the core of the serverless architecture. Lambda lets us execute custom functions in response to the occurrence of certain events. AWS Lambda is a good fit for the following use-cases:

- Chatbots
- Image / Video Processing
- Websites
- ETL jobs

https://www.simform.com/serverless-examples-aws-lambda-use-cases/

## How much does it cost?

With AWS Lambda, you are only charged for what you use. The pricing is based on the **number of requests** for your functions and the **amount of time** it takes for your code to execute.

More details about pricing can be found here.

# AWS Lambda Pricing

Region: US East (Ohio)

| | Price |
|---|---|
| Requests | $0.20 per 1M requests |
| Duration | $0.0000166667 for every GB-second |

The price for **Duration** depends on the amount of memory you allocate to your function. You can allocate any amount of memory to your function between 128MB and 3008MB, in 64MB increments. The table below contains a few examples of the price per 100ms associated with different memory sizes.

| Memory (MB) | Price per 100ms |
|---|---|
| 128 | $0.0000002083 |
| 512 | $0.0000008333 |
| 1024 | $0.0000016667 |
| 1536 | $0.0000025000 |
| 2048 | $0.0000033333 |
| 3008 | $0.0000048958 |

## Examples

Wix: Serverless Platform for End-to-End Browser Testing using Chromium on AWS Lambda

Wix built a remote end-to-end browser testing platform using AWS Lambda. This platform can run 700 different tests in parallel.

Innovapost: Scaling to 5M Package Deliveries with Serverless

Innovapost built its package delivery pipeline on top of SQS & Lambda. Delivery messages are sent to SQS which are then processed by Lambda and then written to RDS.

## Additional Resources

The following presentations from the re:Invent conference provide good insights into how Lambda works under the hood as well some of the best practices for Lambda:

- A serverless journey: AWS Lambda under the hood
- Asynchronous-processing best practices with AWS Lambda
- Building microservices with AWS Lambda

---

# Amazon DynamoDB

## What is it?

Amazon DynamoDB is a managed NoSQL database service. DynamoDB provides a simple API to store, access, and retrieve data.

Some of the reasons why DynamoDB is popular:

- **Schemaless:** To create a new table, only the primary key attributes need to be defined.
- **On-demand capacity**: DynamoDB scales up/down automatically to handle the traffic

## How does it work?

### Key concepts

#### Tables

A table is a collection of items. For e.g. you could have an Employee table which stores information about every employee at a company.

#### Items

An Item is a single record in a table. An item is uniquely identified by its Primary Key. In our previous example, an Employee would be an item in the Employees table. The primary key or unique identifier could be their Employee ID.

#### Attributes

An attribute is a field or piece of data attached to an item. Examples of attributes attached to an Employee item could be Name, Age, Office Location, etc.

#### Primary Key

A primary key is a unique identifier and is used to uniquely identify each item in the table. The primary key is the only required attribute when creating a new table. It can not be empty or null.

There are two types of primary key:

- **Partition key**: This is a simple primary key that is unique to each item in the table.
- **Composite primary key:** This is a combination of partition and sort key, which together is unique to each item in the table.

Choosing the **right primary key** is critical for the optimal performance of DynamoDB. This guide provides good insight into how to choose the right key for your application.

#### Secondary Index

Secondary indices provide the ability to query a table without using the primary key. An application generally benefits from different access patterns and secondary indices enable efficient data access without using the primary key.

A secondary index consists of a subset of attributes from the table.

**Provisioned & On-Demand**

DynamoDB provides multiple pricing models:

- **On-demand capacity**: The simplest or most straightforward pricing model. Pricing is based on storage and requests.
- **Provisioned capacity**: In the provisioned mode, read and write throughput capacity needs to be set for each table. The read and write capacity specify the allowed operations per second on the table.

This article is a good resource to understand which model might be appropriate for your application.

## When to use it?

DynamoDB can be a good fit for the following use-cases:

**Serverless applications**

DynamoDB works well with other serverless services like AWS Lambda and is often an integral part of serverless applications.

