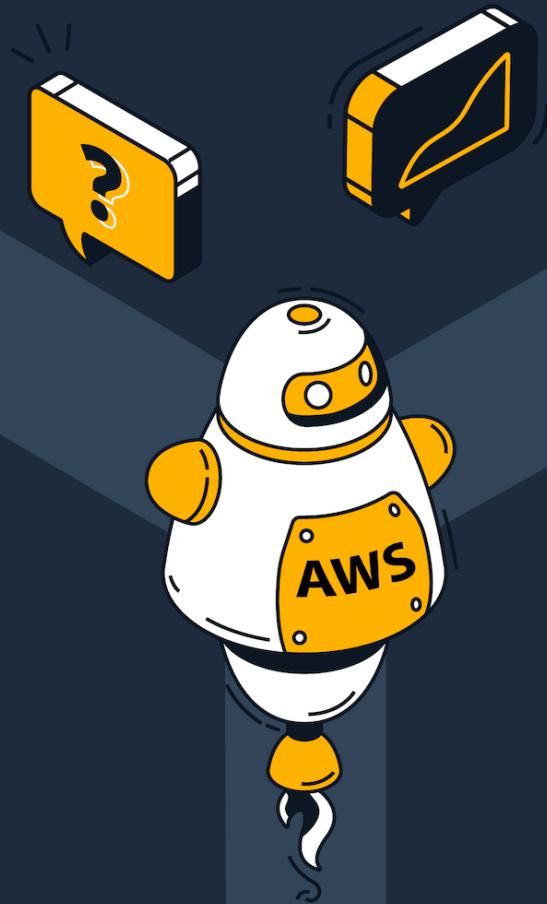


# AWS Fundamentals

Tobias Schmidt & Alessandro Volpicella



AWS for the Real World  
Not Just for Certifications





# AWS Fundamentals

AWS for the Real World - Not Just for Certifications

**Tobias Schmidt**

**Alessandro Volpicella**

# Table of Contents

- Introduction
  - About the Scope of This Book
  - Why Did We Bother to Write This?
  - Who Is This Book For
  - Who Is This Book Not For
- Getting Started
  - Creating Your Own AWS Account
  - Account Security Key Concepts and Best Practices
  - Avoiding Cost Surprises
  - Understanding the Shared Responsibility Model
  - About going Serverless and Cloud-Native
- AWS Core Building Blocks for all Applications
  - AWS IAM for Controlling Access to Your Account and Its Resources
  - Compute
    - Launching Virtual Machines in the Cloud for Any Workload with EC2
    - Running and Orchestrating Containers with ECS and Fargate
    - Using Lambda to Run Code without Worrying about Infrastructure
  - Database & Storage
    - Fully-Managed SQL Databases with RDS
    - Building Highly-Scalable Applications in a True Serverless Way With DynamoDB
    - S3 Is a Secure and Highly Available Object Storage
  - Messaging
    - Using Message Queues with SQS
    - SNS to Build Highly-Scalable Pub/Sub Systems
    - Building an Event-Driven Architecture with AWS EventBridge
  - Networking
    - Exposing Your Application's Endpoints to the Internet via API Gateway
    - Making Your Applications Highly Available with Route 53
    - Isolating and Securing Your Instances and Resources with VPC
    - Using CloudFront to Distribute Your Content around the Globe
  - Continuous Integration & Delivery
    - Creating a Reliable Continuous Delivery Process with CodeBuild & CodePipeline

- Observability
  - Observing All Your AWS Services with CloudWatch
- Define & Deploy Your Cloud Infrastructure with Infrastructure-As-Code
  - CloudFormation Is the Underlying Service for Provisioning Your Infrastructure
  - Using Your Favorite Programming Language with CDK to Build Cloud Apps
  - Leveraging the Serverless Framework to Build Lambda-Powered Apps in Minutes
- Credits & Acknowledgements
- About the Authors

# Introduction

With this book, we hope to get you a deeper understanding of AWS, beyond just passing fundamental certifications. It covers a wide range of services, including EC2, S3, RDS, DynamoDB, Lambda, and many more, and provides practical examples and implicit use cases for each one. The book is designed to be a hands-on resource, with step-by-step instructions and detailed explanations to help you understand how to use AWS in real-world scenarios.

Whether you're a developer, system administrator, or even an engineering manager, this book will provide you with the fundamental knowledge you need to successfully build and deploy applications on AWS.

# About the Scope of This Book

This book is all about the fundamentals of AWS. The goal is to get you started on how to use AWS in the real world.

First, we'll show you how to create your first AWS Account, how to set up your root users, and how to make sure you will receive billing alerts.

The next part covers the most important AWS services. AWS consists of more than 255 services. We picked out the services that you will use in almost any cloud application. Example services are Elastic Container Service (ECS), Lambda, Simple Queue Service (SQS), Simple Notification Service (SNS), EventBridge, and many more. We dive deep into these services and give you recommendations for the best configuration options, jump into use cases and provide you a list of tips and tricks and things to remember.

In the last part, we give you an introduction to Infrastructure as Code. We want you to understand the differences between various frameworks. For that, we've created a brief introduction and history lesson on IaC. CloudFormation, Serverless, and the Cloud Development Kit (CDK) are three frameworks that are used a lot. We show you examples of how to create infrastructure with all three of them.

# Why Did We Bother to Write This?

Why did we bother writing another book about AWS?

Working with AWS both felt like using superpowers. On the one side, you can build applications that are globally available without caring about infrastructure. On the other side having the skill of using AWS is globally in demand.

We want to pass on the knowledge of both points as well. By knowing how to build on AWS you can boost your career. But you can also finally work on your SaaS idea.

We're both lucky in the way we learned AWS. During our studies, we worked in companies where experienced employees could teach us the basics directly but we've also had the freedom to learn ourselves and make mistakes. Through the years, we could harden our skills and gain a lot of insights into different areas. We've seen how AWS progressed, but the fundamentals still remained the same.

We saw colleagues and friends struggling a lot with learning the core services of AWS service and its underlying principles and how to apply them in the day-to-day work. The typical learning path is to get started with certifications. While this is not inherently a bad way it is often not enough. Certificates can be really hard to master. But they often don't bring enough value if you don't put the learnings into immediate practice. People are often still overwhelmed by which services they should use in which situation and how to configure them accordingly. This is the main motivation of this book.

Learning AWS doesn't need to be hard. It is important to focus on the basics and to understand them well. Once this is done all new services or features can be understood really well.

Each cloud application consists of the same set of services and principles.

We both never thought about writing a book. But during our time working, and especially once we started to create content we saw the need. There were so many questions and misconceptions that we wanted to create a resource on how to learn AWS for the real world.

# Who Is This Book For

This book is for everybody who wants to learn about the fundamentals of AWS. We cover the core building blocks of AWS and Infrastructure as Code.

We will show you example use cases and configuration options for each service. With that, you are ready to understand how to apply it in the real world.

Programming experience doesn't matter for this book. While infrastructure is code nowadays you don't need to know any specific programming language. Programming is a tool you will use to build on the cloud, but it is not a prerequisite as it can be acquired along the path.

This book is also for everybody who did some certifications like the Cloud Practitioner or Solutions Architect Associate but is still overwhelmed with how to apply the learnings in real-world projects.

If you're an entrepreneur who wants to start building on AWS this is also a great resource for you on how to get started.

Or if you are the technical manager of a team and somehow lost contact with AWS and its configuration options you can brush up your knowledge fast and reduce the knowledge gap in your engineering team.

If you are working with AWS for quite some time but still are not sure about some configuration basics (like when to use long & when to use short polling), this is also for you.

# Who Is This Book Not For

Honesty is important. We only want people to buy this book if they can profit immensely from reading it.

Firstly, if you're really proficient with AWS and you've worked in the area for many years, likely this book is not for you. We don't require previous knowledge about basically anything, and that's also where we start. By exploring every core service as deeply as possible, we want to give aspiring cloud engineers a fundamental tool to start building their own applications or simply to get hired in this area.

This book is also not for people that don't want to or simply won't directly or indirectly work with AWS. If your future or current focus is Azure or Google Cloud Platform, there's more value in purchasing another book. It can make sense to understand how AWS is handling things, but if you aim to work with another cloud provider, learning their specifics is key.

Furthermore, this book doesn't focus on passing certifications. You'll learn the principles that are required to know how to build applications from scratch and how to apply that knowledge, but passing certifications often require very deep knowledge that goes way beyond. This book is a good tool to set yourself up for a good baseline for fundamental certifications like the Cloud Practitioner or the Solutions Architect Associate. But if you focus on passing certifications, doing practice exams, or courses that strictly focus on exam questions, will do a much better job.

# Getting Started

## Creating Your Own AWS Account

Reading about the fundamentals of AWS is important, there's no question about it. But learning anything in the area of engineering always requires practical hands-on. Because of that, building and exploring your own AWS account is a must and it goes hand-in-hand with reading.

Creating an account requires you to have a credit or debit card in your name that will be charged a small amount of \$1 or less by AWS. It won't actually be deducted but is only used for validation purposes. After some time, the charge will disappear.



### Sign up for AWS

**Explore Free Tier products with a new AWS account.**

To learn more, visit [aws.amazon.com/free](https://aws.amazon.com/free).



**Root user email address**

Used for account recovery and some administrative functions

**AWS account name**

Choose a name for your account. You can change this name in your account settings after you sign up.

**Verify email address**

OR

**Sign in to an existing AWS account**

Besides the payment information, you only need an email address and phone number. The latter one will be used to verify your identity as you'll receive a text message or voice call with a one-time password. In the last step, you'll be asked about a support plan - pick the free option as you don't need enhanced support for learning AWS.

And that's basically it. Your account is created and you've received a unique 12-digit account identifier (which is not considered a secret). You're ready to log into the console and get started with hands-on.

Don't feel scared or overwhelmed after seeing the dashboard or the service search for the first time. You'll quickly get used to AWS core concepts and to how interfaces are structured. Navigating through services and configurations will get much easier over time and you'll now where to look to find what you're searching for.

# Account Security Key Concepts and Best Practices

With great power, there comes great responsibility. Having an AWS account is a great power, as you're able to launch unimaginable computing powers within seconds.

As you're solely responsible for your account, you should as the first major task in your cloud journey, how to protect it properly.

This mainly consists of two steps:

1. enabling multi-factor authentication
2. understanding and applying the identity and access management concepts of AWS

We beg you to not skip this chapter, even if the desire to start jumping into the services is great. At AWS, there's no limit to what you can do and therefore also no limit to what you can get charged for.

## Enabling Multi-Factor Authentication to Have Another Layer of Security

Credentials can get lost or leaked easily, that's why you should rely on a physical second factor for authenticating to AWS with both your root user and your daily IAM users.

Enabling multi-factor authentication (MFA) is very straight forward and AWS supports both virtual MFA devices (e.g. the Google Authenticator) and hardware MFA devices like the YubiKey.

Click on the top right on your user and select **Security Credentials** for being redirected to the security dashboard of your account.

**Multi-factor authentication (MFA) (1)**

Use MFA to increase the security of your AWS environment. Signing in with MFA requires an authentication code from an MFA device. Each user can have a maximum of 8 MFA devices assigned. [Learn more](#)

[Remove](#) [Resync](#) [Assign MFA device](#)

Device type	Identifier	Created on
Virtual	arn:aws:iam::157088858309:mfa/root-account-mfa-device	204 days ago

You'll see options to change your password, create access keys for command line access and

assign an MFA device. By clicking on the latter one, you'll be prompted to choose your preferred method.

Select MFA device [Info](#)

Select an MFA device to use, in addition to your username and password, whenever you need to authenticate.

 **Authenticator app**  
Authenticate using a code generated by an app installed on your mobile device or computer.

 **Security Key**  
Authenticate using a code generated by touching a YubiKey or other supported FIDO security key.

 **Hardware TOTP token**  
Authenticate using a code displayed on a hardware Time-based one-time password (TOTP) token.

After entering two generated, consecutive one-time passwords, your MFA will be active and required at the next login.

### About Saving MFA Methods at Your Password Store Application

Yes, this is very convenient, but completely undercuts the purpose of having a second factor at all, as it's supposed to require another physical or virtual device. As the compromise of an AWS account can have an immersive impact on your financial resources, we recommend really keeping your second factor a real second factor.

### Staying Away from Using Your Root Accounts Credentials for Your Daily Business to Reduce Risks

Your root user has full control over your account, your billing, and every resource you'll ever create. As a best practice, it's recommended to lock it away immediately and switch to Identity and Access Management (IAM) users. Don't ever use the root credentials for your daily work, neither in the AWS console nor with infrastructure as code or the AWS command line interface.

Also, citing the AWS documentation about root users:

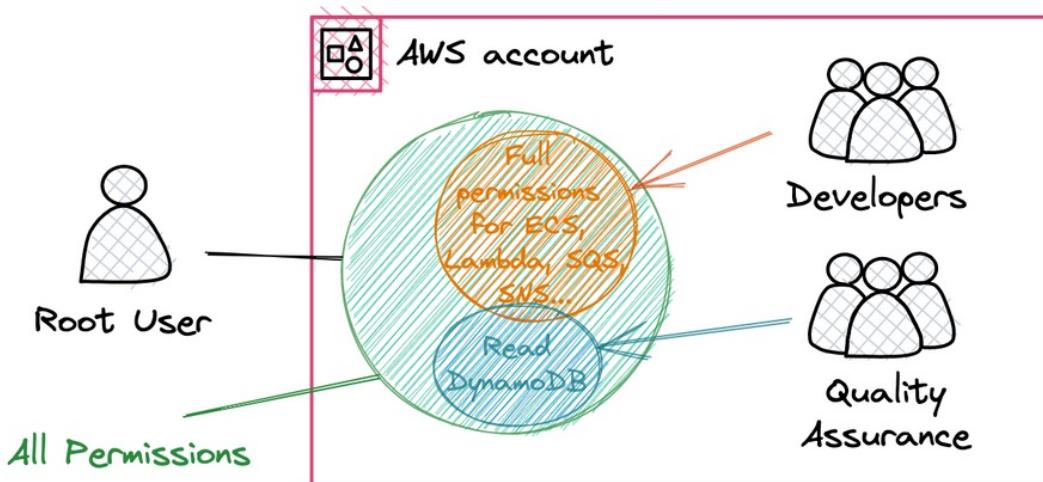
*We strongly recommend that you do not use the root user for your everyday tasks,*

*even the administrative ones. Instead, adhere to the best practice of using the root user only to create your first IAM user.*

And that's exactly what we'll do in the next step - together with exploring the fundamentals of AWS IAM. We'll go into IAM more deeply in its own chapter, but we'll quickly revise the identity types beforehand so we can lock away our root user as soon as possible.

## **AWS IAM Puts You in Full Control of the Security of Every Aspect of Your Account by Offering Users, Roles, and Groups**

AWS Identity and Access Management, or short AWS IAM, is the glue that connects everything in your AWS accounts. It's also managing how you can access your account and what permissions are available.



Let's go through each of the basic concepts.

### **Your Account Is a Closed Bucket of Resources**

An account contains everything: all the resources you've created, all the configurations you've applied to those resources, and everything you've built. A good analogy is to think of it as a closed bucket of cloud resources.

### **The Root User Is the Owner of Your Account**

The root user is the owner of your account and has all privileges to manage, modify or delete all resources or even the account itself. Some configurations can only be set by the root user like changing the account name, updating of payment information, or assigning the account to an organization.

### **IAM Users for Tailor-Made Permissions to Fulfill Daily Tasks**

IAM users on the other hand can be created individually for different purposes. They can be either used by a person or by an application and there can be several for a single account. Contrary to the root user, IAM users do not have any permissions in the beginning. You need to assign them explicitly. Those permissions can be as restrictive as possible, as long as they can fulfill their desired tasks.

### **Groups to Easily Manage Permissions for Small or Large a Set of Users**

You've probably already imagined: even with code-based configurations for your users, managing permissions can be tedious based on the number of users for your account. That's why AWS implemented groups.

A group can have its own permissions assignments. Users in turn can be assigned to those groups to inherit the group's permissions. With this construct, maintaining allow and deny rules for a large number of users becomes easier. Each group is an oversee-able security cluster that can be better maintained than individual rights one by one.

# Avoiding Cost Surprises

We've already touched it before, but due to its importance, we'll talk about it again: there's no enforceable spending limit for your account.

You may have heard horror stories from small start-ups about exploding costs due to recursive Lambda functions, high NAT Gateway data traffic, or exploding logs that result in terabytes of ingested logs at CloudWatch.

But you're not out of options to protect yourself from costs that get out of control:

1. AWS provides you with its Free Tier that allows exploring services without paying much or anything.
2. There are pricing calculators that enable you to estimate costs properly.
3. You can set up alarms for cost estimations that will cross defined thresholds.

As with the key account security principals chapter before, this is one you should not skip.

## **Experimenting without Any Costs by Making Use of the AWS Free Tier**

Almost all core services either enjoy a certain free usage for either the first 12 months or every month. Famous free-tier offerings include:

- Lambda - 400,000 GB-seconds of execution. For small-sized Lambda functions, this results in several weeks of a non-stop running function. Don't worry here: we'll explore how the weird metric "GB-seconds" is calculated in the next paragraph.
- API Gateway - 1 million HTTP requests, which is quite a lot for starters and enables you to run and expose many small-scale applications for free.
- DynamoDB - 25 GB of storage and 25 Read & Write Capacity Units. A must for Serverless and Lambda fans which covers a significant concurrency for your application's database access.
- S3 - 5 GB Storage, 20,000 GET, and 2,000 PUT requests. As S3 is part of nearly every application, this is another generous Free Tier that allows you to go deep with S3.
- EC2 and RDS - both with 750 hours of running certain micro instances. If you do the math, this allows you to run such a micro instance for the whole month without getting

charged at all.

Certainly, this is just a small fraction of the offerings, but the ones that do matter a lot.

AWS also regularly increases the Free Tier limits for many services so it's always worth checking the current state of the offerings.

Especially for services that are managed and Serverless (not directly using containers or virtual machines, but only paying for your actual use without any upfront costs), you're able to do a lot without spending a dime in the first place. But more about this is in the later parts of this introductory chapter.

## Exploring Your Cost Structure with the Billing Dashboard

Before starting hands-on with any service, it's a must to get familiar with its pricing structure. Are there any upfront costs, are you charged per hour, per usage, and/or per induced traffic? It's critical to have a rough estimate of what you'll pay. There's no need to get into the deepest levels and calculate costs for every day, week, and month.

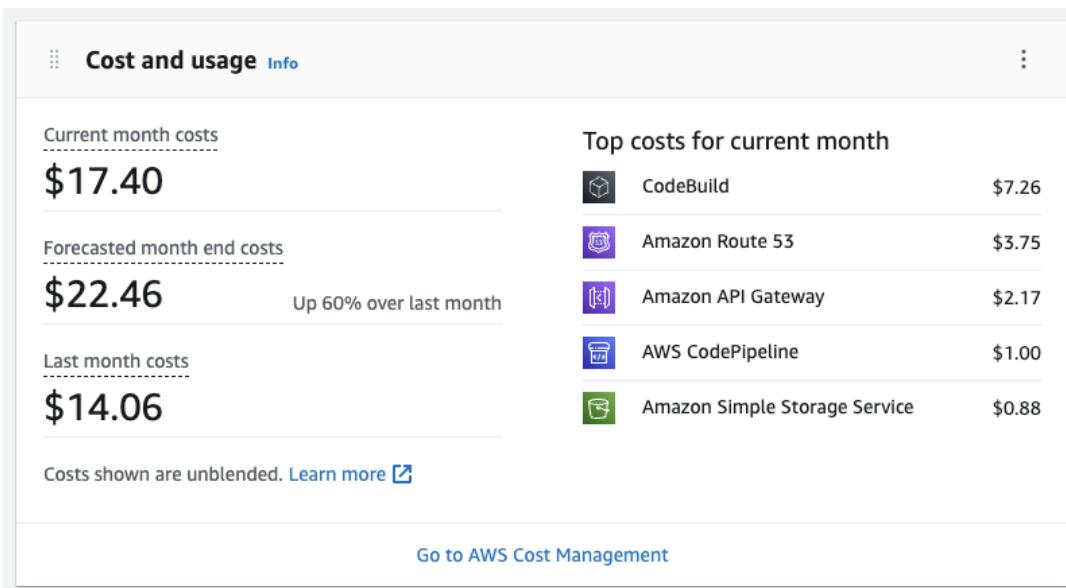
But as with other areas of life: if you're not aware of your spending you're wandering in the dark and you can easily get an unwanted surprise at the end of the month.

Let's have a look at popular services you'll learn about in the book and their pricing structure:

- **CloudWatch** - the costs per used GB of storage is comparably low but the ingestion of logs can be pricy. If you're logging full-blown and extensive JSONs, be sure to not escalate with debugging or trace levels as ingested gigabytes of logs can quickly add up and contribute significantly to your bill.
- **Lambda** - you'll see that you're charged per GB-seconds, which is not an intuitive declaration at all. This means that you'll pay for your Lambda function per executed second, but dependent on your Lambda function's provisioned memory. In other words: with the Free Tier of 400,000 GB-seconds you can execute a Lambda function with 1 GB of memory for 400,000 seconds, or roughly 6666 minutes, or 111 hours, or 4.6 days. Equally, if you're running a function with 10 GB memory, you'll be charged after only about 11 hours of execution. Even though it's expected at first, higher memory settings don't have to increase your bill. We'll explore this deeply in the Lambda chapter.
- **DynamoDB** - you'll pay for used storage and reads and write operations. In addition, it depends on whether you want to have on-demand or provisioned capacity. On-demand means, you'll only be charged for actual usage - so each read and write - while provisioned

will introduce fixed costs as long as your table exists, but include all read and write operations. Both modes have certain advantages over the others depending on your usage patterns. As with all core services: we'll talk about pricing in the corresponding chapter.

As seen, it's often not easy to make a rough guess about costs as the pricing structures differ from service to service. Practical advice is to regularly check your billing dashboard - weekly or bi-weekly - to get a better feeling of how costs evolve and which services significantly contribute to your bill.



The AWS Dashboard also offers a widget that is active by default and shows the cost and usage of your AWS account, drilled down by service. So having a look at the current month's cost is easy as you'll always see this after logging in.

## Getting a Forecast about Your Costs and Get Alerted on Exceeding Thresholds

As mentioned earlier: you can't set any spending limitations per service or even per account. You'll always pay for introduced costs. We've encountered a lot of people that strictly avoided learning anything about the cloud as they are scared about unexpected costs at the end of the month.

The left panel shows the 'Set alert threshold' configuration for a budget. It includes fields for 'Threshold' (80 % of budgeted amount) and 'Trigger' (Forecasted). A summary states: "When your forecasted cost is greater than 80.00% (\$40.00) of your budgeted amount (\$50.00), the alert threshold will be exceeded." Below this, 'Notification preferences - Optional' allow selecting email recipients (schmidt.tobias@outlook.com) and choosing an Amazon SNS ARN (arn:aws:sns:us-east-1:924441585974:slack-alerts-chatbot). A note about SNS pricing is shown.

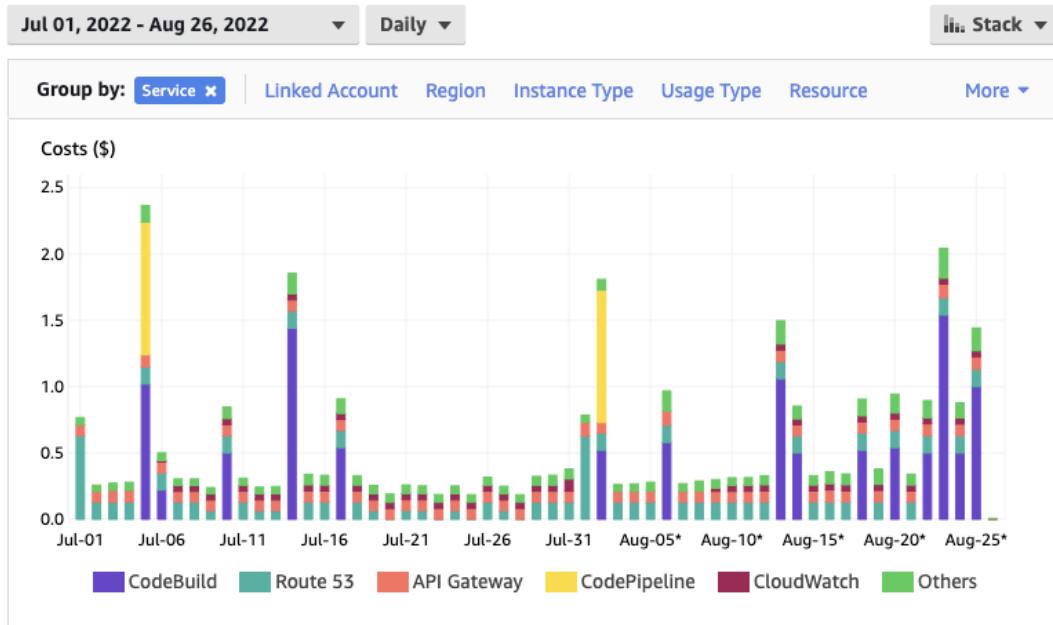
The right panel shows the 'Budget preview' dashboard. It displays a bar chart titled 'Cost Data' for the period Aug 2021 - Aug 2022 (MTD) | Unblended costs. The Y-axis ranges from 0 to 50. The X-axis shows months from Aug 2021 to Jun 2022. Actual costs (blue bars) are compared against the budget (red dashed line at 40) and the forecasted cost (green bars). An alert for 'Forecasted cost > 80%' is listed under 'Alerts'.

What you can do: Use AWS Budgets to create alerts if thresholds for costs are exceeded. AWS also tries to calculate the future costs of your account via estimates based on current and previous usages, which can also be used for budget alerts. The alerts are not in real-time as forecasts are only updated in time intervals, but it will nevertheless inform you via email if your predefined spending limits are or will be breached.

Budgets (1) <a href="#">Info</a>						
<input type="text"/> Find a budget		<a href="#">Show all budgets</a>		<a href="#">Download CSV</a> <span>Actions</span> <span>Create budget</span>		
Name	Thresholds	Budget	Amount used	Forecasted amount	Current vs. budgeted	Forecasted vs. budgeted
Default	<span>OK</span>	\$50.00	\$17.40	\$22.46	34.80%	44.92%

## Further Drilling down Costs with the Cost Explorer and Cost Allocation Tags

AWS Cost Explorer allows you to gain very deep insights into your cost structure. It allows you to drill down features by service, region, resource, or even instance type. With one glance, you're able to determine which services contribute mostly to your bill and where you maybe could improve at cost optimization.



Another major feature offered by Cost Explorer that brings even more flexibility is Cost Allocation Tags. These tags can be defined to measure the costs per component level or any granularity you want as you decide how to structure tags and where they will be applied. It's a perfect tool to get detailed insights into your cost structure.

You'll find out which parts - regardless if it's a component, sub-component, certain cluster of services, or anything else - of your infrastructure heavily contribute to your costs.