Amazon DynamoDB and Serverless - The Ultimate Guide is a great resource to learn more about how DynamoDB can be used to build serverless applications.

**Applications with access patterns which are compatible with a key-value store**

DynamoDB is a key-value store and doesn't support relational data structures. If your data is self-contained and you won't need JOINs across multiple tables to query your data, then DynamoDB might be a good fit.

### Additional Resources

Some additional resources to understand when to use DynamoDB

- Why Amazon DynamoDB isn't for everyone
- 11 Things You Wish You Knew Before Starting with DynamoDB

## Examples

Handling AWS Chargebacks for Enterprise Customers

Building enterprise applications using Amazon DynamoDB, AWS Lambda, and Go

Event-driven processing with Serverless and DynamoDB streams

## Getting Started

Helpful resources to get started with DynamoDB:

- DynamoDB, explained
- Data modeling with Amazon DynamoDB
- Amazon DynamoDB deep dive: Advanced design patterns

---

# Amazon Route 53

## What is it?

AWS Route 53 is a highly available and scalable Domain Name System (DNS) web service. It helps route internet traffic to the appropriate servers hosting the requested web application by translating the domain name into IP addresses.

Route 53 also provides the ability to register domains directly from the AWS console as well.

## What are the key features of Route 53?

- **Highly scalable**: It can handle millions of requests
- **Highly reliable**: AWS provides a guarantee of at least 99.99% up time during any monthly billing cycle
- **DNS Failover**: Route 53 can automatically detect an outage using health checks and route queries to the resources that are healthy
- **Traffic Flow:** Ability to route traffic intelligently that can be configured based on various parameters like health of endpoints, latency, among others.
- **Private DNS for Amazon VPC**: Route53 can be configured to respond to DNS queries within private hosted VPC zones.
- **Domain Registration:** Route 53 allows you to manage your domains (buying or transferring) from the AWS console.
- **ELB Integration:** Route 53 integrates natively with AWS Elastic Load Balancers and increases the fault-tolerance & reliability of the application.

### Key Concepts

- **Hosted Zone:** It represents a collection of records that are managed together and these records belong to a single parent domain. All resource records within a hosted zone should end with the hosted zone's domain as a suffix.
- **Record**: An entry in a hosted zone that defines how traffic is routed for the domain or subdomain.
- **Alias record**: A type of record available with AWS Route 53 that lets you route traffic to AWS resources like Cloudfront distributions, S3 buckets and ELBs.
- **Routing Policy:** A configuration that determines how Route 53 responds to DNS queries
- **Private DNS**: A local version of DNS that lets you route traffic for a domain and its subdomains within one or more AWS VPCs.

### Types of Routing Policy

- **Simple**: Default routing policy. Used to route traffic to a single resource. For e.g. a web server that serves content for a particular website.
- **Failover:** Can be used to configure active-passive failover. For e.g. if a particular resource becomes unhealthy, traffic can be routed to a different healthy resource.
- **Geolocation:** Can be used to route traffic on the basis of the geographic location of the user.
- **Geoproximity:** Can be used to route traffic on the basis of the location of your resources

- **Latency:** Can be used when you have resources in multiple locations and you want to route traffic to the resource that provides the best latency.
- **Weighted**: Can be used to route traffic to multiple resources in the proportion defined by the user.

## How much does it cost?

With AWS Route 53, you are only charged for what you use. The pricing is based on:

- Number of hosted zones: There is a charge for each hosted zone within Route 53
- Serving DNS queries: There is a charge for every DNS query answered by the AWS Route 53 (except for Alias records under certain circumstances).
- Managing domain names: There is an annual charge for each domain name registered via Route 53

More details about pricing can be found here.

## Examples

SimilarWeb: Route 53 calculated health checks for an active/active multi-region architecture

SimilarWeb, an analytics platform, leverage Route 53 health checks to build an active/active multi-region architecture that is resilient to the failure of a single region.

Netflix: Multi-region resiliency and Amazon Route 53

Netflix built a resilient system leveraging Amazon Route 53's georouting, Alias records and failover capabilities.

---