## Sleeping Better by Restricting IAM Permissions To Launch Expensive Resources

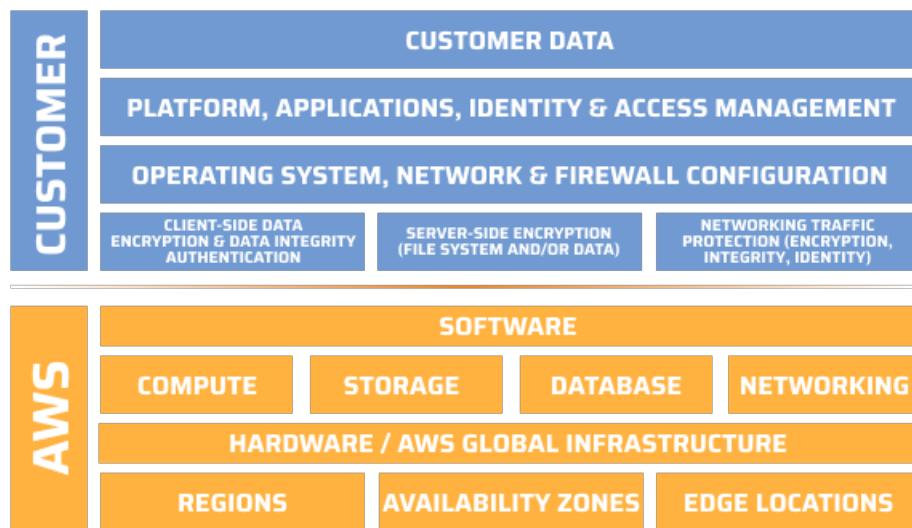
If you're starting out with the cloud and you want to focus on certain core services like Lambda and other related services like SQS, SNS, or DynamoDB, you can take advantage of IAM and adapt your user's permissions to only allow actions on those services.

This will effectively deny very costly actions like launching expensive EC2 instances or buying reserved capacity. Even though those are explicit steps you'd have to take to actually spike your AWS bill, this will calm your mind as your subconsciousness knows it's just not easily possible to invoke these actions.

# Understanding the Shared Responsibility Model

We're in a paradigm shift. We're transitioning from a model where everything is your liability - including every infrastructure - to moving more and more responsibilities to the platform provider. This means: application developers can focus more on building the actual product instead of spending a significant amount of time creating and operating infrastructure.

AWS separates the responsibility into two categories "Security **of** the Cloud" and "Security **in** the Cloud". The first one, in the hands of AWS, means that it protects the infrastructure that runs all of its services. The second one is about the correct and proper usage of services, focusing on a secure configuration. This heavily depends on the service and its abstraction level. It requires much less effort to use and secure a fully-managed service like S3 than operating virtual machines running on EC2.



Another interesting thing to note: the term hardware was created as hardware was hard to change. Switching software to new servers, or scaling clusters vertically or horizontally was a non-trivial task and required a lot of effort in the past. The software was seen as much more easier to change, as it is just code running on some platform that can be quickly replaced or adapted.

This world has changed: due to AWS, Azure, and GCP, getting infrastructure at almost every place in the world is mostly just one click (or line of code) away and can be bootstrapped in a matter of seconds. On the other hand, software eats the world and takes over more and more difficult tasks and processes, and is integrated into almost every aspect of our life. Large-scale

applications with architectures that grew over many years can become very hard to change, as side effects are often just hard to grasp or understand. Also, many services are considered not just critical but absolutely necessary to run without much or any interruptions which in turn leads to even more burdens of adapting or extending software processes.

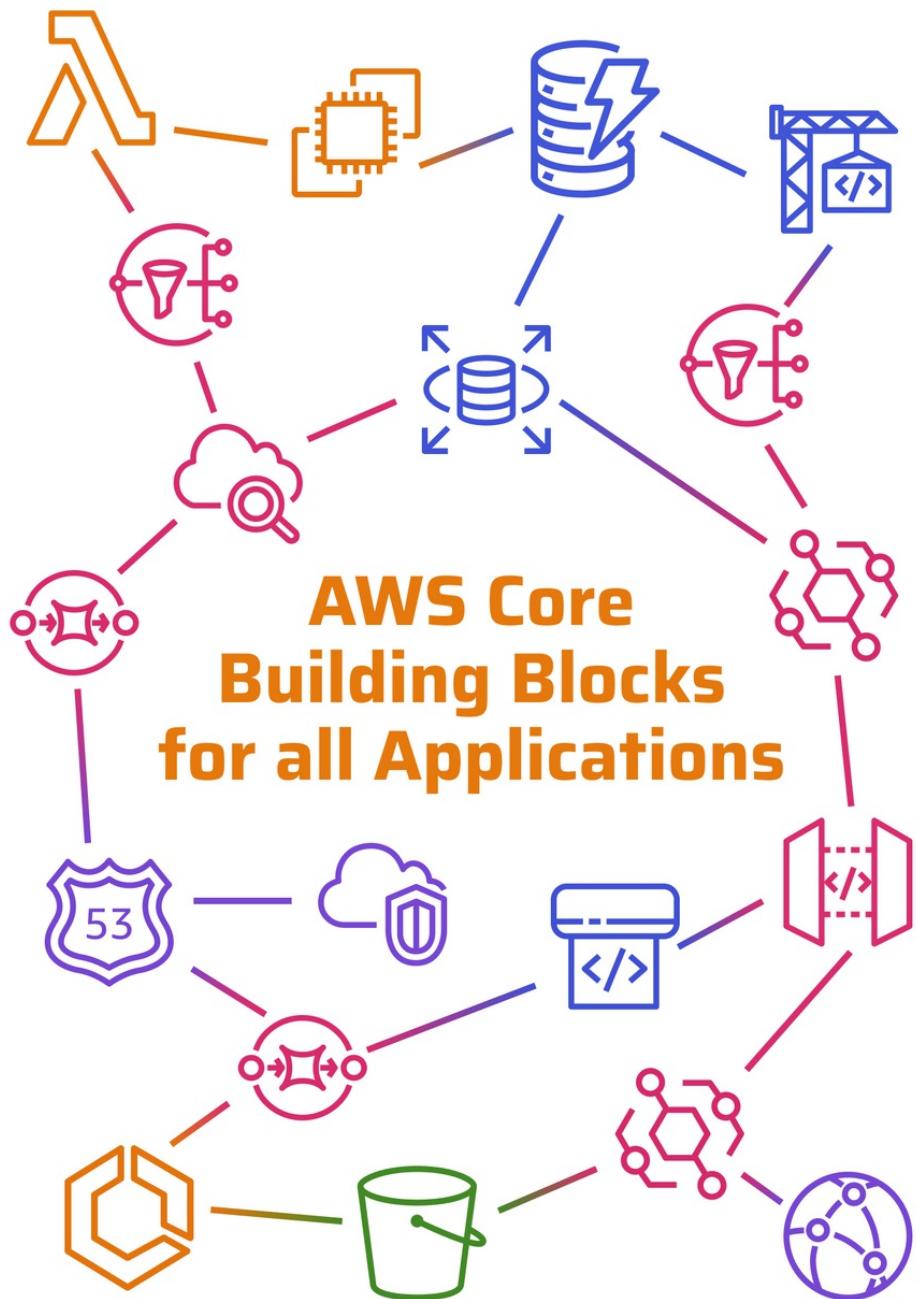
# About going Serverless and Cloud-Native

We strongly believe the future is cloud-native and Serverless. But what does this actually mean?

The traditional software development process often splits into two parts: the actual application development first and the hosting of the solution as a second part. Cloud-native on the side means that application development and infrastructure selection, usage, and integration go hand in hand. This means leveraging platform technologies that will directly benefit the application processes themselves.

The first step that could be observed while public clouds have started to take over was that companies have given over responsibility for infrastructure by running containers via cloud providers. With evolving and new services like Lambda - which are even more abstracted so that there's even not a container anymore to manage on the customer side - this has taken even more steps forward. Application developers gain more time to focus on features and actual business process implementation instead of spending time on operations.

As infrastructure can be created and destroyed within a blink of an eye and solely with code it's also possible to replicate whole application ecosystems without spending much or any effort. This results in enabling developers to develop not locally but also remotely with a fully functioning duplication of a production system without having to worry about affecting customers. With the rise of Serverless technologies and pay-what-you-use models, you also do not have to think much about exaggerating costs for development stages as they do not introduce costs if they are not actively used.



# AWS Core Building Blocks for all Applications

AWS offers a wide range of services and it's increasing from year to year. Nevertheless, there are core services that are important for the majority of applications. In this book, we want to focus on those core building blocks.

Before we jump into each service's details, let's quickly go over each of them. Each of them is part of a specific area or cluster due to its features and responsibility.

AWS Identity and Access Management (**IAM**) is an overarching, central service that is part of every application you'll ever build and the glue that keeps everything together. It can be seen as a part of every service.

For running applications or any workload:

- **EC2:** Amazon Elastic Compute Cloud provides resizable compute capacity in the cloud. Instead of running your own servers in a data center, you get virtualized machines. It's only the first abstraction layer but offers the greatest flexibility.
- **ECS:** Amazon Elastic Container Service is a fully managed service for running Docker containers. It requires even fewer operations than EC2 and simplifies the process of deploying and scaling applications.
- **Lambda:** AWS Lambda allows you to run code without provisioning or managing servers. It's the largest abstraction, as you're not responsible for any infrastructure. Everything is managed by AWS in the background. You'll also only pay for the time when your code actually executes.

For storing data of any kind:

- **RDS:** Amazon Relational Database Service is a managed service for running relational databases. It's one of the most mature database services out there.
- **DynamoDB:** Amazon DynamoDB is a fast, fully managed NoSQL database service. It's one of AWS's flagship services that scale on-demand to almost any requirement.
- **S3:** Amazon Simple Storage Service is an object storage service that offers industry-leading scalability, data availability, security, and performance. It's a part of almost every application and is used for storing and retrieving any amount of data at any time.

Connecting components and building resilient, event-driven architectures:

- **SQS**: Amazon Simple Queue Service is a fully managed message queuing service. It enables you to decouple and scale microservices, distributed systems, and serverless applications.
- **SNS**: Amazon Simple Notification Service is a fully managed pub/sub messaging service. It allows you to send messages to a large number of subscribers.
- **EventBridge**: AWS EventBridge is a Serverless event bus that makes it easy to connect applications together. It allows applications to share data and events without creating dependencies.

Securely running and exposing your applications to the internet:

- **API Gateway**: Amazon API Gateway is a fully managed service that makes it easy to create, publish, maintain, monitor, and secure APIs. It's not a simple HTTP mediator but offers advanced features like request validation and data transformations without writing boilerplate code.
- **Route 53**: Amazon Route 53 is a highly available and scalable cloud Domain Name System web service. It enables you to build multi-region architectures with low latencies and automatic failovers in case of incidents.
- **VPC**: Amazon Virtual Private Cloud enables the provisioning of a logically-isolated section of the AWS Cloud. It enables you to separate and secure resources on the network level.

Building and delivering your applications in a reliable way:

- **CodeBuild & CodePipeline**: CodeBuild and CodePipeline are continuous delivery services for creating reliable, reproducible build automations and deployments.

Observing your infrastructure and applications:

- **CloudWatch**: Amazon CloudWatch is a monitoring service for AWS resources and the applications you run on AWS. It's integrated with almost any service and automatically collects metrics about usage and performance.

We hope you're already excited to jump into each of those great services.



## AWS IAM

# AWS IAM for Controlling Access to Your Account and Its Resources

## Introduction

AWS Identity and Access Management (IAM) enables you to manage identities and access AWS services and resources securely. Instead of maintaining dozens of credentials, you'll work with roles and policies that allow fine-grained permissions that can not only be assigned to users but also to resources.

AWS IAM is a fundamental part of AWS security, as it ensures that only authorized users have access to your resources. Gaining deep knowledge, in the beginning, ensures that you don't create environments with crucial security flaws that are hard to revise in the future.

IAM is one of the services that is straightforward to get started with, but hard to master, as the power lies in the wide range of features.

## Amazon Resource Identifiers: How Resources Are Identified in a Uniquely Manner

First things first: for making AWS IAM possible, there needs to be a concept of how resources are uniquely identified. This is done via **Amazon Resource Identifiers** or short ARNs. Each of those ARN strings consists of several parts, including the resource type, AWS region, and the account ID of the resource.

ARNs are used not only in IAM but in various contexts, including AWS CloudFormation templates, and AWS service APIs. They allow you to specify and authorize access to resources in a secure and standardized way.

Let's have a look at different examples:

- An Amazon S3 bucket: `arn:aws:s3:::my-bucket`
- An Amazon EC2 instance: `arn:aws:ec2:eu-west-1:123456789012:instance/i-01234567890abcdef`
- An Amazon RDS database: `arn:aws:rds:eu-west-1:123456789012:db:mydatabase`

- An Amazon DynamoDB table: `arn:aws:dynamodb:eu-west-1:123456789012:table/mytable`

The resource type is the first part (e.g. `s3`, `ec2`, `rds`, `dynamodb`). After the general prefix `arn:aws` which just indicates that it's an ARN for "global" AWS (there's also AWS China or AWS CN, which is strictly separated and identified with the prefix `arn:aws-cn` ).

The region is specified by the second part (e.g. `us-east-1`), and the account ID is specified by the third part (e.g. `123456789012`). Note that some services like S3 are not bound to a single region even though their resources may live in one region. In this case, the region identifier is left blank.

The specific resource is identified by the remaining parts of the ARN, which can include the resource name or identifier. For example, the ARN for an Amazon DynamoDB table includes the table name (`mytable`) after the `table/` prefix. This allows you to uniquely identify the table and authorize access to it using IAM policies.

## Users, Roles, and Groups Are the Three Important Identity Concepts

IAM comes with the concept of identities. An identity represents a user and provides access to resources within your AWS account. They can also be assigned to groups to manage rights for users more easily. Each identity can be associated with one or more policies that state which permissions for which resource are granted to that identity. Policies can also be attached to roles that don't represent identities or users but can be assumed by them.

Let's dive into the different types more deeply in the following paragraphs: users, groups, and roles.

### User: A Person or Service That Interacts with One or Several AWS Accounts

Users are identities that can interact with AWS and its APIs. They consist of a name and credentials and their AWS access type(s). It's recommendable to use speaking, "friendly" names for your users.

The access type can be either **programmatic** (via access keys that can be used to make calls to the AWS API), via the AWS **Management Console** (via password), or **both**.

## Add user

1 2 3 4 5

### Set user details

You can add multiple users at once with the same access type and permissions. [Learn more](#)

User name\*

[+ Add another user](#)

### Select AWS access type

Select how these users will primarily access AWS. If you choose only programmatic access, it does NOT prevent users from accessing the console using an assumed role. Access keys and autogenerated passwords are provided in the last step. [Learn more](#)

- Select AWS credential type\*  **Access key - Programmatic access**  
Enables an **access key ID** and **secret access key** for the AWS API, CLI, SDK, and other development tools.
- Password - AWS Management Console access**  
Enables a **password** that allows users to sign-in to the AWS Management Console.

## Securing Your Account's Root User

We talked about this in the beginning, but due to its importance, we think it's a good idea to revise it quickly one more time.

Each AWS account comes with a single pre-defined identity that has complete access to all resources and services. It's called the root user and you can sign in with the email address you've used for the registration. This root user is not the same as an IAM user with administrator permissions, as some actions are only possible with the root user, including management of payment methods, assigning permissions to access billing and cost management, or completely closing your account and with that deleting all of its resources. AWS strongly recommends not using this user for any daily tasks due to its critical permissions.

Therefore your first task should be creating a dedicated IAM user and securing your root's credentials. This means:

1. enabling multi-factor authentication.
2. storing your credentials in a secure place.
3. deleting access keys that could be used to access the AWS API.

If you already created your first AWS account and you didn't take those steps until now, jump back to the security introduction chapter and catch up on this.

## Credentials, Access Keys, and the AWS Security Token Service API

AWS can be used in different ways, including interactive access via the AWS management console and your browser, or programmatic access via the AWS API, depending on the user

credentials. The most prominent ones are console passwords and access keys.

- **Console passwords** - they allow the user to sign into an interactive session at the AWS management console. Users will be prompted for the unique 12-digit account identifier, their IAM user name, and their password. The account ID is required as IAM user names don't have to be unique over all AWS accounts like S3 bucket names, but only per account.
- **Access keys** - they are for programmatic access to AWS via its API. Generally speaking, it's good to remember that everything at AWS is an API. If you're using the AWS management console, the API calls are abstracted into a clickable user interface and the loaded scripts in your browser translate your action into calls to the AWS API. If using access keys, you can directly submit calls to the AWS API for creating, updating, deleting, or listing resources. There are multiple tools to make the use of the AWS API easier:  
**PowerShell** for Windows or **aws-shell** for Linux or macOS.

With an enabled MFA for your user, the API access via the Access Key ID and Secret Access Key also changed. Now, you need to request temporary credentials via your keys and the one-time security token that's generated by your MFA application. This is somewhat cumbersome if you want to do this manually, so tools like AWSSsu.me or Leapp abstract this nicely.

### **Group: A Collection of Users to Easily Manage Several Users**

With increasing users of an AWS account, maintaining individual permissions can become tedious. It's recommendable to cluster users into dedicated permissions groups based on their requirements for their daily work.

Let's look at a simple example, a team that consists of **four** different roles: **administrators**, **developers**, **quality assurance**, and **business analysts**.

Each of the roles can be put into a group for which fitting permissions are assigned.

- **administrators**: having enhanced IAM permissions to create and manage users while the company or team grows, shrinks, or adapts in its structure.
- **developers**: permissions to create AWS resources to build and deploy applications on AWS.
- **quality assurance**: permissions to access delivery pipelines, databases, and reports.
- **business analysts**: permissions to access production data and usage reports to gather detailed insights about customer behavior.

Even if not noted in this example, users can also be part of multiple groups which will then gain the commutated permissions of all of the assigned groups.

### **Role: An Identity with a Permission Set That Can Be Assumed**

Roles have some similarities to users: it's an identity that is associated with permissions to determine which actions can be taken at AWS. However, roles are not associated with a single person but can be assumed by anyone or anything who needs it.

This includes AWS service principals that are used for example for Lambda functions or container agents at ECS.

### **Issuing Temporary Credentials to Enforce Time-Limited Access**

With the help of AWS Security Token Service (AWS STS), you can provide short-term security credentials that are not stored with users but are provided on demand when requested. They can be configured to last anywhere between just a few minutes to several hours.

Before the temporary credentials expire, users can request new temporary credentials if they still got the permission to do so.

This is a great feature to offer short-lived credentials to external services which are not in your control. For external security breaches, access can be revoked quickly and easily.

## **Controlling Access to Your Resources via Policies**

Every request to AWS goes through an enforcement check to determine if the requesting principal is authenticated and authorized for the targeted action. The decision is based on the assigned policies either directly to the IAM user or the role that is currently assumed.

### **Policies for Granting Permissions to Access Your Resources**

A policy is an object in AWS that determines `allow` or `deny` actions for services and resources. They are mostly stored as structured JSON documents and come in different types.

#### **Statements to Determine the Scope of a Permission**

Each policy comes with one or several statements that define **which actions** are granted to **which resource** under **what conditions**.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllowGetObject",
            "Action": "s3:GetObject",
            "Effect": "Allow",
            "Resource": "arn:aws:s3:::mydata/*"
        }
    ]
}
```

**Example A:** This policy allows the user or group that the policy is attached to retrieve objects from the **mydata** bucket. The **Resource** element specifies the Amazon Resource Name (ARN) of the bucket, and the **/\*** at the end of the ARN allows access to all objects in the bucket.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "DenyIPRange",
            "Action": "*",
            "Effect": "Deny",
            "Resource": "*",
            "Condition": {
                "NotIpAddress": {
                    "aws:SourceIp": ["192.0.2.0/24", "203.0.113.0/24"]
                }
            }
        }
    ]
}
```

**Example B:** This policy denies all actions on all resources for requests that originate from the specified IP ranges. The **NotIpAddress** condition specifies the IP ranges to be denied, and the **aws:SourceIp** value specifies the source IP address of the request.

Let's have a detailed look at each of the elements of our two example policies:

- **Version:** specifying the version of the policy language you want to use. The latest one is `2012-10-17` and you shouldn't use a previous version.
- **Statement:** a container holding one or several items that define the permissions.
- **Effect:** stating whether actions are granted (**allow**) or denied (**deny**). A deny statement always overwrites allow statements.
- **Principal:** missing in our example, but for example used with resource-based policies to determine the account, user, or role for which the statement applies.
- **Action:** a list of API actions for the target service that is either allowed or denied.
- **Resource:** the resources for which the statement should allow or deny the given actions. This is not required for resource-based policies, as it's implicitly applied to the resource to which the policy is attached.
- **Condition:** specifying under which circumstances the statement should be applied. This is optional but can be used to further drill-down permissions.

Policies come with size restrictions and it's a best practice to split your policies by the resources you're granting access to.

### **Identity versus Resource-Based Policies**

The core types you'll stumble upon guaranteed in your daily work are **identity-based** and **resource-based** policies.

On the one hand, Identity-based policies grant permissions to identities (users, groups, or roles). AWS distinguishes between two sub-types:

- Managed Policies: policies that can exist on their own (standalone) and can be attached to multiple identities within your AWS account. Managed policies in turn also divide into two further categories:
  - AWS-managed policies: policies that are created and managed by AWS.
  - Customer-managed policies: policies that you create based on your requirements.
- Inline Policies: policies with a strict one-to-one relationship to an identity.

Resource-based policies on the other hand are not attached to identities but to AWS resources, e.g. an S3 bucket. These policies grant specified principals permissions on that resources. Resource-based policies are implicitly inline policies as they can't live without their corresponding resource.

## AWS-Managed Policies Provided by AWS for Quickly Starting

AWS offers a large set of managed policies that you can use and come with your account by default. You can't modify or delete them, but make use of them. This is especially useful when starting with AWS to quickly somewhat fine-grained access to services and resources without fiddling around with IAM too much.

## Custom Policies for Fine-Grained Permissions for Every Use Case

When you create your own tailor-made policies, you can only assign necessary permissions for an identity. In contrary to AWS-managed policies, you are in full control of any granted or denied permission and you don't need to worry about permission updates that might come up.

## Controlling Access to Resources via Tags

Access to resources can also be managed via tags. This can make your life much easier as you can grant a set of permissions easily and clearly.

One way of doing this is using conditions in your policy statements, e.g. with the `StringEquals` comparator.

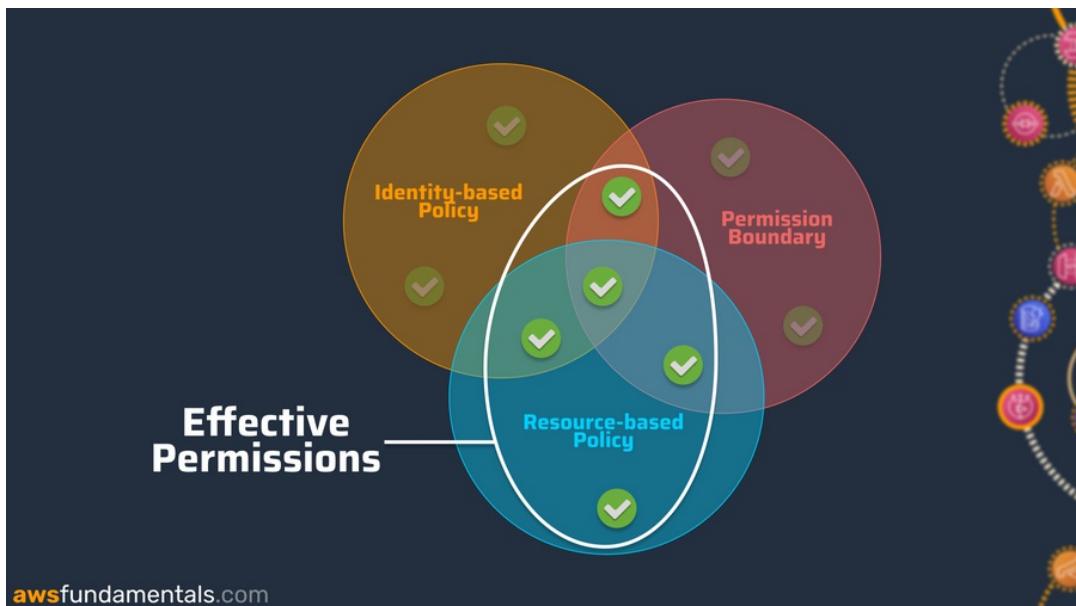
```
{
  sid: "ResourceTag",
  effect: "Allow",
  actions: [
    "cloudfront:*",
    "apigateway:*",
    "acm:/*"
  ],
  resources = [
    "arn:aws:acm:*:*:certificate/*"
  ],
  condition {
    test: "ForAnyValue:StringEquals",
    values: ["myapp"],
    variable: "aws:ResourceTag/App",
  }
}
```

Looking at the example statement above, we are granting **all** permissions for ACM, API Gateway, and CloudFront for resources that do have the tag `App` with value `my-first-app`. If we

properly assigned our certificate, API gateway deployment, and CloudFront distribution of our target app the proper tag, we can use this small statement to allow access to all resources of those three services.

### Permission Boundaries for Enforcing Role-Boundaries

AWS offers the advanced concept of permission boundaries, helping to create managed policies that restrict the maximum permissions that an identity-based policy can grant to any IAM entity. It's another dedicated policy type.



Looking at an example:

- Role A: having a policy attached that grants **Query** and **UpdateItem** permissions for **all DynamoDB tables** in the AWS account
- Role B: having a policy attached allows the creation (**CreateTable**) of DynamoDB tables.
- A permission boundary that is attached to both roles and assigns **Query** permissions for a **dedicated table**.

This will result in the following:

- Role A: is only able to Query the table explicitly allowed by the permission boundary.
- Role B: doesn't have any access to DynamoDB.

Worth noticing: Resource-based permissions **can't be restricted** via permission boundaries.

## More Policy Types for Advanced Use Cases

Additionally, there are less known policy types that you'll likely not stumble upon frequently:

- **Organizations Service Control Policies (SCPs):** you can attach AWS accounts to an AWS Organization to have centralized billing and permission management. This allows making use of SCPs that can be applied to one or several AWS accounts within your organization to restrict the permissions for entities in those accounts.
- **Access Control Lists (ACLs):** ACLs are service policies not defined as JSONs that control access from principals of other AWS accounts. They are similar to resource-based policies but are only available for a small set of AWS services, including S3, WAF, or VPC.
- **Session Policies:** in the earlier paragraphs we talked about temporary credentials. Session policies define who can create temporary credentials and which permissions can be included in those sessions. The session policy ultimately limits the permissions by only granting the intersection between both the resource-based & identity-based policy and the session policy.

## Creating and Testing Policies with Tool Support

A large part of working with IAM is learning about actions for your services so that you can apply the necessary permissions to users or roles. This can be tedious as the number of actions drastically vary from service to service and can include dozens of different actions. Even with a large set of actions, it's important to not fall back on using wildcards to assign permissions that are too wide and not necessary for your application to fulfill their work.

Because of that, AWS offers two fundamental and very useful tools for analyzing your policies again best practices and testing them beforehand.

### Validating Your Policies against Best Practices with AWS IAM Access Analyzer

IAM Access Analyzer validates any given policy against both best practices and policy grammar. If you're creating your policy within the AWS console, you'll be prompted with security, errors, warnings, and suggestions if you're in the JSON editor mode.

The screenshot shows the AWS IAM Policy Editor interface. At the top, there are tabs for "Visual editor" and "JSON", with "JSON" being the active tab. To the right of the tabs is a link "Import managed policy". The main area contains a JSON policy document with line numbers 1 through 24. Lines 19 and 20 contain ARN strings that are highlighted in red, indicating they are invalid. Below the policy editor, there is a status bar with "Security: 0", "Errors: 1", "Warnings: 0", and "Suggestions: 1". A search bar says "Search errors" and a "Learn more" link. On the right, there is a "Feedback" button and a close button. At the bottom, a message says "Ln 19, Col 16 Invalid ARN Account: The resource ARN account ID 01234567891 is not valid. Provide a 12-digit account ID. Learn more" with a link.

```

1- {
2-     "Version": "2012-10-17",
3-     "Statement": [
4-         {
5-             "Sid": "",
6-             "Effect": "Allow",
7-             "Action": "s3>ListAllMyBuckets",
8-             "Resource": "arn:aws:s3:::*"
9-         },
10-        {
11-            "Sid": "",
12-            "Effect": "Allow",
13-            "Action": [
14-                "dynamodb:Query",
15-                "dynamodb:PutItem",
16-                "dynamodb:GetItem"
17-            ],
18-            "Resource": [
19-                "arn:aws:dynamodb:eu-west-1:01234567891:table/preview-analytics",
20-                "arn:aws:dynamodb:eu-central-1:012345678912:table/preview-analytics"
21-            ]
22-        }
23-    ]
24-}

```

Errors: 1

Search errors

Feedback

Ln 19, Col 16 Invalid ARN Account: The resource ARN account ID 01234567891 is not valid. Provide a 12-digit account ID. [Learn more](#)

## IAM Policy Simulator to Test Your Policies before Applying Them

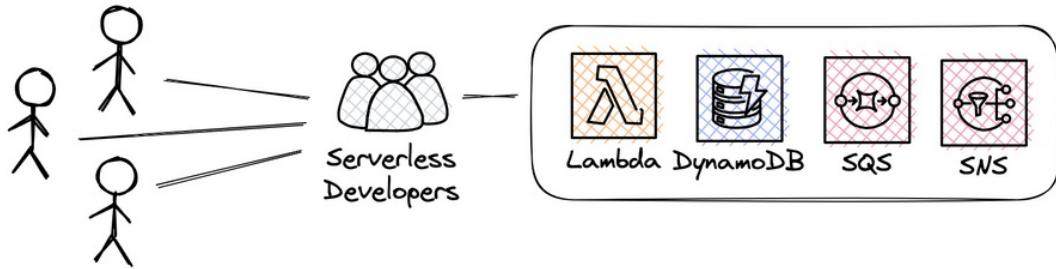
With AWS Policy Simulator you can test identity-based and resource-based policies. The simulator evaluates your policy in the same way as the engine that is used when real requests are received at AWS.

## Use Cases for AWS IAM

If taken seriously, this has to be an endless list as IAM is everywhere in your AWS ecosystem. Let's focus on beginner-friendly aspects that are easy to set up in your own AWS account.

### Creating Users with Restricted Permissions That Are Managed via a Group

Working with AWS can feel terrifying in the beginning, as you can't enforce any limits on costs. Restricting your daily users from specific services that you currently don't need, but could lead to immersive costs when not handled properly, can calm you.



In the example, we can create a dedicated group for Serverless developers who are working with a very specific set of services. In our case with Lambda, DynamoDB, SQS, and SNS. By restricting their permissions to only those services, we can ensure that no pricy EC2 instances or RDS clusters are created and never cleaned up.

### **Creating an S3 Bucket That's Only Accessible for Users with Activated MFA**

Amazon S3 is a great storage for all kinds of unstructured data. This includes cloud trail logs or archived application logs that need to be kept for compliance reasons.

Often, such data contains sensitive information that should be protected as well as possible. As known, human failure is how many data leaks are created and how sensitive data got exposed to the internet. Due to the sheer unlimited amount of accounts a single person does have when being a regular internet user, people tend to reuse passwords or keep them simple.

Due to that, a security incident in one service can lead to data theft in another service. That's one more reason for using multi-factor authentication where possible.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Action": "s3:*",
      "Resource": ["arn:aws:s3:::my-restricted-bucket/*"],
      "Condition": {
        "BoolIfExists": {
          "aws:MultiFactorAuthPresent": "false"
        }
      }
    },
    {
      ...
    }
  ]
}
```

```
        "Effect": "Allow",
        "Action": "s3:*",
        "Resource": ["arn:aws:s3:::my-restricted-bucket/*"]
    }
]
}
```

For sensitive S3 buckets, we can create a resource policy that only allows access for IAM users that do have MFA enabled.

This policy denies all S3 actions on the **my-restricted-bucket** bucket if the user does not have MFA enabled while allowing all S3 actions if the user has MFA enabled. As deny statements always overwrite allow statements, non-MFA users are restricted from accessing the bucket in any way.

## Tips & Tricks for the Real World

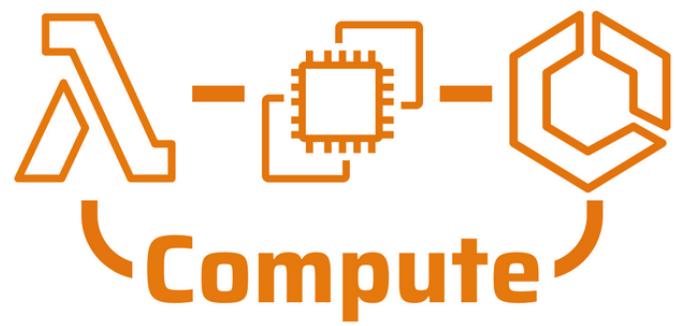
For IAM it's crucial to understand all of its fundamentals, but there's a list of the most important things to remember:

- **Try to go for the least privilege** - every time you create a new policy or a new statement to allow access for users or roles, try to grant the minimal set of actions to the minimal set of resources.
- **Rotate access keys regularly** - keys can be compromised. Rotating them regularly helps to reduce the risk of misuse.
- **Use groups to manage permissions** - with groups, you can manage the rights of multiple users in a single place. This reduces operations while also increasing security.
- **Use multi-factor authentication** - enforcing temporary security credentials from another device adds an extra layer of security to your IAM users, making it more difficult for an attacker to gain access to your resources.
- **Don't use your root account for your daily work** - access keys should be rotated regularly to maintain the security of your resources and help you to detect and prevent misuse of keys.

## **Final Words**

Completely mastering IAM is like a holy grail, as it's a complex topic and there will always be days when you're stuck on some permissions issue, regardless of your experience.

Nevertheless, it's worth investing the time to understand its core principles and gain knowledge while working with IAM regularly and avoid using wildcards.



# Compute

When we're building applications on AWS, we need to run our code somewhere: a computation service. There are several well-known and mature services that we can choose from. Let's go through a small history of how computing evolved in over the past years.

On-premise servers are physical machines that are owned and operated by an organization and are typically located in a dedicated data center. These servers require significant up-front investments, as well as ongoing maintenance and management.

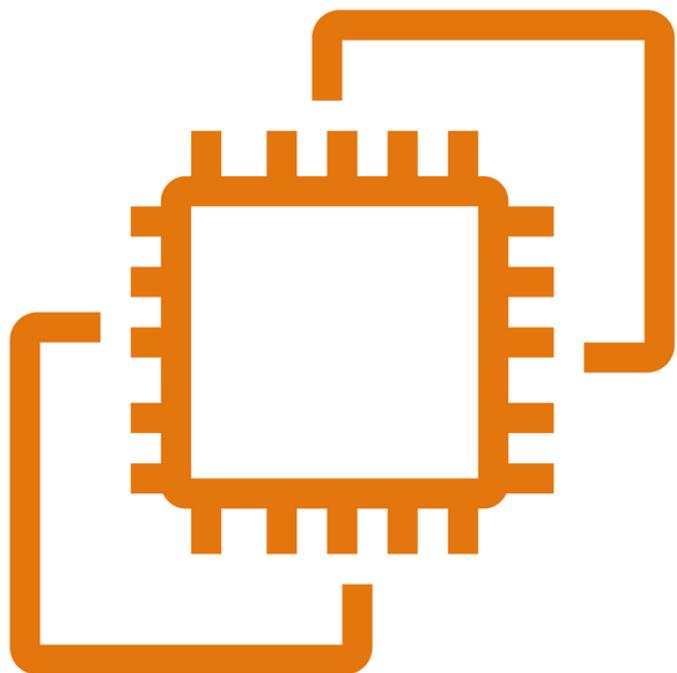


With the advent of cloud computing, it's now possible to rent virtual machines, such as with **EC2**. This allows scaling compute resources up and down as needed, not requiring significant up-front investments. However, this virtualization still requires management and maintenance by its users, like operating system updates or security patching.

The next abstraction came with containers. Containers allow the bundling of an application and all its dependencies into a single package that can be easily moved between environments. This allows for a much more efficient use of resources, and better isolation between different applications. Containers don't require a host operating system, just a container engine like Docker. For managing multiple containers, services like **ECS** were developed. It takes over the orchestration of containers so users don't have to take on the burdens of running, stopping, and managing clusters of containers.

The latest development in computing is Functions-as-a-Service, like **Lambda**. With Lambda, you can execute code without provisioning or managing any infrastructure. This approach is known as "serverless" computing, because the cloud provider handles all of the underlying infrastructure, scaling, and management, allowing developers to focus solely on writing and deploying code.

In the next chapters, we'll go through each of those three fundamental computing services.



Amazon **EC2**

# Launching Virtual Machines in the Cloud for Any Workload with EC2

## Introduction

Amazon Elastic Compute Cloud (Amazon EC2) is a web service that provides resizable computing capacity in the cloud.

It is designed to make web-scale cloud computing easier for developers and is one of the first services launched by AWS back in 2006. With EC2, you can rent virtual machines to run your own applications. This allows you to scale your application's capacity up or down as needed, making it a cost-effective solution for businesses. EC2 provides various instance types to suit workloads, including memory-optimized instances, compute-optimized instances, and GPU support. Generally speaking, EC2 comes in more than 500 variations which are perfectly designed for every specific need.

## Good Reasons for Choosing EC2 over Serverless Compute Options like Lambda

We've mentioned this in previous chapters several times: the future is cloud-native and serverless. But there are still several benefits and valid use cases for preferring Amazon EC2 (or orchestrated container services like ECS) over serverless approaches like Lambda, including:

1. **Computation** - EC2 provides much higher performance for certain workloads, particularly those that require a high level of (parallel) computing power. In comparison to AWS Lambda, EC2 instances can be dedicated to your own use and not shared with other customers. This is especially useful for computing intense operations like machine learning.
2. **Flexibility & Control** - Lambda limits you to certain runtimes and completely abstracts away underlying infrastructure and the operating system itself. With EC2, you have complete control over the operating system, libraries, and every software that runs on your instance. This allows you to run any application or workload on EC2, giving you maximum flexibility to build and deploy the solutions you need.
3. **Cost Control** - depending on your workloads and requirements, EC2 can be more cost-efficient than serverless approaches. Instead of having an on-demand charge based on the computation times and the number of requests like at AWS Lambda, you'll be billed for your chosen instances that can be a perfect fit for your needs in regard to instance types and sizes.

Overall, EC2 and other container approaches can be used together with serverless solutions like Lambda. EC2 goes well with steady and high computing requirements. Lambda offers you the best approach for small, often event-driven workloads that can benefit from the rapid scaling and flexibility of the serverless model.

## **Fundamentals on Working with EC2**

Before working with Amazon Elastic Compute Cloud (EC2), it is important to remember the fundamental concepts on which it is based.

### **Virtual Machines**

EC2 instances are virtual machines (VMs) in the cloud. Contrary to physical servers, virtual machines use software to create an abstraction from their underlying hardware. This allows securely hosting multiple virtual machines (even from different AWS customers) on the same physical hardware.

An instance comes with strong performance and security guarantees, even when the underlying hardware is shared. This is because AWS dedicates some resources of the host computer such as CPU, memory, and instance storage to the particular instance.

### **Amazon Machine Images (AMI)**

AMIs are AWS-maintained configurations that are required to launch an instance. It contains the operating system (e.g. Amazon Linux 2), architecture (32/64-bit x86 or 64-bit ARM), launch permissions, and storage for the root device.

Developers can also create shared images (Shared AMIs) that are made available for others to use. AWS can't ensure the integrity or security of the AMIs, **so it's within your own responsibility.**

### **Instances**

An EC2 instance is a virtual server that runs in the cloud. You can launch an instance from an AMI, and you can customize the instance's hardware and software configuration as needed. You pay for EC2 instances based on the type and number of instances you launch, as well as the usage of the underlying resources.

## **Regions and Availability Zones**

EC2 is available in multiple regions around the world, and each region is divided into multiple availability zones. Regions and availability zones allow you to launch instances in physically separate locations, which can be useful for disaster recovery, compliance, and performance.

## **Configuring an Instance for Your Needs**

When configuring an EC2 instance, there are several factors to consider, most importantly the instance type and storage as it will have a major impact on performance and the bill you'll receive at the end of the month.

### **Selecting a Fitting Instance Type**

Launching an instance requires you to specify an instance type. This determines the hardware configuration of your instance, including the number of CPU cores, amount of memory, and network performance. Choosing the right instance type is important to ensure that your instance has sufficient resources to meet the needs of your application. It should also not be over-provisioned so you don't end up paying for resources that you don't actually need.

The instance types are grouped into instance families:

- **General Purpose** - a balance between computing, memory, and networking resources.
- **Compute Optimized** - instances that offer high-performance processors.
- **Memory Optimized** - for in-memory processing of large data sets.
- **Storage Optimized** - best fit for workloads requiring high-performance reads and writes for large, locally saved, data sets.
- **Accelerated Computing** - instances that make use of hardware accelerators and co-processors for the fastest processing of specific operations like graphics processing or pattern matching, while also supporting the highest parallelism.

### **Choosing the Right Storage**

EC2 comes with a diverse set of possible storage options, each with a unique combination of durability and performance.

Regardless of your requirements, there will be a great fit available:

- **EC2 Instance Store** - providing temporary (ephemeral) storage for your instance, located on disks that are physically attached to the host computer. Use it for cached data, buffers, and temporary content. Data will be lost if the underlying disk fails or the instance stops, hibernates, or terminates.
- **Elastic Block Storage (EBS)** - block storage volumes that can be mounted to instances. Their persistence is independent of your instance's lifetime. Use it for long-term storage of data that require low-latency access, e.g. for databases.
- **Elastic File System (EFS)** - scalable file storage that can scale based on workload requirements. Use it for any workload that can suddenly increase or decrease storage needs. It can also be used for shared volumes.
- **Simple Storage Service (S3)** - data storage for unstructured data, that are stored with no hierarchy and are only accessed by a unique object identifier.

### **Launching Your Instance and Connecting to It**

Let's launch our first instance. After switching to EC2 via the management console, click on the Instances tab and select `Launch` on the top right to start the Launch Wizard for EC2.

#### **Going through the Launch Wizard**

The first thing we need to configure is the name of our instance. It will be assigned via the `Name` tag and you're also able to add additional tags based on your needs.

Next, we need to choose our Amazon Machine Image (AMI). As learned previously, the AMI is a template that contains the software configuration, the operating system, and further development tools. Besides choosing an existing one provided by AWS, you're also able to select one from the Marketplace (which can also introduce additional costs) or create your own one.

Let's stick to using the latest Amazon Linux 2 AMI.

The screenshot shows the AWS EC2 'Launch an instance' wizard. The current step is 'Application and OS Images (Amazon Machine Image)'. The interface includes fields for 'Name and tags', 'Software Image (AMI)', 'Virtual server type (instance type)', 'Firewall (security group)', and 'Storage (volumes)'. A search bar at the top allows users to search for AMIs. Below the search bar is a 'Quick Start' section with icons for various operating systems: Amazon Linux, macOS, Ubuntu, Windows, Red Hat, and SUSE. A 'Browse more AMIs' link is also present. The selected AMI is 'Amazon Linux 2 AMI (HVM) - Kernel 5.10, SSD Volume Type' (ami-01cae1550c0adea9c). The architecture is set to '64-bit (x86)'. The AMI ID is 'ami-01cae1550c0adea9c'. A 'Verified provider' badge is visible. On the right side, there is a summary panel with sections for 'Number of instances', 'Software Image (AMI)', 'Virtual server type (instance type)', 'Firewall (security group)', and 'Storage (volumes)'. A tooltip for the 'Free tier' is displayed, stating: 'Free tier: In your first year includes 750 hours of t2.micro (or t3.micro in the Regions in which t2.micro is unavailable) instance usage on free tier AMIs per month, 30 GB of EBS storage, 2 million I/Os, 1 GB of snapshots, and 100 GB of bandwidth to the internet.' Buttons for 'Cancel' and 'Launch Instance' are at the bottom.

Besides the image, you can also choose your preferred architecture including x86 and ARM. ARM runs on the latest Graviton2 processors by AWS that offer a better performance/price ratio but may cause conflicts with the supported software that you require.

The screenshot shows the AWS Lambda 'Create Function' wizard. The first panel, 'Instance type', displays the selected instance type as 't2.micro', which is 'Free tier eligible'. It includes details about the family (t2), vCPU count (1), memory (1 GiB), and pricing (On-Demand Linux: 0.0126 USD per Hour, On-Demand Windows: 0.0172 USD per Hour). A 'Compare instance types' link is also present. The second panel, 'Key pair (login)', shows a dropdown menu with 'Select' and a 'Create new key pair' button with a plus icon.

Going further, the next selection aims at the instance type. For testing purposes, you can always go with an instance type that is free tier eligible like the `t2.micro`. This means that you don't need to pay anything if you run one of those instances for the whole month. The EC2 free tier is only valid for the first 12 months after your account creation.

The next panel allows you to either select an existing SSH key pair or to create a new one that allows us to connect to the instance via our local terminal. Let's create a new pair here and we'll continue with the details in the upcoming sub-chapter of [Connecting to your Instance](#).

The screenshot shows the 'Network settings' section of an AWS instance configuration. At the top right is an 'Edit' button. Below it, under 'Network Info', is the VPC identifier 'vpc-03e559f382af5deda'. Under 'Subnet Info', it says 'No preference (Default subnet in any availability zone)'. Under 'Auto-assign public IP Info', it is set to 'Enable'. A 'Firewall (security groups)' section follows, with a note about security groups controlling traffic. It offers two options: 'Create security group' (radio button unselected) and 'Select existing security group' (radio button selected). A dropdown menu labeled 'Select security groups' contains one item: 'default sg-03e3f8df769310c30 X'. To the right of the dropdown is a 'Compare security group rules' link.

In the network section, you're prompted to select a VPC and a security group. Those networking settings are a part of VPC and help you to protect your instance from unwanted access.

Each AWS region comes with a **default VPC** for every region and a corresponding security group that allows all inbound and outbound traffic. We can select those and continue to the next step.

The screenshot shows the 'Configure storage' section. At the top right is an 'Advanced' link. It displays a configuration for a root volume: '1x 8 GiB gp2' selected from a dropdown. Below this is a note: 'Free tier eligible customers can get up to 30 GB of EBS General Purpose (SSD) or Magnetic storage' with a close 'X' button. An 'Add new volume' button is present. At the bottom, it shows '0 x File systems' and an 'Edit' button.

Lastly, we need to choose our preferred storage option, each with a different set of features as we learned earlier. Specify the storage settings for the instance, such as the size and type of the

EBS volumes which will be attached to our instance.

Now we're able to launch our instance and switch back to the overview panel.

Instances (1) <a href="#">Info</a>			
<input type="text"/> <a href="#">Find Instance by attribute or tag (case-sensitive)</a>			
<input type="checkbox"/>	Name	Instance ID	Instance state
<input type="checkbox"/>	awsfundamentals	<a href="#">i-0388d84f1c8fc730</a>	Pending

The instance will be available and ready after a few seconds when its state has switched to `Running`.

### Connecting to Your Instance

Now that we have created an instance we need a way to connect to it. The typical way is to use SSH. With SSH you can connect your local computer to the virtual machine. For securely connecting via SSH we either need a username and password or a key pair.

When we launched our instance previously, we were prompted to either create a new public & private key set or use an existing one. When generating a new pair, AWS will store the public part on the instance (concisely at `~/.ssh/authorized_keys`), while asking you to download the private part. AWS won't store the private part afterward, so if you lose it, you can't recover it.

## Create key pair

X

 We noticed that you didn't select a key pair. If you want to be able to connect to your instance it is recommended that you create one.

Key pairs allow you to connect to your instance securely.

Enter the name of the key pair below. When prompted, store the private key in a secure and accessible location on your computer. **You will need it later to connect to your instance.** [Learn more](#)

Create new key pair

Proceed without key pair

Key pair name

awsfundamentals

The name can include up to 255 ASCII characters. It can't include leading or trailing spaces.

Key pair type

RSA

RSA encrypted private and public key pair

ED25519

ED25519 encrypted private and public key pair (Not supported for Windows instances)

Private key file format

.pem

For use with OpenSSH

.ppk

For use with PuTTY

Cancel

Create key pair

You can also use AWS Systems Manager Session Manager to connect via a browser-based shell or AWS CLI. As an alternative, you can solely rely on AWS IAM to connect to your instances, not needing to manage any key pairs. This is achieved with EC2 Instance Connect. It's a good so make use of the session manager as it doesn't need to you have to open ports in your security groups.

Nevertheless, let's dive into how to connect via our SSH key and our terminal for example purposes.

1. Firstly we need to either get the public IP address or the domain name of the EC2 instance we created before. Those can be found by clicking on your instance ID and selecting the **Networking** tab.
2. Open up your terminal command prompt on your local machine and use the following command to connect to the instance: `ssh -i <$PATH_TO_SSH_FILE> ec2-user@$DNS_NAME` Replace the two variables with the path to your SSH file and the domain name of your instance and you should be able to connect and get a welcome message from your instance.

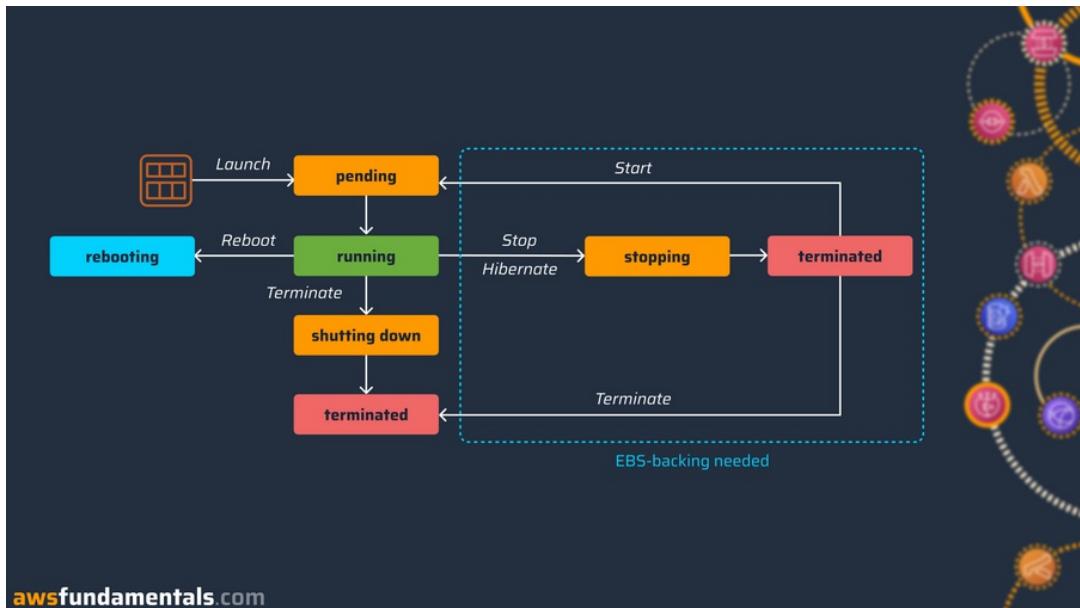
**Side note on this process:** depending on the operating system that is running on your EC2 instance, you may have to use another user name. For example, the user name for Amazon Linux instances is **ec2-user** while the user name for Ubuntu instances is **ubuntu**. You can find the correct user name for your instance in the EC2 management console under the description tab for the instance.

### Understanding the Instance Lifecycle States

Between the launch and the termination of your EC2 instance, there are more lifecycle states it transitions through.

- **Launch:** creating and launching an instance from an AMI into a VPC, subnet, and availability zone. From creating the machine to being available it will be in the pending state.
- **Stop & Start:** with EBS volumes you're also able to start and stop your instance. It keeps its instance ID and you won't be billed for the instance while it's stopped, but you'll still pay for the storage of any EBS volume.
- **Hibernate:** If supported by the OS, the instance can save its RAM contents to the EBS root volume. This pre-warmed state takes less time to initialize if it's again needed in production and isn't billed.
- **Reboot:** Restart your operating system. Your instance will keep private and public IP addresses.
- **Retire:** if underlying hardware experiences irreparable failure, AWS will schedule your instance for termination.
- **Terminate:** deleting your instance if no longer needed.

- **Recover:** instances can automatically recover either on status checks or hardware failures.



### Managing Block Stores and Snapshots for Your Instances

Contrary to the instance storage, block store volumes do persist over time and can be detached from and re-attached to your instances. EBS volumes also allow you to take point-in-time snapshots of your volumes, which are stored in Amazon S3 and can be used to create new volumes or restore existing ones.

Equally to your instances, EBS snapshots come with lifecycle management. This allows you to automate the creation, retention, and deletion of snapshots.

### Configuring Your Network and Securing Your Instance

We've launched our EC2 instance into the default VPC and attached it to the default security group. This made it available to the world and allows it to connect to the world on any port. That's perfectly fine for testing things out and exploring EC2's capabilities.

For the future, it's important to know about the networking and security features of VPC that are tightly coupled to EC2. We have dedicated a separate chapter to VPC but let's quickly revisit the fundamentals you need to remember for running your instances:

1. **Virtual Private Network** - Amazon VPC is a virtual network that enables you to isolate your AWS resources from the rest of the world and from other resources in your own AWS

account to enforce security boundaries by defining IP address ranges, subnets, and security policies with security groups (SGs) and network access control lists (Network ACLs or NACLs).

2. **Subnets** - Subnets are further sliced down parts of your VPC. A subnet can be public, meaning that instances in this subnet can communicate with the internet and are accessible over the internet, or private, meaning that instances are by default not connected to the internet and are not accessible via the internet. Private subnets should be used for critical components like databases, which should never be exposed to the internet but only for internal components like EC2 instances, containers, or Lambda functions that need to work with the saved data.
3. **Security Groups** - A security group can be viewed as a virtual, stateful firewall for your instances. It controls the traffic to and from each instance, allowing you to specify which inbound and outbound network traffic is allowed. Worth remembering: the default security group allows all outbound traffic and all inbound traffic from resources in that group. It's possible to create multiple security groups and also assign multiple groups to one or several instances.
4. **Network ACLs** - NACLs are another layer of security that allows you to control traffic to and from subnets in your VPC. Contrary to security groups, NACLs are stateless, meaning that you explicitly need to configure outbound allow rules, for your service to be able to answer requests. The stateful security groups allow such responses by default, if the requesting (inbound) direction was allowed.
5. **Elastic IP Addresses** - An Elastic IP is an address that can be assigned to your EC2 instances. Each elastic IP address can be mapped to an instance and also be rapidly re-mapped to another instance if a failure occurs.
6. **Gateways** - There are two important gateway types in AWS: the Internet and the NAT gateway. On the one hand, the internet gateway allows communication between instances in your public subnet and the internet. A NAT gateway on the other hand enables instances in a private subnet to connect to the internet or other AWS services but prevents the Internet from initiating connections to those instances.

Summarized, the network and security options for EC2 instances enable you to control the network connectivity and access to your instances and protect them from external threats.

## **Thinking Ahead to save Money by Diving into the Different Purchase Options**

EC2 provides different purchase options that drastically vary in pricing. If you plan to use EC2 regularly, it's a duty to understand the purchase options.

- **On-Demand** - the default option, billed by the second after an instance is launched.
- **Reserved / Savings Plan** - make a commitment to either an instance configuration (reserved) or usage in USD per hour (savings plan) for a dedicated time frame (1 to 3 years) to significantly lower your bill in comparison to on-demand pricing.
- **Scheduled** - reserve capacity based on a schedule for a year. This is currently not available at AWS and is not planned to be reactivated.
- **Spot** - use spare EC2 capacity for pricing based on offer & demand, much lower than on-demand. Keep in mind that your workload can be interrupted if there is not enough spot capacity available.
- **Dedicated Instances & Hosts** - launch instances onto physical servers that are isolated at the hardware level. Dedicated instances might share hardware with other instances in the same AWS account, while hosts are not.
- **On-Demand Capacity Reservations** - preplan EC2 capacity for your instances in an availability zone. Those reservations can be created at any time (no yearly commitments needed) and are independent of the discounts of saving plans or reserved instances.

## **Monitoring and Troubleshooting Your Instances and Configurations**

There is a variety of services that helps you to monitor the availability and performance of your instances.

The simplest tools are system status checks and instance status checks. System status checks are provided and taken care of by AWS and detecting problems with the software or physical hardware, e.g. network connectivity or power loss. Instance status checks are there to detect configuration issues that need to be addressed by yourself, including misconfigured networks or exhausted memory.

CloudWatch and EventBridge help you gain more insights into your instances and also react to incidents with automated routines. EC2 automatically sends metrics to CloudWatch, including CPU utilization as well as network and disk usage.

## **Using the AWS Marketplace to Get Pre-Configured AMIs for Almost Any Requirement**

While many AWS-provided AMIs come with additional software, especially for software development purposes, it most likely doesn't include all the software you need.

As described earlier, shared AMIs come to tackle this issue. Additionally, there are shared images that are also paid and therefore increase the price for your instance types. When selecting a paid AMI from AWS Marketplace you'll be informed about the additional usage fees.

## **Going One Step Further: Creating Auto-Scaling Rules to Adapt to Changing Loads**

You're able to organize your instances into logical groups that can be used to scale based on policies, including:

- **Target Tracking:** specify target values for a metric, e.g. average CPU or memory consumption.
- **Step Scaling:** define one or several thresholds for metrics that will then trigger a scale-in or out event.
- **Scaling based on SQS:** if your instances' workloads are received via SQS, you can scale based on the queue's backlog. This helps you add more workers to a fast-growing queue and remove workers if the queue's state is less busy.
- **Scheduled Scaling:** scale based on time and date if you're aware of traffic or workload patterns.

## **Use Cases for EC2 Instances**

When thinking about use cases for EC2, the possibilities are really endless depending on your need. There's nothing you can't do.

We'll focus on some small-scale and rather simple scenarios to get you started feeling comfortable working with EC2.

### **Use Case 1: Running Compute-Intensive Jobs with Spot Instances**

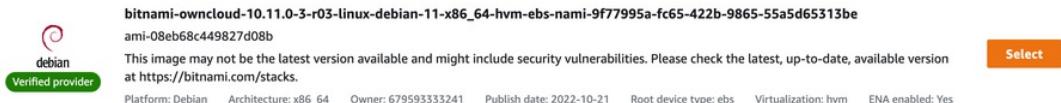
One of EC2's major benefits is the option to run tasks on compute-optimized instances. With spot instances, you'll also benefit from significant discounts. By selecting `Spot Requests` you can create a bid for a desired number of spot instances, which will automatically launch the instances when they are available at the price you specified.

Once the instances are launched, you can use them to run your compute-intensive job. Monitor the status of the Spot Instances to ensure they are running as expected, and be prepared to handle interruption if the instances will be terminated by AWS.

After your job is done, you should terminate the instances to avoid unnecessary charges.

### **Use Case 2: Hosting Your Own Cloud Backup Storage for All Your Devices with an AMI from the Marketplace**

You can use EC2 to set up your own private dropbox-like cloud storage. With AMIs, you don't have to build this yourself but rely on a marketplace or community image, e.g. for ownCloud.



Generally, there are a lot of great AMIs available for all different use cases. You can also run your own email server, host your own mail server for Minecraft, and much more.

### **Use Case 3: Building a Continuous Integration & Delivery Pipeline for iOS Applications**

With EC2, you can run any operating system you want. That's especially helpful for system-bound development chains like iOS applications. You can't avoid working with macOS.

By launching an EC2 instance with macOS, you can build your own delivery pipeline for iOS applications with the CI/CD tool of your choice. Please note that setting up such a solution can be complex and will require knowledge of macOS, XCode, and CI/CD tools.

### **Use Case 4: Setting up a Simple VPN Gateway**

EC2 instances can be used to run simple proxy servers, that can be used to connect to the internet using a different IP or to access specific services.

## **Tips & Tricks for the Real World**

- **Keeping a focus on your costs** - it's easy to forget about running instances if you're used to Serverless services like AWS Lambda. You'll be billed by the second. Remember to monitor your EC2 usage and costs, and stop or terminate instances when you no longer need them in order to minimize unnecessary expenses. If you're using EBS volumes, remember that you'll be still billed for existing volumes even if your instance is stopped.
- **Making use of the Marketplace** - often, there's no need to reinvent the wheel. If you have a specific need for something you want to do with an EC2 instance, there's probably an AMI on the marketplace that has everything you need. There's also a good chance that the AMI doesn't come with additional costs.
- **Proper configuration of security groups and NACLs** - make sure that your instance's security group has only the necessary open ports and IP ranges it really needs. Maybe a private subnet is enough to fulfill its job, so it doesn't need a route from or to the internet.
- **The default storage is ephemeral** - when you're running through the launch wizard and you're not selecting EBS, S3, or EFS for persistent data storage, all the data on your instance will be lost when it's terminated. This includes possible hardware failures in the data centers of AWS.

## Final Words

EC2 is the key service for applications that require a high level of customization and control, while serverless services like Lambda are a good choice for applications that need to run code in response to specific events or triggers. Both services can be used together, as well as with other AWS services, to build a wide range of applications and solutions.



Amazon **ECS**

# Running and Orchestrating Containers with ECS and Fargate

## Introduction

Amazon Elastic Container Service (ECS) is a highly scalable and fast container management service. It offers a management plane to orchestrate containers of your cluster. Simply run, stop & manage containers.

ECS comes along with many features to ease your development process and reduce operations and liabilities.

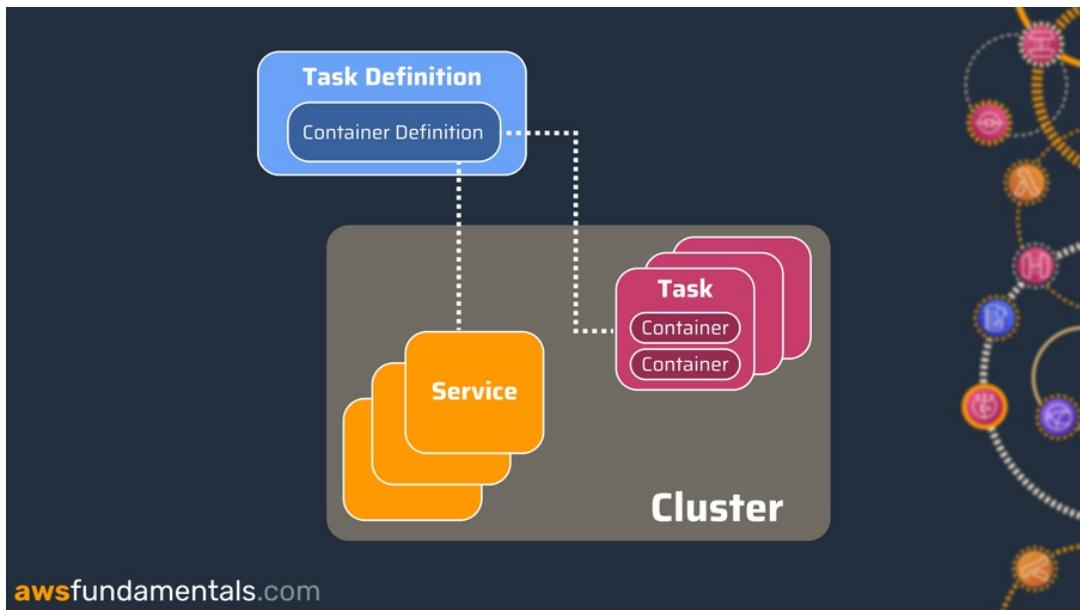
- Don't worry about underlying infrastructure with the **Fargate launch type**. You'll only determine which container image you want to run and what workload capabilities you require in terms of memory or virtual CPUs.
- It's fully integrated with AWS IAM. You can define fine-granular permissions based on your requirements and never think about users or passwords. Define any level of isolation that you want to or require from a compliance perspective.
- CloudWatch integration with metrics and log messages.

As we're believing in a cloud-native future, we'll only break down the basics of the EC2 launch types and not go into detail, but focus on Fargate.

ECS is one of the most battle-tested services of AWS and often a perfect fit for critical core infrastructure that needs to handle high-volume request microservice APIs.

## Understanding the Key Terms of ECS

While exploring and learning about ECS, you'll face a lot of key terms that are neither intuitive nor easy to grasp in the first place. Nevertheless, they are crucial to understanding how all the internals of ECS plays together. Let's dive into containers, tasks and task definitions, services, and clusters.



### Containers: Lightweight and Portable Packages That Actually Run Your Applications

One of the core building blocks of any container service is **Docker**. Docker's key feature is that it enables you to create lightweight environments to run your application, regardless of the underlying operating system.

This lightweight environment is called a **container** - so it contains all the necessary information to be executed on any machine - like certain versions of libraries or languages. Additionally, you're able to run multiple containers on the same machine. This includes intra-container communication without compromising the security of the host machine as everything is strictly separated.

With a growing application and therefore a growing number of containers, you'll be facing operating challenges regarding the management and orchestration of your container landscape: deployments, scheduling, and scaling of your containers, and many more.

ECS was built to help with that tedious task by offering a management plane that simplifies all those tasks so that developers can focus more on application development.

### Task Definition: The Blueprint to Run Your Containers

A task definition is a blueprint to launch one or several containers.

Properties that are part of your general task definition:

- **Launch type** - which service do you want to use to execute your tasks, either EC2, Fargate, or External?
- **Roles** - you'll need to have two dedicated roles for ECS. On the one hand, a task execution role needs the required permissions to start containers defined in a task (e.g. pulling images from ECR, retrieving secrets from AWS Secrets Manager), on the other hand, a task role that grants permissions to the actual application that runs inside your containers (e.g. querying or writing data from DynamoDB or sending messages to SQS).
- **Container image** - the docker image you want to run which resides in a container registry, e.g. Amazon's Elastic Container Registry.
- **CPU & memory allocation** - the compute resources that you want to assign to the task definition. The values can be assigned depending on your launch type. Generally, EC2 offers way higher compute resource capabilities than Fargate.
- **Environment variables** - key-value pairs that are injected into your service. Those are mostly used to inject stage-depend parameters into your application so that you can use generic container images that don't include hardcoded values.
- **Secrets** - you can securely inject sensitive data from AWS Secrets Manager or Systems Manager Parameter Store. ECS will take care of retrieving the secret values at execution times and providing them to your container instance. Don't forget to pass the necessary read permissions for your secrets to the task execution role.
- **Logging configuration** - define the log driver you want to use and the destination where ECS should send your logs. The available log drivers are also dependent on your launch type, e.g. Fargate only supports `awslogs`, `splunk`, and `awsfirelens` while EC2 additionally offers `fluentd` or `json-file`.
- **Exposed ports** - if your containers need to accept inbound traffic, this is the place to define how ports are mapped between ECS and your container image.

#### **Task - A Containerized Application That Is Deployed to Run on EC2 or Fargate**

A task is an actual execution of a task definition. They consist of a set of containers that are run together on the same host. Tasks are defined using the Docker Compose file format, which allows you to specify the container image, environment variables, port mappings, and other options for each container in the task.

You're also able to launch tasks directly. The task will stay active until it's either stopped or

exists on its own. In both cases, no replacement will be launched.

### **Service - Managing a Group of Tasks**

A service is a long-running process that manages a group of tasks and guarantees that a desired number of tasks are running all the time. If a task stops because a container exists due to errors and the number of healthy tasks falls below your threshold, ECS will take care of it and start a new task.

### **Clusters - A Logical Grouping of Container Instances**

A cluster is a logical grouping of tasks or services. Tasks and services run on infrastructure that is registered to a cluster, either provided by AWS Fargate, EC2 instances managed by yourself or on-premise servers, or virtual machines that are remotely managed.

### **Launch Types - The Way in Which Tasks Are Run and Managed**

We've gone over the fundamentals of containers and ECS' ability to orchestrate those. But which service actually runs your containers? With ECS, you can choose between multiple options.

#### **EC2 - Using Your Own Instances to Run Tasks within a Cluster**

Deploy EC2 instances in your clusters to run your containers. With EC2, you have full control over the underlying infrastructure, including the instance type, operating system, and security groups.

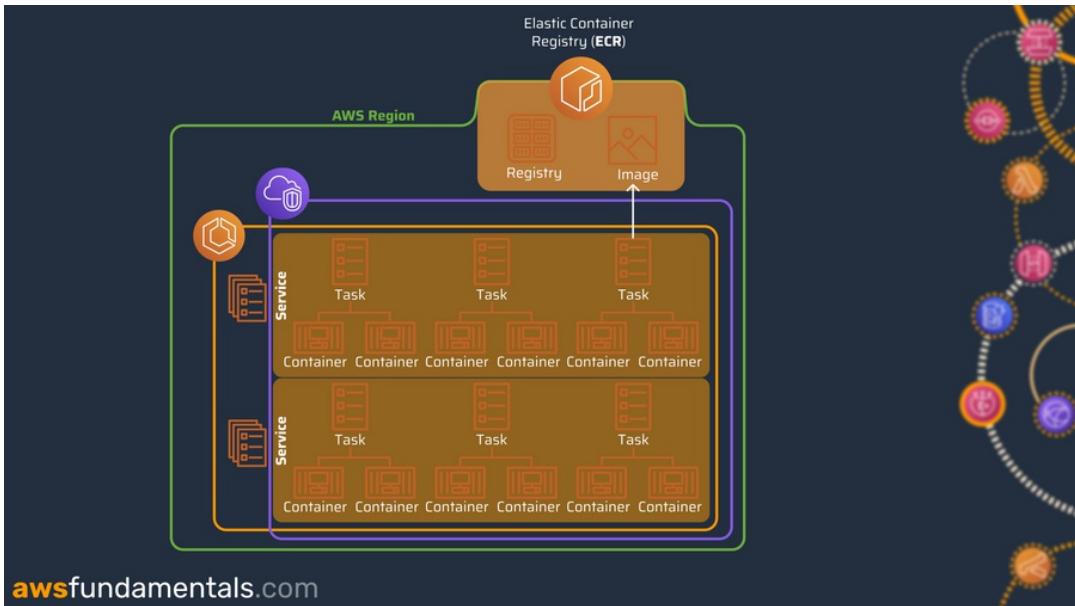
It's a good launch type choice for workloads that require consistently high CPU and memory, workloads that are optimized for pricing, or applications with persistent storage requirements.

#### **Fargate - Run Tasks without Having to Manage the Underlying Infrastructure**

Serverless pay-as-you-go option where you don't maintain any infrastructure. With Fargate, you only need to specify the task definition, cluster, and desired number of tasks, and ECS takes care of the rest. Fargate automatically provisions the necessary compute resources, such as Amazon Elastic Compute Cloud (EC2) instances, and runs the tasks on those instances.

It's the recommended launch type for tiny to large workloads that may require low overhead and experience occasional bursts. If you're not perfectly certain about your full requirements,

always go for the Fargate launch type to minimize operations and liabilities.



#### External - Orchestrating Your on-Premise Containers

If you're running containers on-premise, you can also make use of ECS orchestration capabilities by registering your containers remotely.

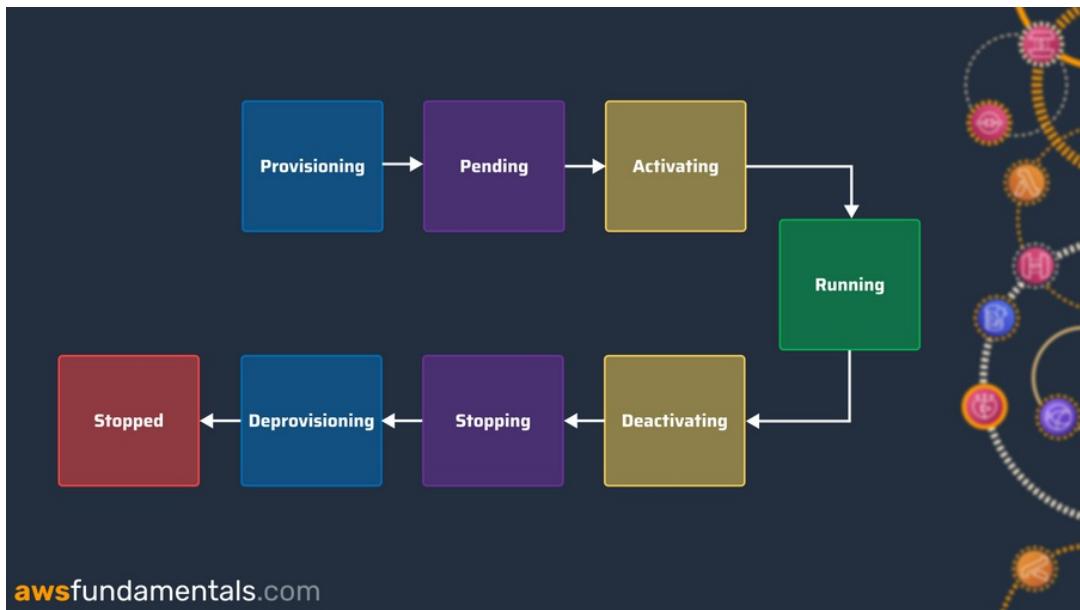
#### Task Scheduling - Running Your Tasks

Task scheduling refers to the process of assigning tasks to container instances within a cluster. ECS uses an efficient, dynamic scheduling algorithm to determine which container instances are available and suitable to run a task, and then schedules the task onto one of those instances.

ECS provides different scheduling capabilities, including a service scheduler or manually running tasks.

#### How Your Task Is Passed via Different States in Its Lifecycle

A task passes through different lifecycle states, regardless if it was started manually or as part of a service. Amazon's ECS container agent tracks all state transitions, the last known state as well as the desired state.



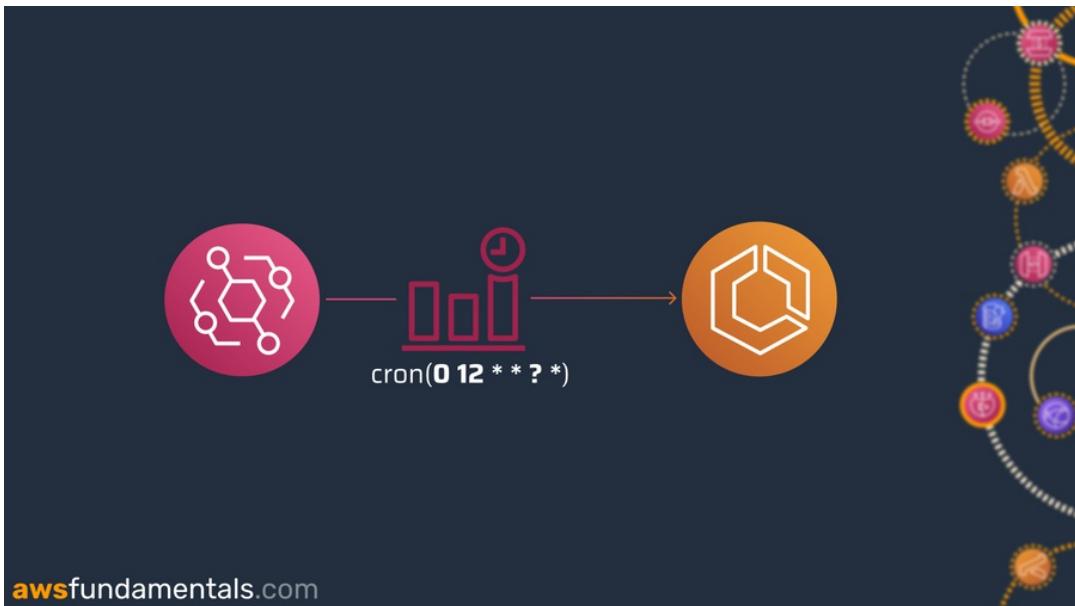
The lifecycle of a task includes the following states:

- **Provisioning** - Preconditions are in progress, e.g. attaching the Elastic Network Interface (ENI) if the task resides in a VPC.
- **Pending** - Waiting until the required resources for the task are available
- **Activating** - Final steps that takes place before moving to the running state, e.g. if attached to load balancing, ECS takes care of registering the task to the target groups.
- **Running** - The task is successfully running
- **Deactivating** - Necessary steps to take place before stopping the task, e.g. detaching from target groups if the task is part of a load balancer.
- **Stopping** - Gracefully shutting down the containers via SIGTERM signals. If a container does not stop within the configured timeout, there will be a forceful shutdown via SIGKILL.
- **Deprovisioning** - Final steps before transitioning to the stopped state, e.g. detaching the ENI if the task resides in a VPC.
- **Stopped** - The task has been successfully stopped.

For batch jobs, the task will simply run through the states while other strategies like the service scheduler try to keep states in the running state indefinitely or respectively scale tasks horizontally as needed.

## Running Tasks on a Regular Schedule

You can run tasks on schedule via EventBridge events.



The example will start a task every day at 12 PM UTC.

The EventBridge service needs several permissions to run the ECS task - similar to the task execution role that we need to assign to the task definition.

## Running Standalone Tasks on Demand

If you're developing an application and you're not ready to deploy it with a service scheduler, you can make use of standalone tasks which you can trigger.

If you've already created a task definition, just go to the definition pane and select the task you want to run. You can either select the latest revision or some previously created revision of your task definition.

Afterward, just click on `Actions > Run Task`. The run page will ask you to select the launch type, the cluster to use, the number of tasks to launch, and a name for the task group.

Depending on your network mode you have to provide VPC details. The following optional steps about task placements, which task placement strategy should be used, and which constraints apply can be skipped together with optional overwrites for environment variables, task, and task execution roles.

Finally, choose `Run Task` to submit your task and wait until your containers move into the running state.

### Persisting Your Images at the Elastic Container Registry

ECS with Fargate solely relies on **container images**. You can host those images at **Amazon Elastic Container Registry (ECR)**. It's a fully-managed container registry that allows you to host and share an unlimited amount of images and artifacts.

It natively integrates with ECS and also Amazon Elastic Kubernetes Service (EKS) and AWS Lambda.

There are no upfront costs. You're paying for the amount of data and transfer costs to the internet. This means: when Fargate is pulling images so it can run your container images inside tasks, this won't introduce any costs.

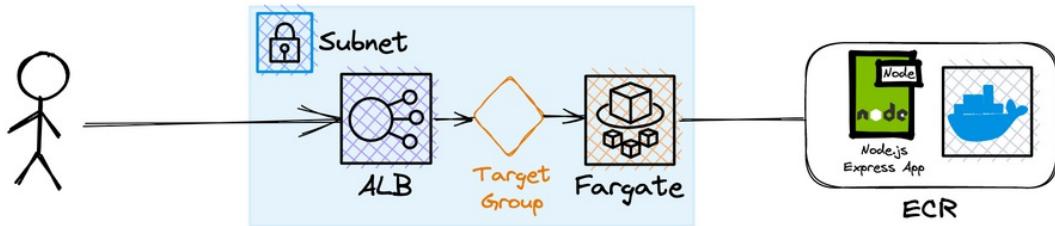
### Creating Our First ECS Service That Runs a Node.js Application in Fargate

After we've gone through all the knowledge, we can build our first service to run a docker-based application with ECS in Fargate.

We need to:

1. Initialize a new container registry at ECR which will later be the source where ECS pulls our application.
2. Set up our sample application, package it into a Docker image, and push it to our new repository.
3. Create a new ECS cluster. The cluster will later have our new service.
4. Set up a new task definition that references our image in ECR. It's the blueprint for running our application in a Fargate task.
5. Finally, launch our Serverless service. The service will take care of orchestrating our containers. It will guarantee that a certain number of tasks are healthy at all times.
6. Expose our tasks via an Application Load Balancer (ALB) to the internet. With this, we'll also create a target group that monitors our Fargate tasks for their healthiness.
7. Submit an HTTP GET to our application to receive our expected response.

A very simple architecture diagram, not going into the task, task definition, service, and cluster details of ECS, would look like the following:



Let's go through each step and revise the fundamentals.

### A New ECR Repository for Our Image

Let's create a new repository in ECR. It should be private so it's not available unless we're authenticated with valid credentials for our AWS account.

Amazon ECR > Repositories > Create repository

## Create repository

### General settings

**Visibility settings** | [Info](#)  
Choose the visibility setting for the repository.

**Private**  
Access is managed by IAM and repository policy permissions.

**Public**  
Publicly visible and accessible for image pulls.

**Repository name**  
Provide a concise name. A developer should be able to identify the repository contents by the name.

157088858309.dkr.ecr.eu-west-1.amazonaws.com/ awsfundamentals

15 out of 256 characters maximum (2 minimum). The name must start with a letter and can only contain lowercase letters, numbers, hyphens, underscores, periods and forward slashes.

**Tag immutability** | [Info](#)  
Enable tag immutability to prevent image tags from being overwritten by subsequent image pushes using the same tag. Disable tag immutability to allow image tags to be overwritten.

**Disabled**

**Note**: Once a repository has been created, the visibility setting of the repository can't be changed.

After clicking create, you'll be taken to the new repository. Click on the top right at `View push commands` to get all the necessary commands we need to work with our new repository.

It will look like this:

```
# for logging into our repository
aws ecr get-login-password --region eu-west-1 | docker login --username
AWS --password-stdin 157088858309.dkr.ecr.eu-west-1.amazonaws.com
# for building an image: in this case from the Dockerfile in the current
Directory
docker build -t awsfundamentals .
# for tagging it properly
# a tag is a unique identifier for an image
docker tag awsfundamentals:latest <account-id>.dkr.ecr.eu-west-
1.amazonaws.com/awsfundamentals:latest
# pushing it to our registry
docker push <account-id>.dkr.ecr.eu-west-
1.amazonaws.com/awsfundamentals:latest
```

Let's only focus on the first command `aws ecr get-login-password`. This will log us into our repository so we can later upload new images into it.

### **Building and Pushing an Image with a Simple Node.js Application**

Our repository is ready to receive its first image. Let's create a Docker image that runs a Node.js application with the Express Framework so we can listen to HTTP requests via the following steps:

1. Create a new directory for your project and navigate to it.

```
mkdir fargate-nodejs
cd fargate-nodejs
```

2. Let's set up a new Node.js project by running `npm init`. We can set the project name and skip everything else. Afterward, we'll install express via `npm i express`.

```
npm init
npm i express
```

3. Now we can write our actual application by creating a new `index.js` file in the project directory. The only thing we do is return `Hello World!` for all requests on the root path.

```
const express = require('express');
const app = express();
```

```

# our express router that listens to the base route
app.get('/', (req, res) => {
    res.send('Hello World');
});

# starting our server by listening to port 3000
app.listen(3000, () => {
    console.log('Server listening on port 3000');
});

```

4. We've got our small application. Now we can put it into a Docker image. Let's create a `Dockerfile` with the contents below. It will build our image with the necessary dependencies from our `package.json` file and add our `index.js` file.

```

# the image base: a simple Linux-based image
FROM node:16-alpine

# set our working directory in our image
WORKDIR /app

# copy our package.json which includes our dependencies
COPY package.json ${WORKDIR}
# install our dependencies
RUN npm install
# copy our actual application code
COPY *.js ${WORKDIR}
# expose the port on which we run our application
EXPOSE 3000
# run the application
CMD ["npm", "start"]

```

5. Our image is ready. We can now push it to ECR. Don't forget to run the Docker login command from above beforehand, or else you'll receive an authentication error. This also requires that your AWS CLI is properly set up with credentials.

```

docker build -t awsfundamentals .
docker tag awsfundamentals:latest <account-id>.dkr.ecr.eu-west-
1.amazonaws.com/awsfundamentals:latest
docker push <account-id>.dkr.ecr.eu-west-

```

1.amazonaws.com/awsfundamentals:latest

After you see the last push message and the final URL for our image, this step is finished and our image should be available at ECR.

### Creating a Cluster

Remember what we learned before: an ECS cluster is a logical grouping of tasks or services that you want to run either on one or several Amazon EC2 instances or in Fargate tasks. Let's create such a cluster that we can later use to run our service by clicking on the `Create cluster` button.

The screenshot shows the 'Create cluster' wizard in the AWS Elastic Container Service console. It consists of two main sections:

- Cluster configuration:** This section contains a 'Cluster name' input field where 'awsfundamentals' is entered. A note below states: "There can be a maximum of 255 characters. The valid characters are letters (uppercase and lowercase), numbers, hyphens, and underscores."
- Networking:** This section includes a 'VPC' configuration where 'vpc-03e559f382af5deda' is selected from a dropdown. Below it, 'Subnets' are chosen from a dropdown, with three subnets listed: 'subnet-055ae88627c3f85b8' (eu-west-1a), 'subnet-0f743066101484e0c' (eu-west-1c), and 'subnet-0ad655648225de94d' (eu-west-1b). Each subnet entry has a delete icon (X) next to it.

We don't need to worry too much about the VPC configuration and subnets. Let's stick to the default ones that are coming with every AWS account and region.

**Default namespace - optional**  
Select the namespace to specify a group of services that make up your application. You can overwrite this value at the service level.

awsfundamentals X

**▼ Infrastructure Info** Serverless

Your cluster is automatically configured for AWS Fargate (serverless) with two capacity providers. Add Amazon EC2 instances, or external instances using ECS Anywhere.

- AWS Fargate (serverless)**  
Pay as you go. Use if you have tiny, batch, or burst workloads or for zero maintenance overhead. The cluster has Fargate and Fargate Spot capacity providers by default.
- Amazon EC2 instances**  
Manual configurations. Use for large workloads with consistent resource demands.
- External instances using ECS Anywhere**  
Manual configurations. Use to add data center compute.

**► Monitoring - optional Info**  
Container Insights is off by default. When you use Container Insights, there is a cost associated with it.

**► Tags - optional Info**  
Tags help you to identify and organize your clusters.

Cancel Create

We want to run our tasks in Fargate so that we don't have to manage EC2 instances. That immensely reduces operations. As we're not managing any servers when using Fargate, this is also categorized as a Serverless technology.

Amazon Elastic Container Service > Clusters

**All Clusters Info**

Clusters (1)					<span style="border: 1px solid #ccc; border-radius: 50%; padding: 2px 5px;">C</span> <span style="background-color: #0072bc; color: white; border: 1px solid #0072bc; border-radius: 5px; padding: 2px 10px; font-weight: bold;">Create cluster</span>
<input type="text"/> Search clusters					< 1 > <span style="border: 1px solid #ccc; border-radius: 50%; padding: 2px 5px;">⚙️</span>
Cluster	Services	Tasks	CloudWatch monitoring	Capacity provider strategy	
awsfundamentals	0	No tasks running	<input checked="" type="checkbox"/> Default	No default found	

After you see your cluster's name, the creation is completed. Obviously, there won't be any active tasks or services right now, but we'll tackle this in the next paragraphs.

## Setting up a Task Definition

Fundamentals reminder: a task definition is a blueprint that describes the containerized application and how it should be launched with the ECS cluster.

Let's create a new task definition.

Firstly, we need to set a task definition name and the container details. This includes the image URL from one of our previous steps and the port we want to expose. In our case, it has to be port `3000`. Generally, you can run multiple containers in a single task definition. This also allows for inter-container communication (like container A talking to container B via localhost - all communication stays inside the task). It's also possible to define which containers are essential, meaning the need to be healthy to keep the task itself healthy.

As we're running a single container in our task, the essential flag is automatically activated.

### Configure task definition and containers

**Task definition configuration**

Task definition family | [Info](#)  
Specify a unique task definition family name.  
  
Up to 255 letters (uppercase and lowercase), numbers, hyphens, and underscores are allowed.

**Container - 1** | [Info](#) | [Essential container](#) | [Remove](#)

Container details  
Specify a name, container image, and whether the container should be marked as essential. Each task definition must have at least one essential container.

Name	Image URI	Essential container
<input type="text" value="awsfundamentals"/>	<input type="text" value="157088858309.dkr.ecr.eu-west-1.amazonaws.com/awstf"/>	<input checked="" type="checkbox" value="Yes"/> Yes

Port mappings | [Info](#)  
Add port mappings to allow the container to access ports on the host to send or receive traffic. Any changes to port mappings configuration impacts the associated service connect settings.

Container port	Protocol	Port name	App protocol
<input type="text" value="3000"/>	<input type="button" value="TCP"/>	<input type="text" value="awsfundamentals-3000-tcp"/>	<input type="button" value="HTTP"/>

[Add more port mappings](#)

Next, we need to configure our environment.

1. We can choose very low vCPU and memory settings for our sample application. We don't have any requirements for computation, as we're only serving a very simple application.
2. We don't need to configure a task role. If you remember, the task role is the role that will be used in our application if we're using the AWS-SDK to invoke other AWS services. As we don't integrate with other services, like a DynamoDB database, we don't need a dedicated role here.
3. We will select `Create new role` for the task execution role. ECS will create a new IAM role on our behalf which has the necessary permissions to launch and manage our tasks.
4. We'll also stick to only using the storage inside the container.

## Configure environment, storage, monitoring, and tags

**▼ Environment**  
Specify the infrastructure requirements for the task definition.

**App environment** [Info](#)  
Specify the infrastructure for the task definition.  
▼  
**AWS Fargate (serverless)** [X](#)

**Operating system/Architecture** [Info](#)  
▼

**Task size** [Info](#)  
Specify the amount of CPU and memory to reserve for your task.  
**CPU** ▼ **Memory** ▼

► **Container size - optional** [Info](#)

**▼ Task roles, network mode- conditional**

**Task role** [Info](#)  
A task IAM role allows containers in the task to make API requests to AWS services. You can create a task IAM role from the [IAM console](#).  
▼

**Task execution role** [Info](#)  
A task execution IAM role is used by the container agent to make AWS API requests on your behalf. If you don't already have a task execution IAM role created, we can create one for you.  
▼

**Network mode** [Info](#)  
The network mode that's used for your tasks. By default, when the AWS Fargate (serverless) app environment is selected, the awsvpc network mode is used. If you select Amazon EC2 instances app environment, you can use the awsvpc or bridge network mode.  
▼

By clicking on **Next** you'll be taken to the overview page. With **Create** we'll create our new task definition now.

### Establishing and Launching a New Service

An ECS service allows you to run and maintain a specified number of instances of a task definition simultaneously in an ECS cluster. The service ensures that the desired number of tasks are running and automatically replaces any tasks that become unhealthy for any reason.

Let's do exactly that by going to our cluster and clicking on the **Create** button in the currently

empty services list.

## Deployment configuration

**Application type** [Info](#)  
Specify what type of application you want to run.

**Service**  
Launch a group of tasks handling a long-running computing work that can be stopped and restarted. For example, a web application.

**Task**  
Launch a standalone task that runs and terminates. For example, a batch job.

**Task definition**  
Select an existing task definition. To create a new task definition, go to [Task definitions](#).

**Specify the revision manually**  
Manually input the revision instead of choosing from the 100 most recent revisions for the selected task definition family.

Family	Revision
<input type="button" value="awsfundamentals"/>	<input type="button" value="1 (LATEST)"/>

**Service name**  
Assign a unique name for this service.

**Service type** [Info](#)  
Specify the service type that the service scheduler will follow.

**Replica**  
Place and maintain a desired number of tasks across your cluster.

**Daemon**  
Place and maintain one copy of your task on each container instance.

**Desired tasks**  
Specify the number of tasks to launch.

► **Deployment options**

► **Deployment failure detection** [Info](#)

We'll stick to the basic compute configuration and then select our previously created task definition in the deployment configuration. Currently, there's only one revision. Each update for the task definition will publish a new version. This for example happens if we update the URL to our image after we've published a new version (= new image tag) to ECR.

At the next step of the wizard, we need to set up a new load balancer. Let's choose the Application Load Balancer (ALB).

▼ **Load balancing - optional**

**Load balancer type** [Info](#)  
Configure a load balancer to distribute incoming traffic across the tasks running in your service.

Application Load Balancer

**Application Load Balancer**  
Specify whether to create a new load balancer or choose an existing one.

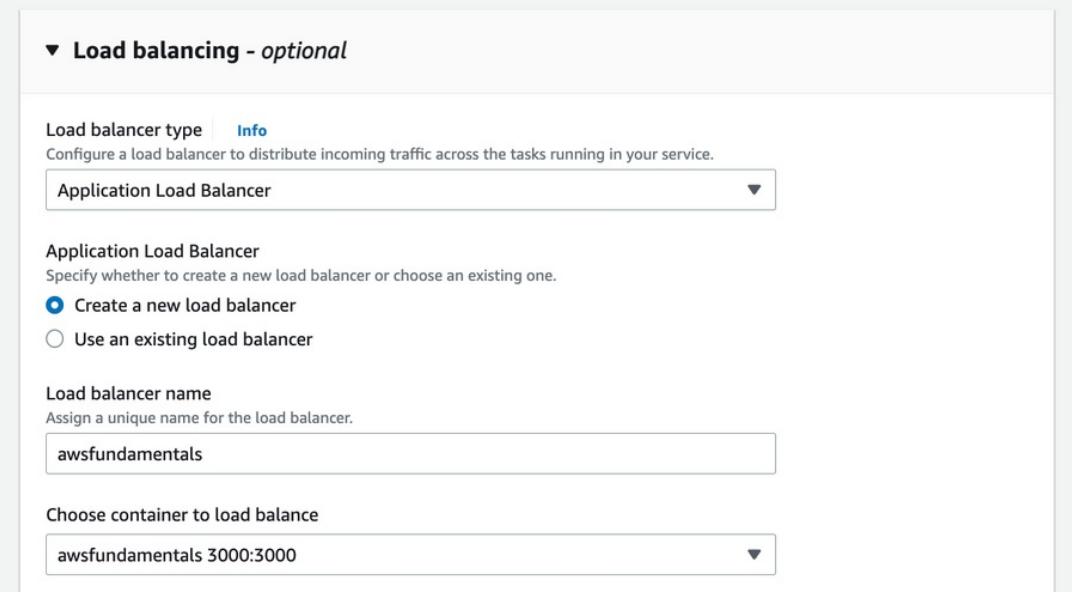
Create a new load balancer  
 Use an existing load balancer

**Load balancer name**  
Assign a unique name for the load balancer.

awsfundamentals

**Choose container to load balance**

awsfundamentals 3000:3000



The listener configuration defines which ports will be mapped to our container's exposed port 3000. For keeping it simple, we'll go with HTTP so we don't need to have a certificate at AWS Certificate Manager.

As our load balancer will be attached to the default security group. This default security group doesn't allow any incoming traffic. We need to jump over to the security groups pane at VPC and allow incoming traffic from the internet by adding a new rule that allows traffic from `0.0.0.0/0`.

We'll do this for example purposes. In a proper real-world scenario, you should strictly separate your resources into security groups that only have to required allow rules they really need.

**Listener** | [Info](#)  
Specify the port and protocol that the load balancer will listen for connection requests on.

Create new listener  
 Use an existing listener  
You need to select an existing load balancer.

<b>Port</b>	<b>Protocol</b>
80	HTTP

**Target group** | [Info](#)  
Specify whether to create a new target group or choose an existing one that the load balancer will use to route requests to the tasks in your service.

Create new target group  
 Use an existing target group  
You need to select an existing load balancer.

<b>Target group name</b>	<b>Protocol</b>
root	HTTP

**Health check path** | [Info](#)  
/

**Health check protocol**

HTTP
------

**Health check grace period** | [Info](#)  
10  
seconds

The target group will be used to route requests from the load balancer to our ECS tasks. We need to specify the protocol for the requests itself and the protocol and port for the health checks. Health checks will regularly call our application to check whether our app is in a healthy state. If the health checks consecutively fail, they won't be added to the load balancer's set of routed tasks.

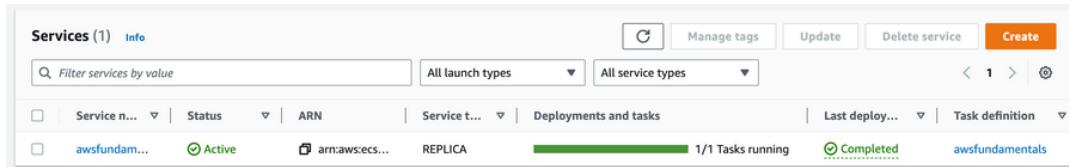
Let's complete our service creation by clicking [Create](#).

You can always check the progress of the service creation - or any other ECS-related infrastructure operations - in CloudFormation. It's neat to see where we are in the current state of the resource creation chain. Also, we can see when and why an operation fails and needs to rollback.

Stacks (3)		
<input type="text"/> Filter by stack name		
Stack name	Status	Created time
<input type="radio"/> ECS-Console-V2-Service-8cf69bcc-d14d-430e-b20c-d69d53c7469a	<span>CREATE_IN_PROGRESS</span>	2022-12-22 07:08:25 UTC+0100
<input type="radio"/> ECS-Console-V2-TaskDefinition-9cc07f35-c39c-478c-a5ee-bc6f84bc1560	<span>CREATE_COMPLETE</span>	2022-12-22 06:58:54 UTC+0100
<input type="radio"/> Infra-ECS-Cluster-6176d621-8dea-41c2-9dee-dd0de6c43fad	<span>CREATE_COMPLETE</span>	2022-12-22 06:21:35 UTC+0100

After the stack has finished the updates, we're already done.

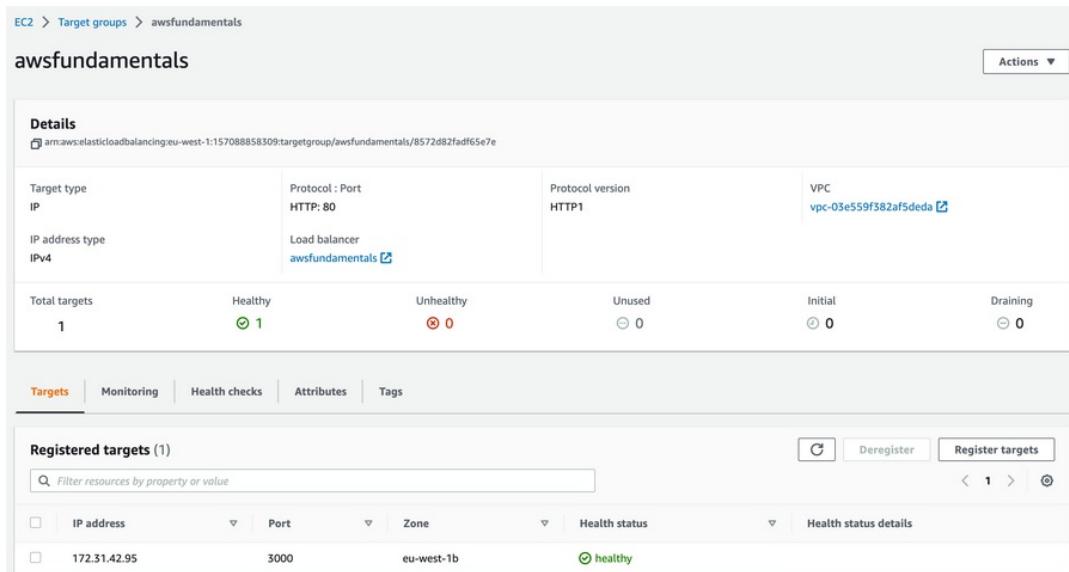
Let's jump back to our service overview and wait for ECS to finish its deployment.



The screenshot shows the AWS CloudFormation Services page. There is one service listed: 'awsfundamentals'. The status is 'Active', and it is associated with the ARN 'arn:aws:ecs...'. The deployment type is 'REPLICA', and there is 1/1 task running. The last deployment was completed successfully. The task definition is 'awsfundamentals'.

Let's also check that our load balancer was created correctly (this can take several minutes) and that our target group is healthy. Go to EC2 and jump to Load Balancing > Load Balancers Groups. Directly copy the load balancer's DNS name so we can later use it to submit a request to our application.

Now go to Load Balancing > Target Groups to view our target groups. You'll see our newly created group that's attached to our ALB. As we've only configured to have one task run at the same time, there should be one healthy target.



The screenshot shows the AWS Lambda Target groups page for the 'awsfundamentals' target group. The target group details are as follows:

Target type	Protocol : Port	Protocol version	VPC
IP	HTTP: 80	HTTP1	vpc-03e559f382af5deda
IP address type	Load balancer awsfundamentals		

Statistics for targets:

Total targets	Healthy	Unhealthy	Unused	Initial	Draining
1	1	0	0	0	0

Registered targets (1):

IP address	Port	Zone	Health status
172.31.42.95	3000	eu-west-1b	healthy

Let's see if we can reach our application via a simple cURL. By appending `-vvv` you'll see the verbose logs of the query.

```
curl http://awsfundamentals-1961464914.eu-west-1.elb.amazonaws.com -vvv
*   Trying 3.248.108.44:80...
* Connected to awsfundamentals-1961464914.eu-west-1.elb.amazonaws.com
* (3.248.108.44) port 80 (#0)
> GET / HTTP/1.1
```

```
> Host: awsfundamentals-1961464914.eu-west-1.elb.amazonaws.com
> User-Agent: curl/7.79.1
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< Date: Thu, 22 Dec 2022 06:52:17 GMT
< Content-Type: text/html; charset=utf-8
< Content-Length: 11
< Connection: keep-alive
< X-Powered-By: Express
< ETag: W/"b-Ck1VqNd45QIvq3AZd8XYQLvEhtA"
<
* Connection #0 to host awsfundamentals-1961464914.eu-west-
1.elb.amazonaws.com
* left intact
Hello World
```

There we go. We've successfully set up our first application with ECS and Fargate.

**Reminder:** as Load Balancers and ECS is charged by the hour, please don't forget to delete all of your resources after you're finished. This can be done by deleting our ECR repository and afterward deleting all of our CloudFormation stacks.

Start with the service's stack, continue with the one for the task definition and lastly delete the stack of the cluster. Don't be scared to delete those; there's nothing that can go wrong as they only include our sample's resources.

## Securing Your Tasks and Clusters

Let's revisit the shared responsibility model by AWS and its two pillars:

- **Security in the cloud** - Customers are responsible to maintain the security of their own data, operating systems, networks, identity & access management.
- **Security of the cloud** - AWS is responsible for securing the overall hardware and global infrastructure.

Transferring this to ECS with Fargate: your liability maps down to securing data and protecting it from misuse of your application and your infrastructure.

There are several steps you can take to ensure the security of your tasks and clusters:

1. **Use ECR instead of other container registries** - ECR is a secure container image registry that is managed by AWS. Access can be solely managed via IAM policies. Using IAM instead of shared credentials as you'd need for other external repositories is less likely to cause security incidents as there's no way to expose security credentials to unauthorized principals easily.
2. **Using IAM roles to control access to ECS resources and assigning minimal permissions** - With IAM roles you can easily assign permissions that only allow running tasks within specific clusters, but not deleting or modifying tasks.
3. **Using Security Groups to control traffic to the container instance** - Security groups allow you to specify which incoming traffic is allowed to reach the container instances, and can be used to protect against network-based attacks.
4. **Encrypt your data in transit and at rest** - ECS supports the usage of TLS to encrypt data in transit between container instances and other services. Additionally, you can encrypt your data at rest at other services like S3 or DynamoDB.

By following these best practices, you can help ensure the security of your ECS tasks and clusters and protect against potential threats.

## Deploying Updated Task Definitions

ECS comes with different deployment strategies: rolling updates, Blue/Green deployments with CodeDeploy, and external deployments.

The most common strategy is the rolling update, which will invoke the ECS scheduler to replace the currently running tasks with new tasks. It's bound to the configuration of the `minimumHealthyPercent` and `maximumPercent` values which determine the upper and lower bounds of number the healthy tasks.

Let's have a look at a service definition example:

- desired count of 3 containers
- minimum percentage: 100%
- maximum percentage: 150%

This means ECS is able to only launch a single new task definition, waiting until it's available

and then stopping one of the old tasks.

### **Speeding up Deployments by Fine-Tuning Different Settings**

Depending on your configuration, deploying new tasks to your cluster can take several minutes, even though your new containers may start quickly.

Let's have a look at a service that's integrated into a target group of a load balancer and the settings that can be adjusted.

### **ECS Deployment Settings**

In your service definition, you can tell ECS the minimum and maximum number of healthy task percentages will the task replacement is in progress. Reducing the minimum and increasing the parallel will help to improve deployment times as ECS is able to launch more tasks in parallel while also being able to stop old tasks faster.

### **Load Balancer Settings**

Clients usually keep connections open to servers to be able to reuse them for subsequent requests. This reduces latency as there's no additional handshake necessary. At task replacement, the target group of a load balancer will stop sending new connections downstream but wait for other connections to disconnect for a certain period of time. If you're not relying on real-time connections, e.g. like with web sockets, you can drain connections faster so ECS' scheduler doesn't have to wait a long time to stop tasks.

The target group of your load balancer also has health check settings which define how many successful health checks for new containers are required until the transit to the active state in the target group and how often these health checks are executed.

If you reduce the interval and required number of successful health checks, the target group will also mark containers as healthy way more quicker.

### **ECS Agent Settings**

When a task moves into the stopped state it sends a SIGTERM signal to the container to notify it about the incoming stopping of the container. This can be used to gracefully stop processes that are currently active and close connections. If your application is not actively listening for this signal or is ignoring it, there's no benefit in waiting for the 30s (the default value) until ECS can finally terminate the container process.

## **Monitoring Key Metrics of Your Tasks and Clusters with CloudWatch**

ECS with launch type Fargate is integrated with CloudWatch and forwards near-realtime metrics about CPU and memory utilization as well as the reservation by default without taking any manual steps. When using the EC2 launch type, it is recommended to use the latest container agent version but requires to have at least version 1.4.0 (Linux) or 1.0.0 (Windows) to be able to forward metrics.

The key metrics and events that should be monitored include:

1. **Task and cluster status & events** - You should monitor the statuses and events related to your tasks and clusters, such as task failures or cluster scaling events. You can use CloudWatch Events to set up alarms and notifications for these events and take appropriate action as needed.
2. **Resource utilization** - CloudWatch automatically collects CPU, memory, and network utilization of your container instances and tasks. Those utilizations should be used to automatically scale your clusters to meet capacity demands and ensure that there is no overloading (resulting in high latencies or timeouts) or over-provisioning (resulting in an unnecessarily higher bill) of your instances.
3. **Network traffic** - You should monitor the network traffic to and from your container instances to ensure that your tasks are communicating properly and to detect any unusual activity.

## **Automatically Scaling Your Containers Based on Traffic Demands**

ECS provides several options for scaling the number of tasks in a cluster based on the workload and available resources.

### **Step Scaling Policies to Scale in and Out Based on Utilization Thresholds**

As described before, we know the CPU and memory utilization of our containers due to ECS metrics that are forwarded to CloudWatch and can therefore determine how much load is on our cluster. Depending on those metrics we can create CloudWatch alerts that trigger scaling actions for our ECS cluster via step scaling policies.

With this policy type, you'll define how ECS should react to different utilisations (either CPU or memory). For example, we want to add a single new task if the CPU usage is above 70% for more than 2 minutes until a maximum number of 10 tasks. On the other hand, we want to stop

one task if the CPU utilization is below 40% for 2 minutes.

It's recommended to define a cool-down period that limits the frequency of starting or stopping tasks. Additionally, you're able to configure a time period for the deployment phase of a task so that the startup of containers, which usually results in spiking CPU usage, won't trigger the auto-scaling actions.

#### **Target Tracking Scaling Policies to Scale Based on Target Utilization**

The target tracking type is the recommended scaling policy by AWS. ECS Service Auto Scaling will create and manage the CloudWatch alarms that trigger the scaling actions. The scaling policy will add or remove tasks as required to keep the metric close to the specified target value.

We can for example set the average CPU utilization as a service metric for the auto-scaling policy to 75%. ECS will now take care of adding or removing tasks to keep the CPU utilization as close as possible to 75%.

#### **Scheduled Scaling Policies to Scale at Different Times of the Day**

Maybe it's already known that different times of the day will result in different traffic load patterns. If so, we can make use of scheduled scaling policies to adjust the number of tasks and therefore containers to our known load patterns.

ECS will then take care of adjusting the desired number of tasks based on the date and time.

#### **The Endless Use Cases of Container Services**

AWS ECS is a core building block in nearly any enterprise that is working with AWS. That's why you can find any use case out there that was already implemented with ECS. And that's where its strength comes from: it simplifies running containers and containers are everywhere in today's software landscape.

Instead of listing a lot of examples for all kinds of different applications with ECS at its heart, we'll explore something that you'll almost definitely have contact with when working as a software engineer in the cloud area.

## Migrating Existing on-Premise Applications to the Cloud

This is not a simple use case, but a real-world scenario for almost any engineer in the cloud stratosphere. As this book wants to give as many examples as possible to get you into the building, this has to be included. A lot of companies are in the process of migrating their on-premise applications to the cloud. Often, the target core service to migrate these applications to is ECS. This is due to multiple reasons, including amongst others:

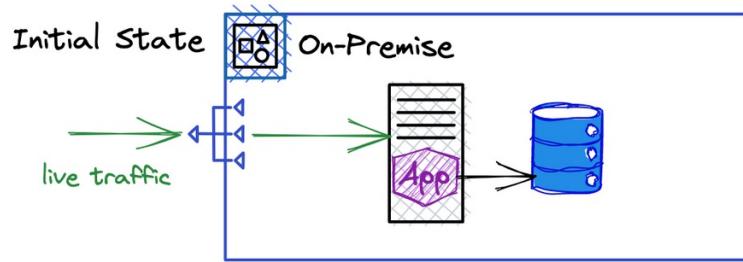
- **Software that already runs on production is here to stay** - if a development process ends up with software being shipped to production so it's used by the company itself, customers, or other third parties, generally, it will be there for a long time. Replacing existing software with new versions or a completely new product is very hard, takes a long time, and introduces risks for companies. That's why it's often only tackled if there's no other option. With the benefits of the cloud, most companies chose to migrate their core to mature and battle-tested cloud services like ECS.
- **On-premise applications are rather monolithic** - software development processes from the previous decades often went through the classic approach. Requirements analysis, planning, developing, testing, and shipping the software. Not for small chunks, but for the whole product. This often resulted in gigantic software products in just a few or one service or application. Containerizing those applications so they can run in ECS is rather simple in comparison to restructuring their whole architecture to go fully Serverless with Lambda.
- **Containerized applications are viewed as cloud-independent** - migrating a container-based application from any cloud container orchestration service to another cloud provider is straightforward. This is due to the fact that there's at least one comparable service in each large cloud provider that offers the same or nearly the same feature set. Being as independent as possible from a single cloud provider is often an important argument for stakeholders like investors. A Serverless, event-driven architecture that is powered by Lambda, SQS, and multiple other natively-connectable services is more difficult to move to another cloud provider.

Proficient knowledge of how to lift applications to AWS without much or any interruption is one of the core skills for getting highly-paid jobs around the world.

But how to do this? There's no single answer, but many. We'll quickly dive into one migration that is not only theoretical but one that was actually executed in the real world.

## A Story of Moving to the Cloud and Switching Database Solutions

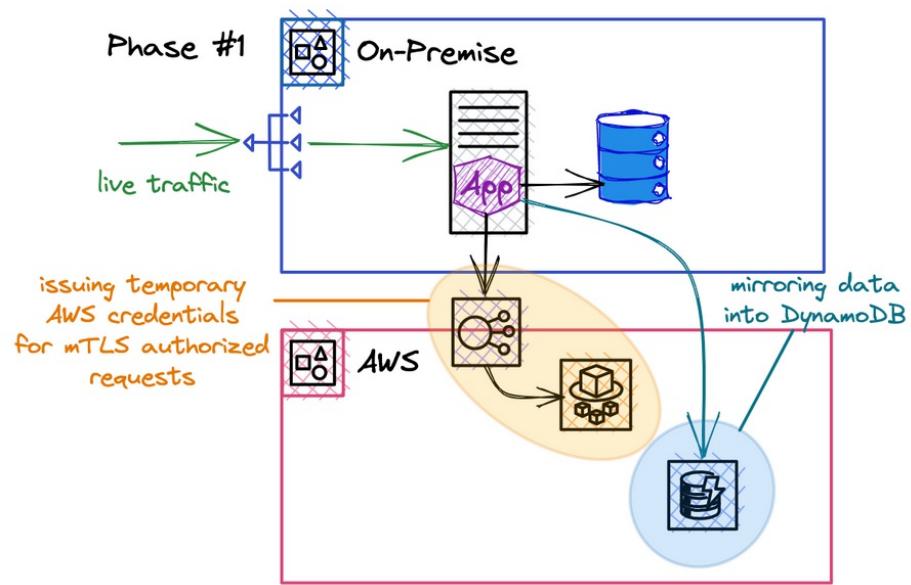
The initial state of the application was simple: a Java-based monolithic application running on on-premise servers with a directory as the main data storage. The application was already running on production and the requirement was a migration with as little downtime as possible.



Our migration plan to AWS was to slowly lift the application in multiple steps without causing any downtime at all:

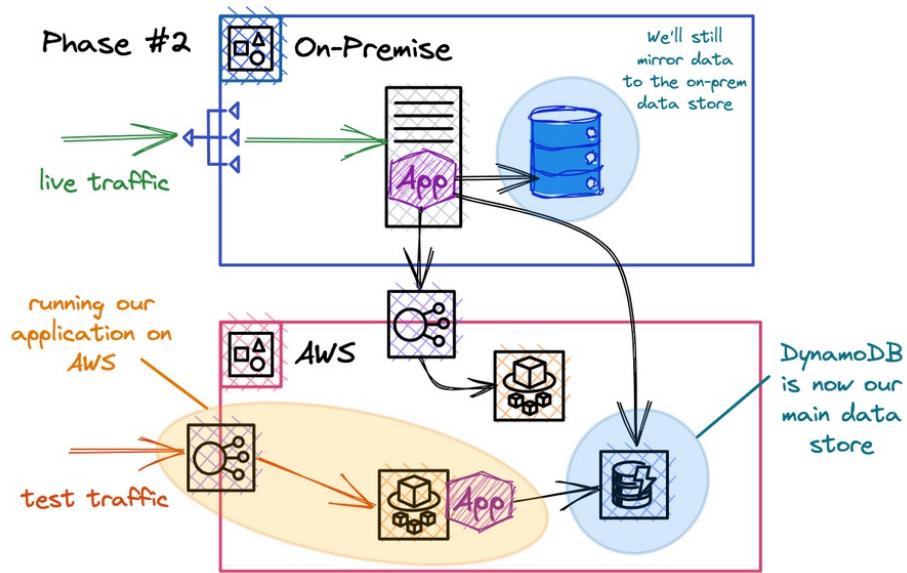
1. Creating an on-the-fly copy of all the on-premise data in DynamoDB within the application.
2. Containerizing the application and running it on a new ECS cluster. Besides that, we're now using DynamoDB as the primary data storage and only mirror data to the on-premise directory.
3. Completely removing the on-premise directory. Running a lot of integration testing on the application that now runs on ECS and is able to access the live data.
4. Switching the live traffic to the ECS cluster and getting rid of the on-premise application.

Let's explore the steps in a little bit more detail.



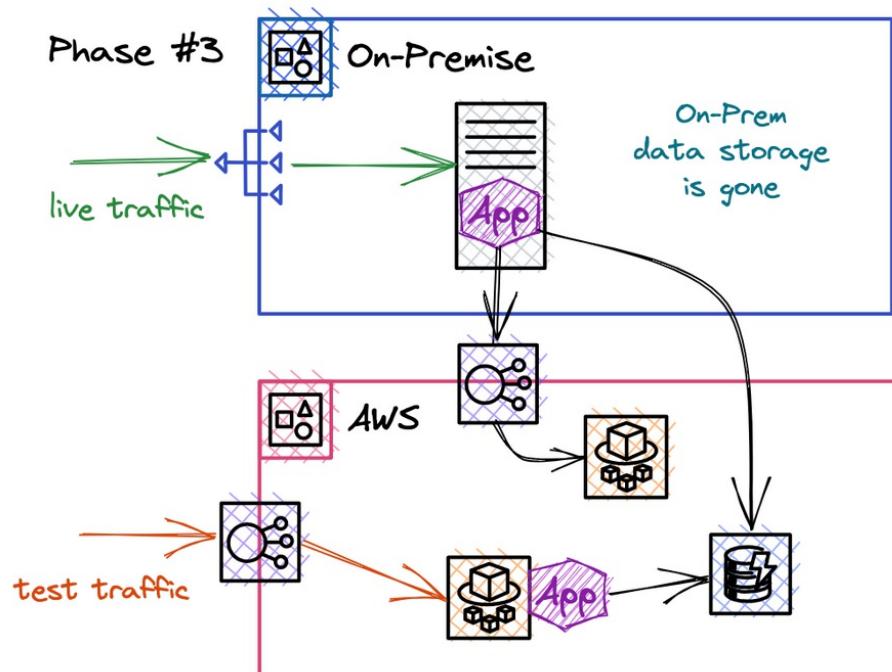
Within the first phase, we needed the option to retrieve temporary AWS credentials to securely write to DynamoDB. We solved this by exposing a Fargate task that supports mutual authentication with a certificate. For a valid request, it returns temporary AWS credentials for a role that is only able to write to our DynamoDB tables.

We then abstracted our data to layer in the application so we can not only use one but as many as we want. One layer is always the root data layer, returning the source of truth. The secondary data layers will receive a copy that is always forcefully overwritten. When we read and write data, we check if the data is in sync and log issues that may be observed. This took quite a few iterations as we're not moving from NoSQL to NoSQL but also switching database solutions from a directory to DynamoDB.

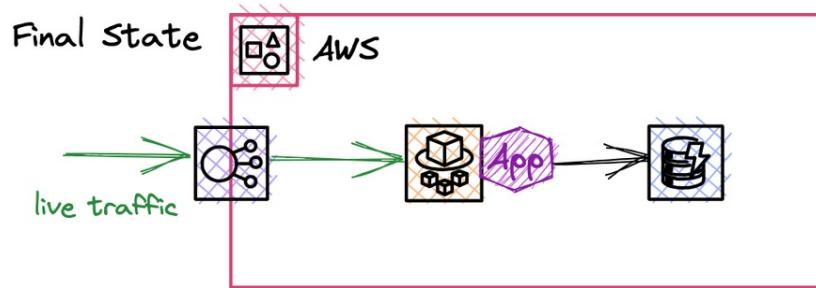


The on-premise application now writes data to on-premise and DynamoDB. Now we've containerized the application and built an ECS cluster for our service which is accessing DynamoDB. Now we have a copy of our application that already runs on AWS and can be tested on real data without affecting customers.

We've also switched the order of the data layers. DynamoDB is now the source of truth and data only gets copied to the on-premise storage.



At the next stage, we've finally got rid of the on-premise data storage. We can now focus on testing the cloud application with integration tests, load tests, and disaster recovery. Really making sure that everything works as smoothly as possible.



Lastly, we only had to switch the DNS records to our cluster on AWS. A few days before we reduced the Time-To-Live for the main record so clients wouldn't cache the record for more than 60 seconds.

There was no outage, not even for one second and nobody even noticed this switch. If anything went crazy, there was always the option to change the DNS records again to the on-premise cluster and fix any issues.

### Tips & Tricks for the Real World

ESC is fabulous and we're enjoying working with it every day. When used with Fargate, a fundamentally well-structured setup often results in worry-less operations. If it runs, it runs.

Nevertheless, let's quickly go through some important points that should be kept in mind to achieve such a result:

- **Use a task role with appropriate permissions to make use of other AWS services** - the magic of the cloud environment is its diversity of service offerings. ECS alone is very powerful, but the integration with other services goes beyond all possibilities.
- **Use the latest ECS container agent version** - when using ECS with EC2, make sure that your instances are using the latest container agent to take advantage of new features and improvements.
- **Automatically expire images in ECR** - when you're regularly updating your images in ECR and you're using immutable tags (a tag that can't be overwritten), you can assign retention policies. These policies define which images should be kept and which should be automatically deleted. You can for example only keep the latest 10 images or expired images that are older than one month.

- **Put environment specifics in environment variables and keep your application generic** - don't hardcore environment-specific strings into your application, but pass it as environment variables. By that, you can use the exact same Docker image for every stage. The only thing that changes are the passed environment variables.
- **You can safely inject secrets from the AWS Secrets Manager into your container** - you don't need to put secrets in plain text into environment variables. You can directly reference secrets via ARNs. Don't forget to assign the proper IAM permissions for your secrets for the action `secretsmanager:GetSecretValue` to your task execution role. Otherwise, ECS can't read the secrets and pass them to your task's container.
- **Think deeply about how to properly organize your auto-scaling policies** - setting up proper auto-scaling rules is not a trivial task for multiple reasons. When you're using scale-up or scale-down actions with CloudWatch (based on metrics like CPU usage) when certain thresholds are met, temporary peaks like start-up times have to be included in your calculations. Else, you'll end up starting too many containers as each new container will increase the average CPU usage at first.
- **Really take advantage of the target group's health monitoring** - the health checks provide a great option to deeply monitor the container's health. If new tasks spawn, they won't get into the load balancer's routing until the checks pass. Use this to really ensure that a container has started properly and it's ready to receive requests and serve them with the expected low latencies. As done in our example, instead of using TCP connection checks you can make HTTP calls to the application itself. You decide when to return a successful HTTP code.

## Final Words

You'll encounter ECS in almost any organization due to its mature service state and reliability for core services. Even when focusing on Function-as-a-Service technologies like Lambda and event-driven architectures, knowing the basics of a container orchestration service is a fundamental requirement for almost any engineering role.

We love ECS and we hope you'll too.



**AWS Lambda**

# Using Lambda to Run Code without Worrying about Infrastructure

## Introduction

Amazon EC2 enables you to leave managing physical servers behind and just focus on virtual machines. With Lambda, launched back in 2014, AWS took this one step further by completely removing customers' liabilities for the underlying infrastructure. The only thing you bring is the actual code you want to run and AWS takes care of provisioning the underlying servers and containers to execute it.

### **Lambda Abstracts Away Infrastructure Management, but It Doesn't Come without Trade-Offs**

If you've never worked with Lambda before, this is maybe the most important chapter as the included information doesn't seem to be very intuitive in the first place. Let's have a look at how Lambda works under the hood and which trade-offs we have to face due to its on-demand provisioning of infrastructure. Also, let's see which measures we can use to slightly mitigate the limitations we face.

#### **Micro-Containers in the Back Which Run Your Code**

One thing that's often missed or misunderstood is: Serverless doesn't mean that there are no servers. These are just abstracted away and intransparent for the application developer.

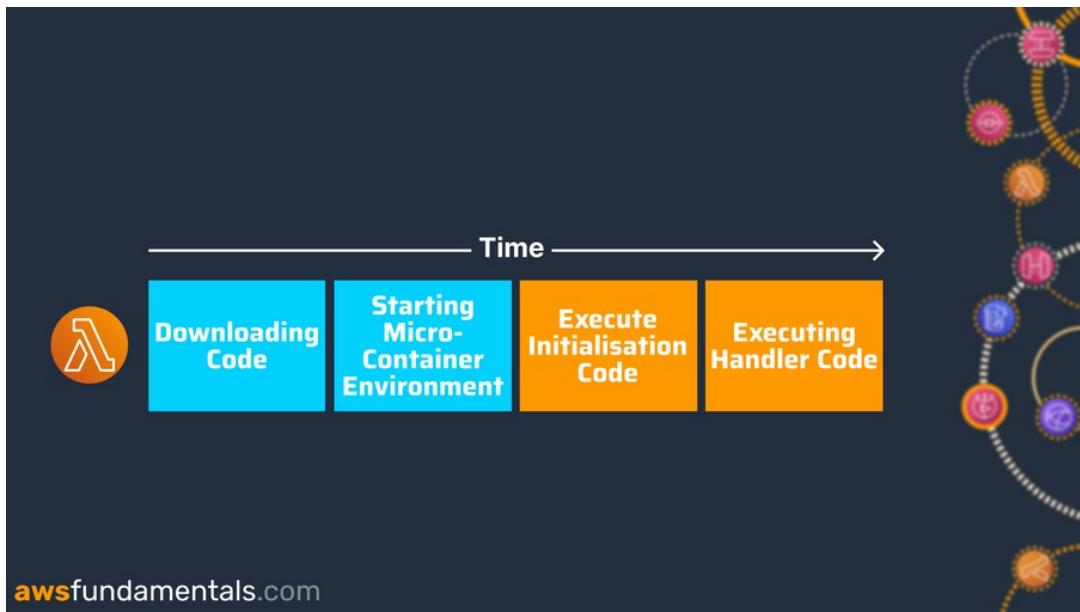
For an incoming request to your Lambda function, AWS will either

1. internally provision a micro-container and deploy your code into it or
2. re-use an existing container that hasn't been de-provisioned yet and is not already busy processing another request.

As you've probably already guessed, the first option comes with a trade-off as resources have to be assigned on demand which takes a noticeable amount of time.

#### **Assigning Resources on-Demand Takes Time - What's Happening in a Cold Start**

Let's take a deeper look at the first scenario. We're not surprised that there's a certain amount of bootstrapping time needed until our code is executed. The process of preparing Lambda's environment so it is able to execute your code is called **cold start**.



Lambda needs to download your function's code and start a new micro-container environment that will then receive and execute it. Afterward, the global code of your function will run. This is everything that's outside of your handler function. This globally scoped code and its variables will be kept in memory for the time that this micro-container environment is not de-provisioned by AWS. See this as some very volatile cache.

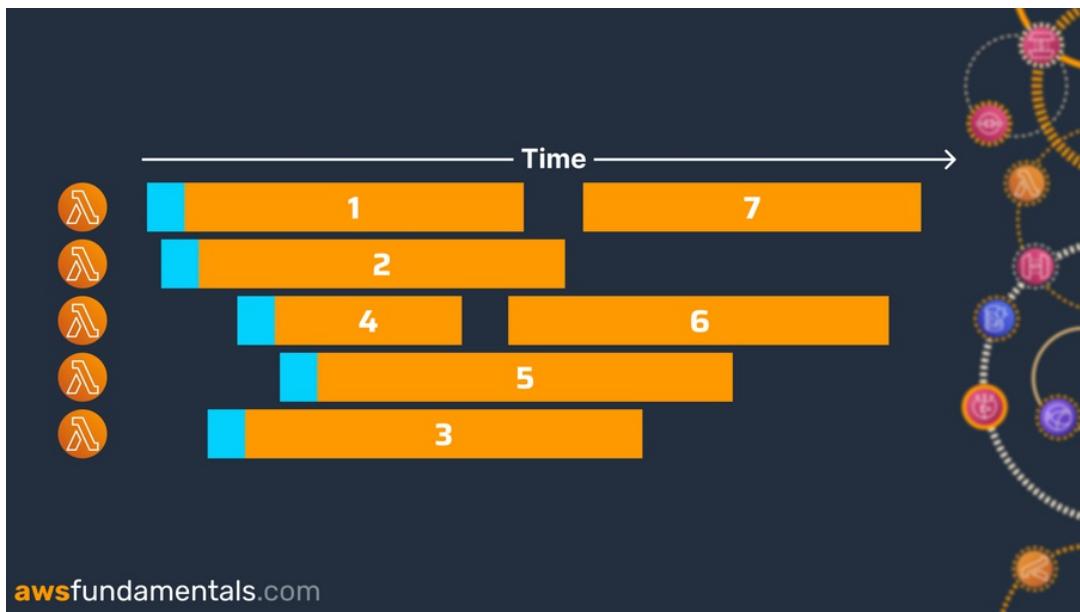
Lastly, your main code is run that's inside your handler function.

**Worth noticing:** Even though it's part of the launch phase until your target code finally runs, the global initialization code of your function is not an official part of the cold start.

If we compare this to traditional container approaches, e.g. running Fargate tasks with ECS, we'll see a difference in the average response times. This is especially noticeable when we focus on the slowest 5% of requests, as they will be slower in Lambda than on Fargate, as the cold starts will immensely contribute to those.

#### A Micro-Container Is Only Able to Serve One Request at a Time

Each provisioned Lambda micro-container is only able to process one request at a time, which means: even if there are multiple execution environments already provisioned for a single Lambda, **there can be another cold start** if all of them are currently busy handling requests.



Looking at the invocation scenario above you can see that five Lambda micro-containers were initially started in the first phase. This was due to the fact, that each consecutive request came in before another container has finished.

The first re-use did only happen at request number six, as micro-container number three (counting top-down) has finished its previous request.

This increases the difficulty of reducing cold starts, especially if your application landscape is built via many different Lambda functions, maybe even requiring mutual synchronous invocations.

#### **Global Code Is Kept in Memory and Is Execute with High Memory and Compute Resources**

If we're looking at a sample handler function, we can see that it's possible to run code outside of the handler method - the so-called **bootstrap code**.

```
bootstrapCoreFramework();
const startTime = new Date();

exports.handler = async (event) => {
    // [...]
    executeWorkload();
}
```

The results of this code execution can result in **global variables that are kept in memory**,

so they continue to exist over **several executions** of this single micro-container. It's only lost after the tear-down of the Lambda environment was executed.

In our example, we'd keep the results of the bootstrap of our core framework and the start time in memory. Those will only vanish when our function's container will be de-provisioned by AWS.

This is not the only great thing about the global scope. AWS executes the code outside of the handler method with a high memory (and therefore with that high vCPUs) configuration, regardless of what you've configured for your function. And even better: the first 10 seconds of the execution of the globally scoped code is **not charged**. This is not some shady trick, but actually, a well-known feature to adopt the usage of Lambda.

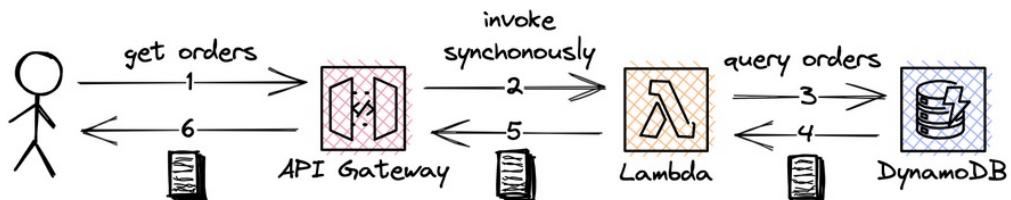
Make use of this and bootstrap as much as possible outside the handler function and keep a global context while your function is running.

A small reminder: Regularly invoking your function via warm-up requests. This will increase the time your global context is kept as the container lifetime is increased. **But it's still limited.** AWS will tear down your function's environment after a certain period of time, even if your function is invoked all the time.

### Your Functions Can Be Invoked Synchronously and Asynchronously

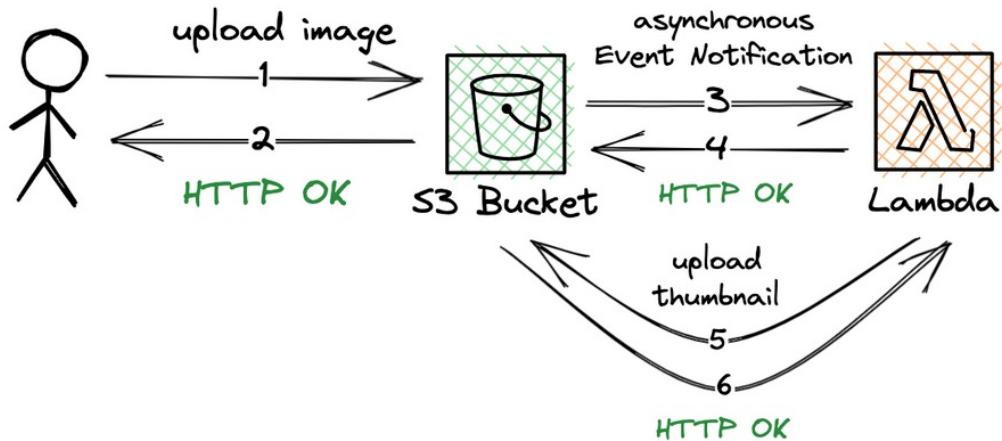
There are two methods to invoke your functions code:

- **synchronous or blocking:** Lambda executes your code but only returns after the execution has finished. You'll receive the actual response that is returned by the function. An example would be a simple HTTP API that is built via API Gateway, Lambda, and DynamoDB. The browser request will hit the API Gateway which will synchronously invoke the Lambda function. The Lambda function will query and return the item from DynamoDB. Only after that, the API Gateway will return the result.



- **asynchronous:** Lambda triggers the execution of your code but immediately returns. You'll receive a message about the successful (or unsuccessful, e.g. due to permission issues) invocation of your function. An example would be a system that generates

thumbnails via S3 and Lambda. After a user has uploaded an image to S3, they will immediately receive a success message. S3 will then asynchronously send an event notification to the Lambda function with the metadata of the newly created object. Only then Lambda will take care of the thumbnail generation.



If you're invoking functions from another place, e.g. another Lambda function, the invocation type depends on how you want to handle results. Synchronous invocation is useful when you need to retrieve the result of the function execution immediately and use it in your application.

```

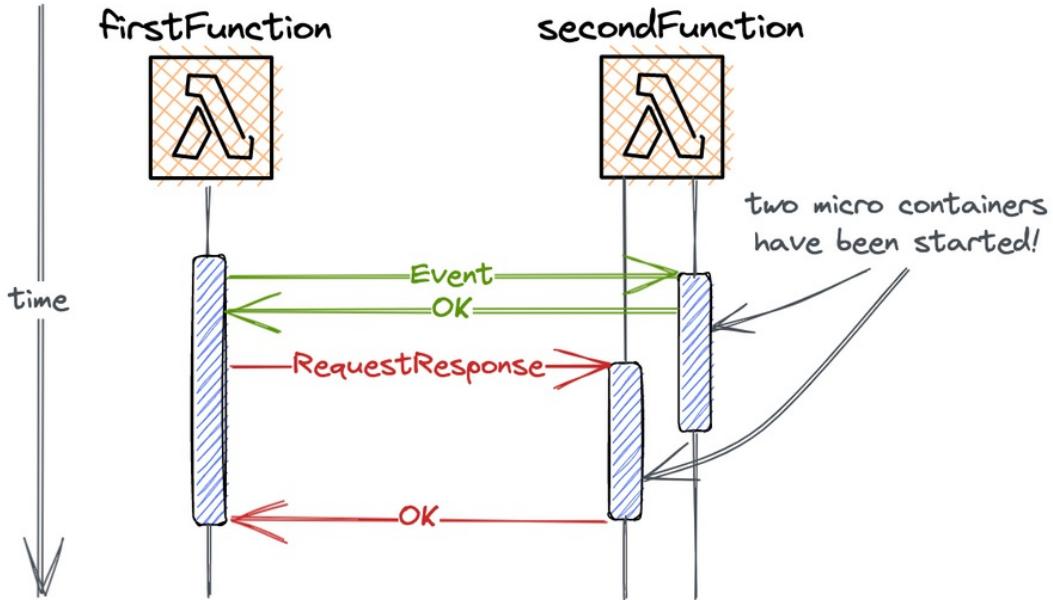
const AWS = require('aws-sdk');
const lambda = new AWS.Lambda();

exports.handler = async (event) => {
    // returns immediately
    await lambda.invoke({
        FunctionName: 'secondFunction',
        InvocationType: 'Event',
        Payload: JSON.stringify({ message: 'Hello, World!' })
    }).promise();

    // returns after 'myFunction' has finished
    await lambda.invoke({
        FunctionName: 'secondFunction',
        InvocationType: 'RequestResponse',
        Payload: JSON.stringify({ message: 'Hello, World!' })
    }).promise();
}

```

Let's have a look at the example Lambda function `firstFunction` above. Its sole purpose is the invocation of another function which is called `secondFunction`.



If we look at the sequence diagram above for the invocation of function `firstFunction`, we see how both execute. The first invocation will return immediately, even though the computation still runs inside the second function. Before this computation can finish, the second invocation hits and therefore starts another micro-container as the other is still busy. Now, the invocation does not return immediately but waits until the computation has finished.

### What's Necessary to Configure to Run Your Lambda Functions

When creating a Lambda function you need to define many properties of the environment. Many can be changed afterward, but some are fixed and can't be changed once the function is created. Let's explore the most important settings and configurations.

#### Choosing the Lambda Runtime and CPU Architecture

There's support for a lot of runtimes at Lambda, including Node.js, Python, Java, Ruby and Go. Besides deploying your function as a ZIP file, you have to option to provide a Docker container image. It's also possible to bring your own runtime to execute any language by setting the functions runtime to *provided* and either packaging your runtime in your deployment package or putting it into a layer.

You can also configure if you want your functions to be executed by an x86 or ARM/Graviton2 processor. The latter one, introduced in 2021 for AWS Lambda, offers a better price performance. Citing the AWS News Blog: “*Run Your Functions on Arm and **Get Up to 34% Better Price Performance***”.

All of the environment and CPU architecture settings can't be changed without re-creating your function.

The different runtimes vary in their cold start times. Scripted languages like Python and Node.js do better than Java currently, but the latest release of AWS Lambda SnapStart could change that drastically, as it will speed up cold starts by an order of magnitude for Java functions.

#### **Finding the Perfect Memory Size Which Also Results in a Corresponding Number of vCPUs**

The memory size of your Lambda function does not only determine the available memory but **also the assigned vCPUs**, meaning that higher settings result in higher computation speeds. You'll be billed for GB seconds, so more memory will result in paying more per executed millisecond.

#### **Timeouts - A Hard Execution Time Limit for Your Function**

A single Lambda execution can't run forever. It's up to you to define a timeout of up to 15 minutes. If an execution hits this limit it will be forcefully terminated, interrupting whatever workload it is executing right now. The function will return an error to the invoking service if it was synchronously (blocking) invoked.

#### **Execution Roles & Permissions - Attaching Permissions to Run Your Functions**

Lambda's execution role will determine the permissions it receives on the execution level.

This role will be set when you create your function: either an existing one or a new one. The execution role is important as it determines all permissions that your Lambda function has while it is running. If your function needs to access an Amazon S3 bucket or write logs to Amazon CloudWatch, the execution role must have the appropriate permissions.

As with other services, it is a good security practice to create an execution role with the **least privilege**. This means it should only have the permissions that are required for the function to perform its intended tasks. This helps to reduce the risk of unintended access to resources and data.

## Environment Variables For Passing Configurations To Your Functions

Environment variables are key-value pairs that are passed to your Lambda function. Besides your custom variables, you'll find some reserved ones which are available in every function. Those include, among others:

- `AWS_REGION`- the region where your function resides.
- `X_AMZN_TRACE_ID` - the X-Ray tracing header.
- `AWS_LAMBDA_FUNCTION_VERSION` - the version of the function being executed.

As the name already suggests, environment variables are perfect to configure your function for a specific environment. You don't need to hardcode stage-specific variables into the function, but see the function as a blueprint and pass your configuration via the environment.

### Edit environment variables

**Environment variables**

You can define environment variables as key-value pairs that are accessible from your function code. These are useful to store configuration settings without the need to change function code. [Learn more](#)

Key	Value	
<input type="text" value="ENVIRONMENT"/>	<input type="text" value="development"/>	<button>Remove</button>
<input type="text" value="APP_PREFIX"/>	<input type="text" value="awsfundamentals"/>	<button>Remove</button>

[Add environment variable](#)

▶ [Encryption configuration](#)

[Cancel](#) [Save](#)

Each of your variables is stored in the function's environment and can be accessed from your code. For Node.js you can use the `process.env` object, for Java it will be the `System.getenv()` method, and Python will provide `os.environ`.

## **VPC Integration - Accessing Protected Resources within VPCs and Controlling Network Activity**

There are services that can only be launched inside a VPC, including ElastiCache. If you need to access such a service from Lambda, you'll also need a VPC attachment for Lambda. Other use cases are enhanced security requirements like restricting outbound traffic from your functions.

Also, running your functions within a VPC will give you greater control over the network environment. If you have functions that do not need internet access, you can put them into a private subnet. This will restrict them from making outgoing calls to the internet which will immensely increase security.

But there are considerations when using VPCs. Even though AWS improved this drastically with the integration of AWS Hyperplane, a VPC integration will increase your function's cold start times. Additionally, VPC integration can increase the costs as you'll be charged for data transfer to other resources in your VPC.

## **Natively Invoke Lambda via Different AWS Services via Triggers**

Lambda is natively integrated with a lot of other services via triggers, meaning you're able to launch Lambda functions based on events that are fired from other services.

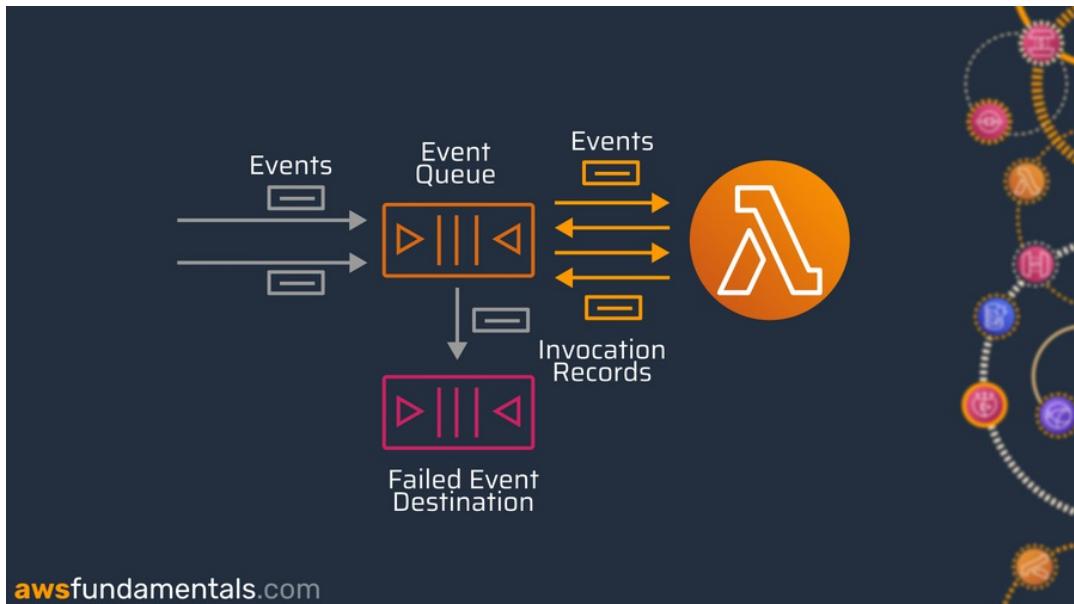
Prominent examples are:

- Integration with API Gateway to respond to HTTP requests .
- Lifecycle events at S3, e.g. launching a Lambda function if an object was created in a specific path of your bucket.
- Consuming events from an SQS queue.
- Scheduling functions based on EventBridge rules.

Triggers are a major feature to build reliable event-driven architectures that are also able to recover in case of outages and errors.

## Triggering Follow-Ups for Successful or Unsuccessful Invocations of Functions via Destinations

The upside of not having to wait for responses on asynchronous invocation is also the downside: you can't immediately decide if the execution didn't result in any errors. That's why Lambda offers destinations so you can react to successful or faulty executions.



In our example, failed invocations or invocations that can't be processed are forwarded to an SQS Dead-Letter-Queue. Later on, this queue can be used to investigate events that failed and find out the reason for the failure. Otherwise, we can poll events from the queue from another function to trigger a reprocessing at a later point in time.

## Code Signing to Ensure the Integrity of Deployment Packages

Your Lambda functions are executed on hardened systems, but how do you ensure your code was never tampered with? With AWS Signer and code signing, you can create signing profiles to enforce that only code by trusted publishers can be deployed to your functions.



### Using Unique Pointers to Functions via Aliases and Versioning

You can have different versions of your function in parallel. Maybe you want to test some code changes without affecting the currently stable version on your staging environment.

When publishing a version you'll get another version number which can be used to invoke your function via the qualified ARN:

```
arn:aws:lambda:us-east-1:012345678901:function:myfunction:17
```

Additionally, you can create an alias for each of your versions. An alias acts as a pointer to your function. The benefit of using aliases instead of the qualified ARNs is that you can use them with event source mappings without having to adapt each of the mappings after you've published a new version. You'll only need to update a single resource: the alias itself.

### Reserved and Provisioned Concurrency to Guarantee Capacities and Reduce Cold Starts

There are two different features that help you to manage the performance and scalability of your functions further than just assigning higher memory settings: **reserved** and **provisioned concurrency**.

Both can help you improve the performance and scalability of your functions, but they are used for different purposes. Reserved concurrency is used to ensure that a certain number of instances of your function are always available to handle requests, while provisioned

concurrency is used to keep instances pre-warmed in anticipation of traffic.

### **Reserved Concurrency for Guaranteeing a Functions Concurrency Capacity**

The default concurrent execution Lambda for an account is 1000. This means it's not possible to have more than 1000 Lambda functions executed in parallel. This also implies that it's possible to run a huge number of functions in parallel, maybe by accident due to recursion with missing exit conditions or similar errors.

For restricting the maximum number of parallel execution you can use reserved concurrency for your function. Each reserved concurrency will be subtracted from your account limits so that AWS can guarantee that this scale of parallel executions is always possible for these specific functions. It also ensures that there's never a chance to run more than that number in parallel.

If a function didn't declare a value for reserved concurrency, it will use the unreserved concurrency capacity that is left in your account which could probably be completely consumed under certain circumstances.

### **Provisioned Concurrency for Reducing Cold Starts**

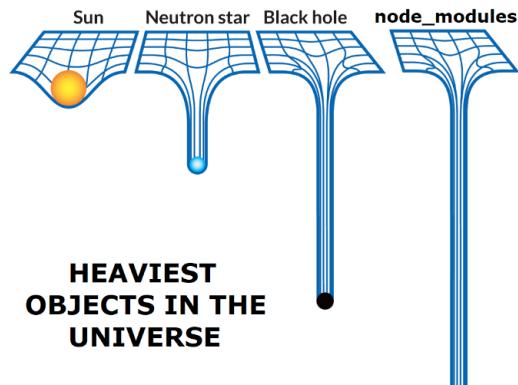
Regardless of the strategies you're using of keeping Lambda functions warm via strategic health checks or your level of permanent requests per second, your micro-containers **will be de-provisioned** at some time.

The only way to get around this is to use provisioned capacity. AWS will keep a certain number of Lambda environments provisioned so they are always ready for execution for incoming requests.

This comes with significantly higher pricing and also increased times for deployments (up from a matter of seconds to a few minutes).

### **Layers Enable You to Externalize Your Dependencies**

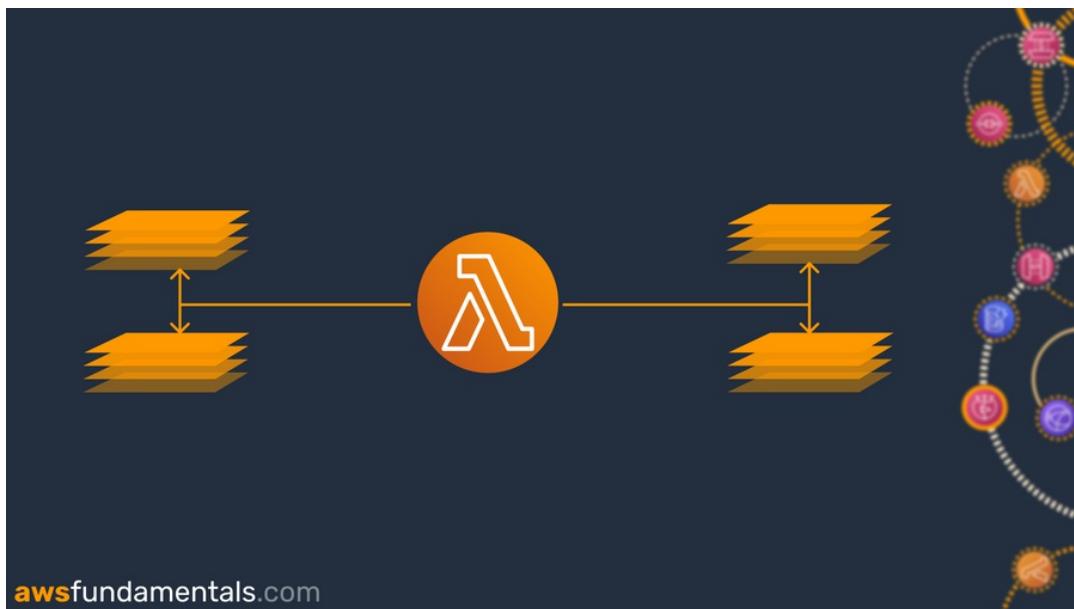
Building extensive business logic often doesn't require you to reinvent the wheel, but to make use of existing libraries. This mostly results in having more code in dependencies than in actual self-implemented business logic which will slow down packaging and deployments.



Also, you'll need to package dependencies for all of your Lambda functions individually as they need to be included in the deployment package - even in the case that most of your function do rely on the same packages.

The solution for this is Lambda Layers. You can create a versioned Layer including the dependencies you need for your Lambda function. Afterward, you can attach one or several functions to the same layer. All of them will get access to the included dependencies.

New function deployments will only require you to package your own code which will drastically increase packaging and deployment times, as you likely only have a few kBytes of code left.



There's a deployment package size limitation, which **includes the size of referenced layers** of 50 MB for zipped files and direct upload and 250 MB for the unzipped archive.

**Make sure you package your dependencies in the right folder.** For example, lambda expects your `node_modules` to be inside the top-level folder `nodejs`.

## Monitoring Your Functions with CloudWatch to Detect Issues

As with other services, Lambda integrates with CloudWatch by default and submits a lot of useful metrics without any further configurations. CloudWatch also automatically creates monitoring graphs for any of these metrics to visualize your usage.

The default metrics include:

- **Invocations** – The number of times that the function was invoked.
- **Duration** – The average, minimum, and maximum amount of time your function code spends processing an event.
- **Error count and success rate (%)** – The number of errors and the percentage of invocations that were completed without error.
- **Throttles** – The number of times that an invocation failed due to concurrency limits.
- **IteratorAge** – For stream event sources, the age of the last item in the batch when Lambda received it and invoked the function.
- **Async delivery failures** – The number of errors that occurred when Lambda attempted to write to a destination or dead-letter queue.
- **Concurrent executions** – The number of function instances that are processing events.

Any log messages you write to the console can also be submitted to and ingested by CloudWatch if your Lambda's execution role has sufficient permissions.

- `logs:CreateLogGroup`
- `logs:CreateLogStream`
- `logs:PutLogEvents`

The first permission is only necessary if you don't create the log group yourself. If you create one yourself, you can easily define a retention policy so that log messages expire after a defined period of time. This helps to avoid unnecessary costs for logs that are not in use anymore.

## Going into Practice - Creating Our First Serverless Project

We've gone through the most important fundamentals of Lambda. Let's jump into the doing and create our first, own small Lambda project.

We'll divide this into a journey of four major steps:

1. **Creating a simple Node.js function.** We'll create a small function, and adapt and deploy code changes within the AWS management console. We'll also test our function here via our own test events.
2. **Adding external dependencies.** We'll add Axios as a dependency so we can execute HTTP calls in a more convenient way.
3. **Externalizing dependencies into a Lambda Layer.** Dependencies update rather rarely compared to our own code. Let's extract our new dependency into a Lambda Layer so we don't need to package and deploy them for each code update.
4. **Invoking another Lambda function.** Let's create another function that we can invoke from our initial function to see the differences between synchronous and asynchronous invocations.

### Creating a Simple Node.js Function

Jump into the AWS Lambda console and click on `Create function`. We only need to define a name, select our target architecture, and chose which runtime we want to use. For our example, we'll go with Node.js.

## Create function Info

AWS Serverless Application Repository applications have moved to [Create application](#).

### Author from scratch

Start with a simple Hello World example.

### Use a blueprint

Build a Lambda application from sample code and configuration presets for common use cases.

### Container image

Select a container image to deploy for your function.

### Basic information

#### Function name

Enter a name that describes the purpose of your function.

awsfundamentals

Use only letters, numbers, hyphens, or underscores with no spaces.

#### Runtime Info

Choose the language to use to write your function. Note that the console code editor supports only Node.js, Python, and Ruby.

Node.js 18.x

#### Architecture Info

Choose the instruction set architecture you want for your function code.

x86\_64

arm64

#### Permissions Info

By default, Lambda will create an execution role with permissions to upload logs to Amazon CloudWatch Logs. You can customize this default role later when adding triggers.

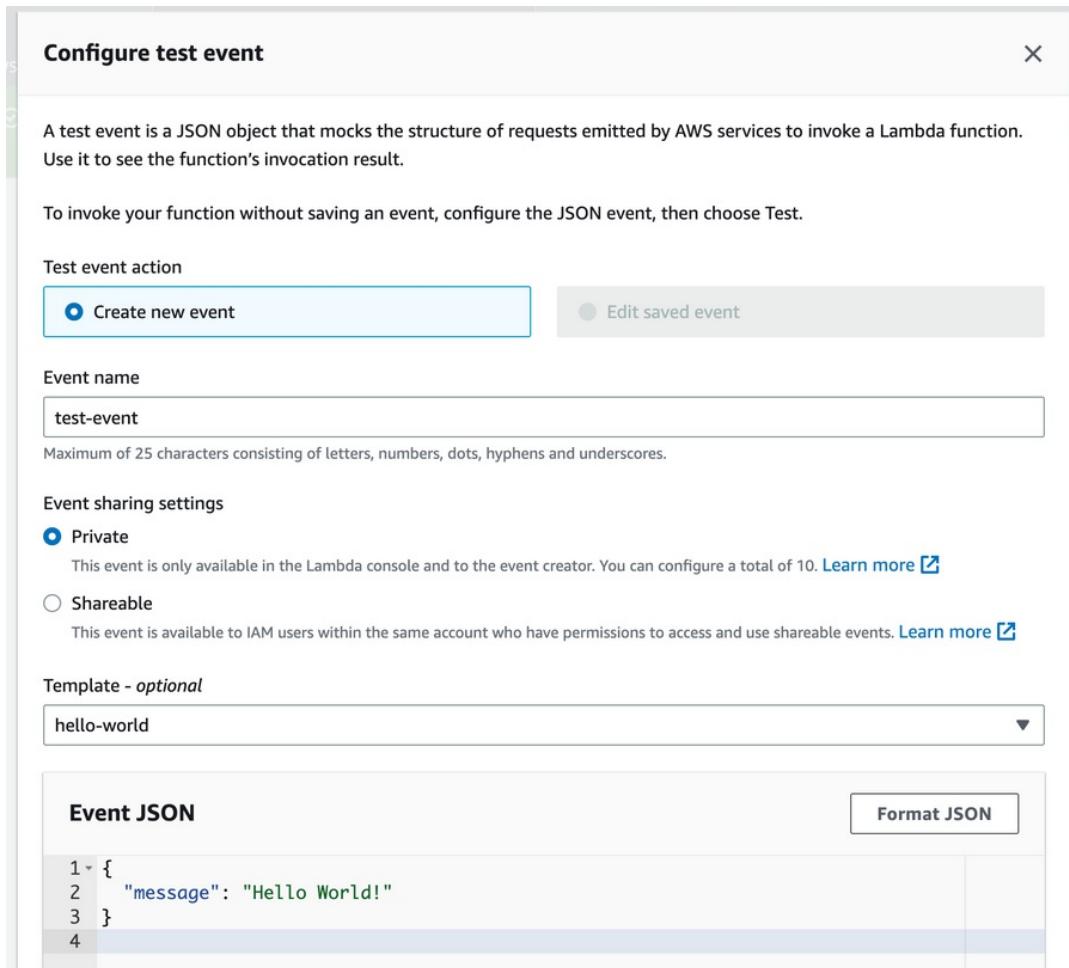
▶ Change default execution role

After clicking create, you'll be taken to your functions overview. You'll immediately notice the live editor for our function's code. This editor can also be used to test our function and to save and deploy updates to our function.

The screenshot shows the AWS Lambda Code source editor. At the top, there are tabs for 'Code source' (selected) and 'Info'. On the right, there are buttons for 'Upload from' and a dropdown menu. Below the tabs is a toolbar with icons for File, Edit, Find, View, Go, Tools, Window, Test, Deploy, and settings. The main area has a search bar labeled 'Go to Anything (% P)' and a sidebar labeled 'Environment' showing a folder named 'awsfundamentals' containing an 'index.mjs' file. The code editor window displays the following code:

```
1 export const handler = async(event) => {
2     // TODO implement
3     const response = {
4         statusCode: 200,
5         body: JSON.stringify('Hello from Lambda!'),
6     };
7     return response;
8 };
9 
```

Let's do exactly that by clicking on `Test`. This will open a modal where we can create our first test event. Let's pass a JSON object with a field `message` to our function.



Now let's adapt our function and return the message we pass in the function's response. After clicking `Deploy` the changes will be deployed to our function. Now we can invoke our function with the test event.

The screenshot shows the AWS Lambda code editor interface. The left sidebar shows an environment named "awsfundamentals". The main area displays the "index.mjs" file with the following code:

```

1 export const handler = async(event) => {
2     const { message } = event;
3     // TODO implement
4     const response = {
5         statusCode: 200,
6         body: JSON.stringify(`Our message: ${message}`),
7     };
8     return response;
9 };

```

We'll get what we expect: our message!

The screenshot shows the AWS Lambda execution results page. It displays the "Execution result" for a test event named "test-event". The response is:

```

{
    "statusCode": 200,
    "body": "\"Our message: Hello World!\""
}

```

Function Logs:

```

START RequestId: 68edb45f-6d7c-4e6b-be32-5bf0694b8de7 Version: $LATEST
END RequestId: 68edb45f-6d7c-4e6b-be32-5bf0694b8de7
REPORT RequestId: 68edb45f-6d7c-4e6b-be32-5bf0694b8de7 Duration: 8.20 ms

```

Request ID: 68edb45f-6d7c-4e6b-be32-5bf0694b8de7

That's it already for the first simple task: you've created, run, updated, and successfully invoked a Lambda function.

### Adding External Dependencies

The first challenge was solved. Let's continue and add some external dependencies. We'll initialize a new project via `npm` and install Axios. We'll also add a file for our function's code.

```

mkdir awsfundamentals && cd awsfundamentals
npm init && npm i axios
touch index.js

```

Let's rewrite our existing code a little bit by adding a new HTTP call via Axios to

<https://ipinfo.io/ip> to get the function's external IP address.

```
const { create } = require("axios");

exports.handler = async () => {
    // create a new axios instance
    const instance = create({
        baseURL: "https://ipinfo.io/ip",
    });
    // make a GET request to retrieve our IP
    const { data: ipAddress } = await instance.get();
    return {
        statusCode: 200,
        body: `The Lambda function's IP is '${ipAddress}'`,
    };
};
```

Now we only need to bundle our code together with our `node_modules` folder into a ZIP archive which we can then upload to our Lambda function via `Upload from > .zip file`.

```
zip dist.zip .
```

After uploading it we can execute it again via the test button. The response should look something like this:

```
Response
{
    "statusCode": 200,
    "body": "The Lambda function's IP is '54.77.191.130'"
}

Function Logs
START RequestId: f5e50828-82c2-49a0-96a8-09d343aae66a Version: $LATEST
END RequestId: f5e50828-82c2-49a0-96a8-09d343aae66a
REPORT RequestId: f5e50828-82c2-49a0-96a8-09d343aae66a Duration: 572.21
ms
Billed Duration: 573 ms Memory Size: 128 MB Max Memory Used: 74 MB Init
Duration: 266.41 ms

Request ID
```

```
f5e50828-82c2-49a0-96a8-09d343aae66a
```

While our function is still very small because we're only including Axios, its size can increase quickly. After a certain threshold, it's not possible to view or edit the code in the Lambda console anymore. We'll also upload a lot of kilobytes for every function upload, even though there are often only changes to our code.

Let's fix that in our next part by extracting our dependencies into a Lambda Layer.

### Externalizing Dependencies into a Layer

Creating a Lambda Layer does improve two major things:

- we'll reduce the size of the deployment unit we need to upload on function updates
- we can share a layer with several Lambda functions which may all have the same dependencies

The process is quick and simple. We only need to include our dependencies for our layer's zip file in an expected directory format: in the Node.js case, the `node_modules` have to reside in the root folder `nodejs`.

```
mkdir -p nodejs
cp -r node_modules nodejs
zip layer.zip nodejs
```

Let's go back to the Lambda console and click on `Layers > Create layer`.

**Layer configuration**

Name  
awsfundamentals

Description - *optional*  
A layer to externalize dependencies

Upload a .zip file  
 Upload a file from Amazon S3

**Upload**

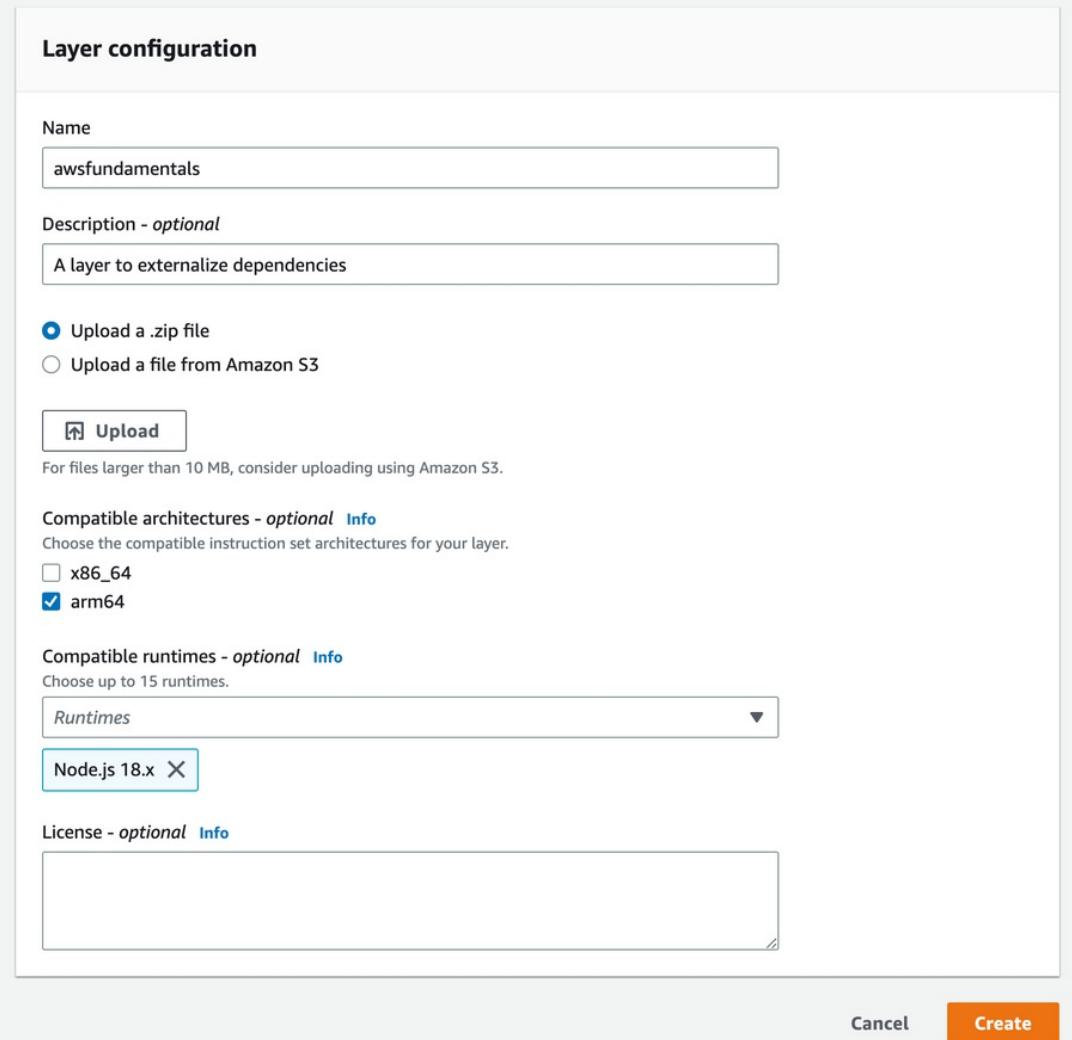
For files larger than 10 MB, consider uploading using Amazon S3.

Compatible architectures - *optional* [Info](#)  
Choose the compatible instruction set architectures for your layer.  
 x86\_64  
 arm64

Compatible runtimes - *optional* [Info](#)  
Choose up to 15 runtimes.  
Runtimes ▾  
Node.js 18.x X

License - *optional* [Info](#)

**Create**



After clicking the `Upload` button, selecting our created `layers.zip`, and finally creating the layer via `Create`, our Lambda Layer is ready to use!

Let's head back to our function and scroll down to the layers overview to connect our new layer view `Add a layer`.

## Add layer

**Function runtime settings**

Runtime Node.js 18.x	Architecture arm64
-------------------------	-----------------------

**Choose a layer**

Layer source [Info](#)  
Choose from layers with a compatible runtime and instruction set architecture or specify the Amazon Resource Name (ARN) of a layer version. You can also [create a new layer](#).

AWS layers  
Choose a layer from a list of layers provided by AWS.

Custom layers  
Choose a layer from a list of layers created by your AWS account or organization.

Specify an ARN  
Specify a layer by providing the ARN.

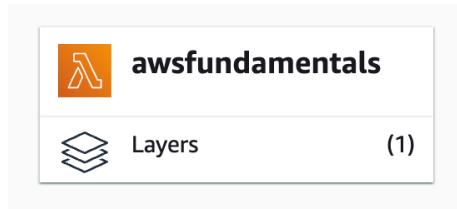
**Custom layers**  
Layers created by your AWS account or organization that are compatible with your function's runtime.

awsfundamentals ▾

Version  
1 ▾

[Cancel](#) [Add](#)

Select custom layers and chose our previously created layer. After clicking **Add**, it will take a few seconds to update the function. Afterward, you'll be taken back to your function's overview and see that the layer is attached successfully.



Let's go into the editor of the function and delete the `node_modules` folder via `right click > delete`, as we don't need it anymore. It will be provided via our Lambda Layer.

Let's run our function again to see that it still works:



The third achievement unlocked. Let's take on our final step in this small getting-started journey of AWS Lambda.

### Invoking Another Lambda Function

For the last part, we'll create a second Lambda function that we'll invoke from our initial function. So had back to the functions overview and click on `Create function`.

For invoking our function we need two things:

1. our first function has to have the `lambda:InvokeFunction` permission and
2. the AWS SDK

For the first point, go to the configuration tab of our first function and click on `Permissions`. You'll see the linked execution role. With a click on the link, you'll be taken to IAM where we can edit the policy.

A screenshot of the AWS Lambda function configuration page. At the top, there are tabs for 'Code', 'Test', 'Monitor', 'Configuration' (which is highlighted in blue), 'Aliases', and 'Versions'. On the left, there's a sidebar with options: 'General configuration', 'Triggers', 'Permissions' (which is selected and highlighted in blue), and 'Destinations'. On the right, under the 'Execution role' section, it says 'Role name: awsfundamentals-role-ifd5zmc'.

Let's add another permission for Lambda via the visual editor for `InvokeFunction`. Let's only choose our new function here.

The screenshot shows the AWS IAM Policy editor interface. At the top, there's a header with 'Lambda (1 action)' and 'Clone | Remove' buttons. Below the header, the 'Actions' section is expanded, showing 'Write' and 'InvokeFunction'. Under the 'Resources' section, 'Specific' is selected, and a single ARN is listed: 'arn:aws:lambda:eu-west-1:157088858309:function:awsfundamentals-invoke'. There are 'EDIT' and 'Add ARN' buttons, along with a checkbox for 'Any in this account'. At the bottom, there's a link to 'Request conditions'.

The final JSON policy should now include our new action.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "VisualEditor0",
      "Effect": "Allow",
      "Action": ["lambda:InvokeFunction"],
      "Resource": [
        "arn:aws:lambda:eu-west-1:157088858309:function:awsfundamentals-
        invoke"
      ]
    }
  ]
}
```

Now we want to update our Lambda Layer to include the AWS-SDK. We'll only do this to see how we can update our layer to a new version. Your Lambda environment always comes with the AWS-SDK so you don't need to provide it unless you want to pin it to a specific legacy version.

```
# remove the first version
rm -rf layer.zip
# install the AWS-SDK
npm i aws-sdk
# put the node_modules into the right folder
cp -r node_modules nodejs
# package it again
zip -r layer.zip nodejs
```

If you go to your layer, there's a button `Create version` to upload a new version of the layer. Afterward, we can go back to our initial function and switch to our new version `2`.

The screenshot shows the 'Edit layers' interface. At the top, it displays 'Function runtime settings' with 'Runtime' set to 'Node.js 18.x' and 'Architecture' set to 'arm64'. Below this is a 'Layers' section with a table. The table has columns for 'Merge order', 'Name', and 'Layer version'. There is one row with a merge order of 1, named 'awsfundamentals', and a layer version of 2. A search bar is present next to the layer version. At the bottom right are 'Cancel' and 'Save' buttons.

That's done. Let's go back to our second function and adapt our code a bit.

```
export const handler = async (event) => {
  const { waitingPeriodSeconds = 10 } = event;
  await new Promise((resolve) =>
    setTimeout(resolve, waitingPeriodSeconds * 1000)
  );
  return {
    statusCode: 200,
  };
};
```

We've added some waiting periods until the function returns so we can later clearly see the impact of the different invocation types. By default, we'll wait 10 seconds, but we allow passing the value via the incoming event.

Let's head back to our initial function and add the invocation part. For now, we'll invoke the function synchronously.

```
const Lambda = require('aws-sdk/clients/lambda');

exports.handler = async (event) => {
```

```

const startTime = Date.now();
// Invoke the target function synchronously
await lambda
  .invoke({
    FunctionName: "awsfundamentals-invoke",
    InvocationType: "RequestResponse",
    Payload: JSON.stringify({
      waitingPeriodSeconds: 5,
    }),
  })
  .promise();
const endTime = Date.now();
const elapsedTime = endTime - startTime;
return {
  statusCode: 200,
  body: `Invocation took ${elapsedTime}ms`,
};
}

```

Let's deploy this update and invoke our function.

```

Response
{
  "errorMessage": "Task timed out after 3.01 seconds"
}

Function Logs
START RequestId: 24b0dca3-52b0-4026-afal-73a143df8e09 Version: $LATEST
2022-12-23T07:57:26.010Z 24b0dca3-52b0-4026-afal-73a143df8e09
Task timed out after 3.01 seconds

```

Well, that's not what we expected. But looking at our function's configuration, it does. The default timeout for Lambda is 3 seconds. As we wait for 5 seconds until our second function finishes its execution, both functions will time out. Let's adapt the timeout at [Configuration > General Configuration](#) and set it to 10 seconds.

The screenshot shows the AWS Lambda Configuration page. The top navigation bar includes tabs for Code, Test, Monitor, Configuration (which is selected), Aliases, and Versions. On the left, a sidebar lists General configuration, Triggers, Permissions, Destinations, and Function URL. The main content area displays the General configuration settings:

General configuration			
Description	Memory	Ephemeral storage	
-	1024 MB	512 MB	
Timeout	SnapStart	Info	None
0 min 10 sec			

An 'Edit' button is located in the top right corner of the configuration table.

Afterward, the function is updated, let's retry the invocation.

```
Response
{
  "statusCode": 200,
  "body": "Invocation took 5107ms"
}
```

That's what we expected. Let's switch to the asynchronous function invocation by changing RequestResponse to Event.

```
const Lambda = require('aws-sdk/clients/lambda');

exports.handler = async (event) => {
  const startTime = Date.now();
  // Invoke the target function synchronously
  await lambda
    .invoke({
      FunctionName: "awsfundamentals-invoke",
      InvocationType: "Event",
      Payload: JSON.stringify({
        waitingPeriodSeconds: 5,
      }),
    })
    .promise();
  const endTime = Date.now();
  const elapsedTime = endTime - startTime;
  return {
    statusCode: 200,
    body: `Invocation took ${elapsedTime}ms`,
  };
};
```

After saving our update and deploying the function again, we'll see in the next invocation that the execution time significantly dropped.

```
Response
{
  "statusCode": 200,
  "body": "Invocation took 719ms"
}
```

Now our initial function doesn't wait until the execution of the second one has finished. The execution is faster, but we don't know if the second function finished its execution without errors.

We've covered a lot in this small project, which is a great starting point to continue playing around with Lambda.

## Exposing Your Function to the Internet with Function URLs

The default way of exposing your Lambda function to the internet via HTTP is by creating an API Gateway. Recently, you can also invoke functions directly via Function URLs, which is a convenient way of exposing your function without creating additional infrastructure.

When enabling function URLs, Lambda will automatically create a unique URL endpoint for you, which will be structured like this:

```
https://<url-id>.lambda-url.<region>.on.aws
```

Your function will be protected via AWS IAM by default but you're able to configure the authentication type to `NONE` to allow public, unauthenticated invocations.

CloudWatch also collects function URL metrics like the request count and the number of HTTP 4xx and 5xx response codes.

## Attaching a Shared Network Storage with EFS

Lambda does only come with ephemeral storage: the temporary directory `/tmp`. Everything which is stored here will be kept until your function is de-provisioned. Just until recently, this was limited to 500MB and just then made configurable to up to 10GB, but also with introducing additional costs.

For a durable storage solution, you can either go for Amazon S3 or EFS. The major advantage

of EFS is that it's a typical file system storage and not an object storage - so you can use it like any other directory. You'll need to keep in mind that you'll pay for the EFS storage, the data transferred between Lambda & EFS, and the throughput. You also have to attach your Lambda function to a VPC, as EFS also has a strict VPC requirement. This will increase cold start times.

### **Running Code as Close as Possible to Clients with Lambda@Edge**

With CloudFront, you're able to execute your Lambda functions on the edge. The capabilities are reduced in comparison to traditional Lambda functions, but they still give you a lot of opportunities. We'll get into detail about this in the upcoming chapter about CloudFront - we just wanted to include this here for the reason of completeness.

### **Lambda Is Charged Based on Memory Settings, Execution Times, and Ephemeral Storage**

One of the major differences between a virtual machine and a container-based solution is a new model of pricing: you're only paying for the actual time at which your code is executed.

What you'll find in the documentation and in the pricing charts is the unit of GB-seconds. This means AWS charges you based on the provisioned memory of your function (the GBs, which also imply the number of vCPUs) and the execution time of your functions.

Let's have a look at the free tier limit of 400,000 GB seconds per month and some different configurations:

- 0.5 GB Memory →  $400,000 / 0.5 \text{ GB} \rightarrow 800,000 \text{ GB-seconds} \rightarrow \mathbf{9.2 \text{ days}}$  of execution
- 1.0 GB Memory →  $400,000 / 1.0 \text{ GB} \rightarrow 400,000 \text{ GB-seconds} \rightarrow \mathbf{\sim 4.6 \text{ days}}$  of execution
- 10 GB Memory →  $400,000 / 10 \text{ GB} \rightarrow 40,000 \text{ GB-seconds} \rightarrow \mathbf{\sim 0.5 \text{ days}}$  of execution

Additional charges apply if you increase the ephemeral storage to over 512 MB.

### **Lambda Comes with Hard and Soft Quotas and Limits**

As with every other service, Lambda comes with limitations. Some quotas can be increased via AWS support, but some are fixed and can't be changed unless AWS updates its policies. Let's have a look at the most important ones as it's likely that you'll face them sometime in the future.

- Maximum Concurrency: 1,000 for old accounts; 50 for new accounts

- Storage for uploaded functions: 75 GB
- Function Memory: 128 MB to 10,240 MB
- Function Timeout: 15 minutes
- Function Layers: 5 Layers per Lambda function
- Invocation Payload: 6 MB (synchronous), 256 KB (asynchronous)
- Deployment Package: 50 MB zipped and 250 MB unzipped (this includes the size of all attached layers) & 3 MB for the console editor
- `/tmp` Directory Storage: between 512 MB and 10 GB

AWS is known for regularly updating its quotas so it's worth checking back with the current state at AWS Quotas.

## A Deep Dive into Great Lambda Use Cases

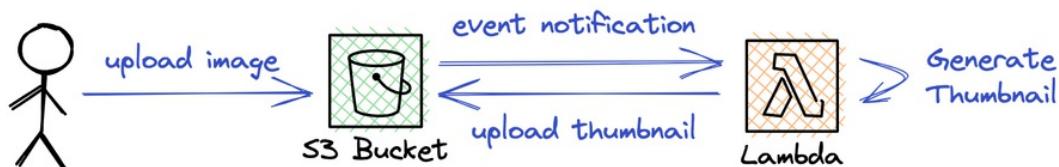
Due to its on-demand pricing, low entry barrier, and ease of use, Lambda is the most flexible service of all AWS offerings. There's almost nothing you can't use Lambda for. The fact that Lambda natively integrates with a lot of other services, often without writing much or any glue code, also majorly contributes to this.

Consider this list as just very few examples as it is possible to go on for weeks or months just writing about the simple or extraordinary use cases for which Lambda is known already.

### Use Case 1: Creating Thumbnails for Images or Videos

Classic approaches to video or image processing involve uploading files to storage and having a server regularly pulling for newly created files.

This will introduce costs as your servers also have idle times if traffic is not constant but with periods of high or low traffic.



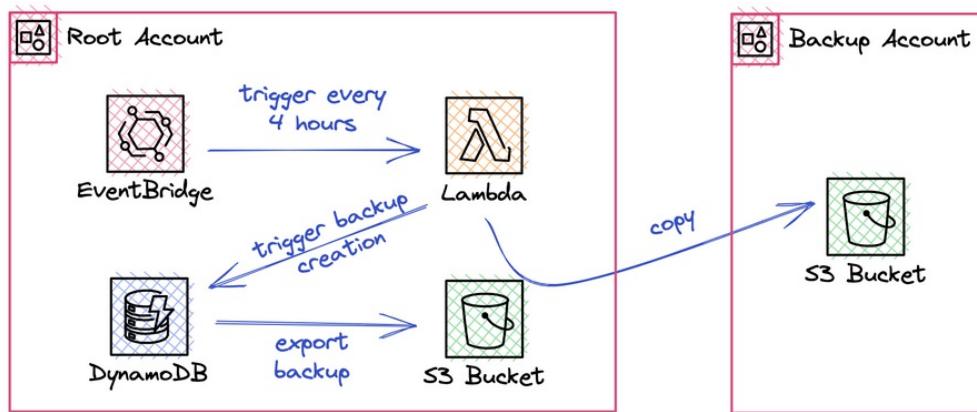
Putting video conversion or thumbnail generation into the hands of Lambda, while uploading

files to S3 gives you a completely different approach. You're able to attach Lambda invocations to S3 object lifecycle events like *ObjectCreated*. With this, new files will automatically trigger your function with an event that contains all necessary information about the lifecycle event and file.

You won't pay for any idle times, as only computations will be billed with Lambda.

### Use Case 2: Creating Backups and Synchronizing Them into Different Accounts

Lambda is the perfect tool for creating backups and synchronizing them between different storages or even accounts to have redundancy and with that high security.

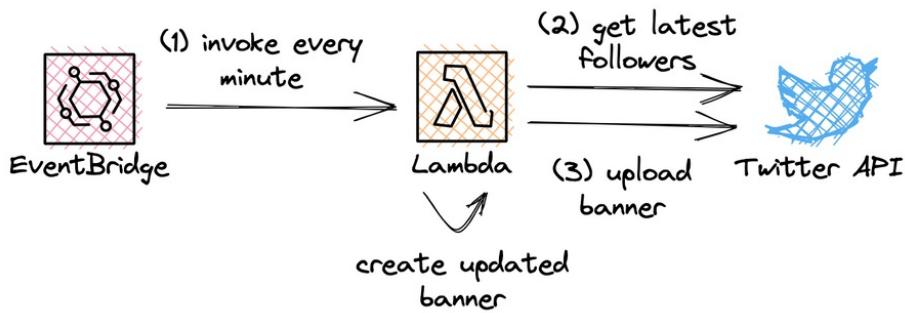


Looking at the previous image and video processing use case: you can also directly synchronize files based on lifecycle events and event notifications.

### Use Case 3: Scraping and Crawling Web Pages and APIs and Using the Data for All Kinds of Purposes

You can use Lambda to scrape and crawl web pages or other data sources to gather information for analysis or other purposes.

A common example for which you can find a lot of blog posts is to automatically update your Twitter banner based on recent followers, the blog post you've published recently, or any other information you want to show in near real-time.



Lambda can gather the required information from the Twitter API, create a new banner with updated information (e.g. via the sharp library), and then upload the results to your accounts again. The regular invocation of your function is taken over by an EventBridge rule, for example with a schedule for every minute.

### Tips and Tricks for the Real World

After hearing about all these great use cases, let's go through a few guidelines on how to make the best out of them. As also mentioned previously, this is not a complete list but a best-practice starting guide.

- **Keep your functions stateless and idempotent.** Function invocations can fail and retry mechanisms should not result in inconsistent states but always return the same result for the same request. As requests can be executed on different Lambda micro-container environments, it's also important that requests do not need to know a central, in-memory state.
- **Use CloudWatch Alarms.** CloudWatch already collects a bunch of metrics for free and it's good practice to keep track of them by setting up alarms. It's for example important to know when concurrency limits are breached or functions have a high error rate.
- **Pick the right database solution.** Lambda's computing resources are always temporary. Due to this fact, it's difficult to work with database solutions that are relying on connection pools, as functions can be de-provisioned at any time and opening and closing connections all the time is time-consuming. It's practical to use low-latency storages like DynamoDB that can be accessed via the AWS-SDK and does not require connection management.
- **Use Step Functions for orchestrating a large set of functions.** If you need to create multi-step workflows, such as automating a process that involves several Lambda functions and other AWS services, you can use AWS Step Functions.

- **Use Layers for shared dependencies.** If you’re working with multiple Lambda functions that do require the same external or internal dependencies, outsource them to a Lambda Layer. This will result in less operational overhead.
- **Try to keep your function’s code environment independent.** Environment variables are there to store configuration values that are specific to different stages of your application (e.g. development and production). This way you can easily configure different settings for different environments without hardcoding them into your function’s code.
- **Focus on event-driven, resilient architectures.** Invocations of your functions can always fail due to multiple reasons. Timeouts, internal errors, unavailable third parties, and much more. If you focus on building an event-driven architecture that embraces failure and allows for reprocessing at every step, you’ll end up with a resilient system that’s able to recover from any failure.
- **Use structured logging.** Rather than using simple text logs, write logs in a structured format like JSON. This makes it easier to search, analyze, and process the logs, as well as to automate certain tasks, such as alerting or aggregating metrics.

#### **The Best Practices to Reduce Cold Start Times**

Cold starts are a major topic and do heavily influence the performance and with that the perceived satisfaction with applications. That is why we want to deep-dive strategies to mitigate or reduce them.

There are a lot of tricks and best practices to reduce cold start times and also improve execution times by an order of magnitude:

- **Keep external dependencies minimal.** Think twice if you really need this new dependency and make use of tree-shaking processes (e.g. WebPack for TypeScript/JS) to only include the code in the deployment package that’s actually used. Each bootstrap of a micro-container does require your code to be shipped to the container instance and it needs to be loaded when the function starts. Fewer code results in faster launches.
- **Make use of warm-up requests that regularly invoke your functions** If your function is invoked regularly, the time window until a micro-container is de-provisioned and its resources are free is increased. You can also align the number of parallel warm-up requests to your traffic patterns to start multiple micro-containers in parallel.
- **Bootstrap as much code as possible outside of your handler function.** AWS grants high memory and vCPU settings for code that is executed before your handler

function, which means that you'll save additional time until your business code executes. More to this in a later paragraph.

- **Find the sweet spot for your function's memory size.** Less memory and therefore compute resources do not automatically result in a lower bill at the end of the month. Lambda is charged based on the configured memory and the executed milliseconds. Nevertheless, it doesn't mean that at the end of the month, you'll always pay more for a 512MB than for a 2GB function. More vCPUs will result in fewer execution times, especially for computing intense tasks. In other words, it's possible to lower your AWS Lambda bill by increasing memory size. You should monitor your cold starts via CloudWatch and custom metrics, e.g. by writing dedicated log messages and creating custom metrics. Afterward, you can see how different configurations will affect the number of cold starts and their duration.

## How to Determine If Lambda and the Serverless Approach Is the Right Fit

As mentioned in the starting chapters, we believe cloud-native is the future. This future heavily evolves around AWS Lambda, as it's the glue that keeps everything together. At the current time, as we've seen with cold starts, there are still some limitations and Lambda is not always the best fit for every requirement.

That's why we want to do a small dive into the requirements analysis to close the Lambda chapter. It's more of an advanced and not a beginner topic, but it's good to have a look into it anyway.

Before you start migrating an existing service or building a new service with a Serverless architecture powered by Lambda, you should ask yourself a set of predefined questions to find out whether Serverless is a fitting approach:

- Does the service need to maintain a **central state**? This can be information that is kept in memory but needs to be shared over all computation resources.
- Does the service need to **serve requests very frequently**?
- Is the architecture rather **monolithic** instead of built out of small, loosely-coupled parts?
- Is it **known how the service needs to scale out** on a daily or weekly basis and how the traffic will grow in the future?
- Are processes mostly revolving around **synchronous operations**?

The more questions answered with **no** the better. If you've answered some questions with yes, it doesn't mean you can't go with Lambda, but you'll face at least some trade-offs in comparison to traditional container technologies like AWS Elastic Container Service (ECS).

## Final Words

There's no other service that allows you to quickly build amazing services without spending time on containers, virtual private networks, gateways, and other infrastructure. You've got the code you want to run and Lambda will offer you the environment to do so without having many strings attached. It's also the glue for every Serverless project you'll build, see, or explore in the future.

Personally, we'd also say it's the best service to get started with learning AWS.



# Database & Storage

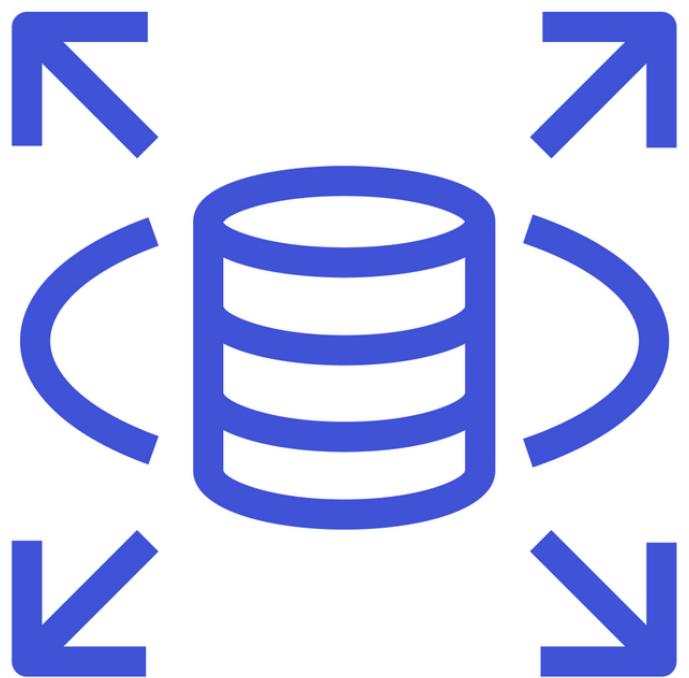
Databases and storage options are crucial for building web applications because they provide a way to persist and retrieve data efficiently. Without them, it would be difficult to store user information, track the application state, and perform other tasks that are necessary for a web application to function correctly.

As we've seen in the previous chapters, it's good to hand over as much responsibility as possible for infrastructure management. That's why Amazon Relational Database Service (**RDS**), Simple Storage Service (**S3**), and **DynamoDB** are great options for building web applications. They are managed, mature, and offer high resiliency and availability. They are also very beginner friendly so there's no previous knowledge necessary.

RDS supports popular databases such as MySQL, PostgreSQL, and Oracle. S3 is a highly durable object storage service that can be used to store files and media. **DynamoDB** is a NoSQL database service that is optimized for low-latency and high-throughput access to data.

Amazon Simple Storage Service (**S3**) is designed for storing large amounts of unstructured data, such as images, videos, and documents. It is a highly durable and scalable service, with data stored across multiple devices in multiple locations, which ensures that data is always available. S3 is also highly secure and customizable, allowing users to set access controls and encryption options to meet their specific needs.

All three services are also highly available, highly scalable, and fully managed, which makes it easy to build highly available web applications. They also support many advanced features such as encryption and backups to help secure and protect data.



Amazon **RDS**

# Fully-Managed SQL Databases with RDS

## Introduction

Amazon Relational Database Service (RDS) is the fully-managed SQL database offered by AWS. It allows you to get a SQL database up and running in a few clicks. You don't need to manage the infrastructure or operating system.

It is a very popular service because it reduces the efforts of provisioning and administration of databases. The alternative before RDS was setting up your own virtual machine. With the VM you needed to take care of

- Operation System configuration
- System updates
- Security
- Network configuration

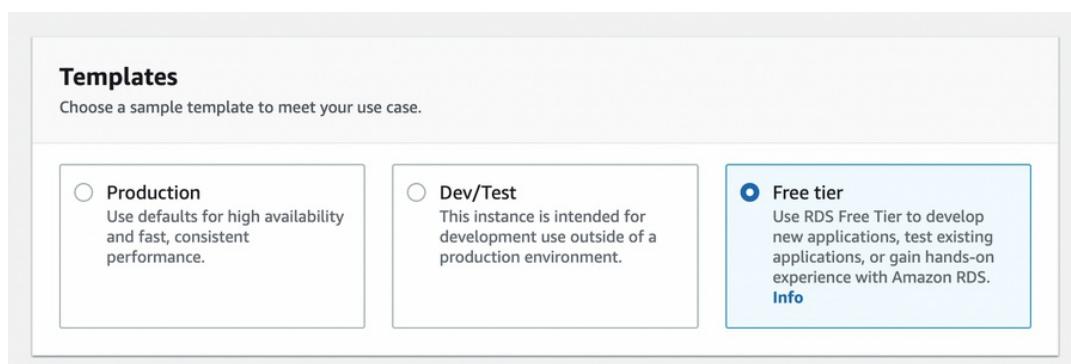
...and much more. With RDS this is much easier.

You set up your database, receive a connection string, and you are good to go.

## Create Your First Database With the Management Console

If you want to follow along we recommend creating a database together. Be aware: RDS is charged by the minute. You can choose the free tier template and AWS will give you the options that you pay close to nothing.

Go to the RDS console and click on `Create Database`. Choose the `Free tier` template.

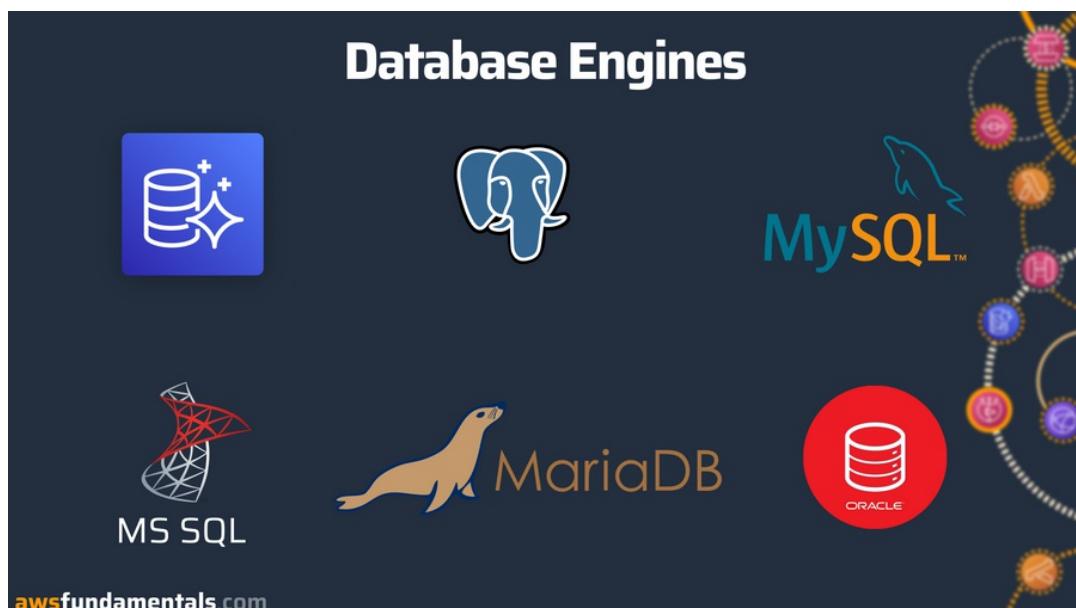


You will be prompted to enter a database admin password. Make sure to note this down in your password manager.

After creating your instance you will receive your endpoint: `database-1.cadwpqb0pb1t.eu-central-1.rds.amazonaws.com`

### RDS Supports Six Different Database Engines

RDS comes with different engines for different requirements.



Among them are:

1. Amazon Aurora
2. MySQL
3. MariaDB
4. PostgreSQL
5. Oracle
6. Microsoft SQL Server

One thing to be aware of these engines is the exact version AWS supports. The engine itself is an official release of the engine. But AWS often supports certain versions of this engine.

For example, in RDS you can use Postgres 14.3-R1. The latest minor version is 14.4. If you need a certain version **check this before** migrating the whole workload to RDS.

### Your Database Can Have Different States from Available to Deleting

In RDS your database can be in different states. The most important states are

State	Description
available	Your database is available and ready to receive requests.
backing-up	There's a running process that will create a backup of your database. You can have a higher load on the CPU.
creating	The database will be created.
deleting	The database will be deleted.
failed	Your DB failed and RDS can't recover it.
maintenance	RDS is doing maintenance work on your database. You defined the maintenance window.
modifying	There are user modifications going on.
stopped	The database is stopped.
storage-full	The storage of your database is full.
upgrading	Your database upgrades to a new version.

There are many more states to discover.

### Use the Multi-AZ Feature to Recover From Failures in Availability Zones

Multi-AZ refers to the functionality of having your database available in multiple availability zones. Let's first see an introduction to AWS Data Center in general.

#### One Datacenter Is Available in One Region and One Region Has Multiple Availability Zones

AWS separates its data center into regions, and inside one region into availability zones.

Once you log in to the AWS console you need to choose a region like `us-east-1`. Your workload will run in this particular region.

AWS doesn't have just one data center in this region but multiple. In the `us-east-1` region, for example, you have **six different availability zones**.

You can define that your database runs in Availability Zone A. However, you don't know where this Availability Zone is physically located. This is for security reasons. But it does make sense to have your workloads in the same AZ. You can figure out your AZ IDs in the **Resource Access Manager Service**.

AZ Name	AZ ID
us-east-1a	use1-az6
us-east-1b	use1-az1
us-east-1c	use1-az2
us-east-1d	use1-az4
us-east-1e	use1-az3
us-east-1f	use1-az5

There you will find IDs to map your workloads even across different accounts.

All current and upcoming regions can be found in the AWS documentation.

#### **Creating Multi-AZ Resiliency With RDS Is Configurable**

Citing Werner Vogels, CTO of AWS: "Everything fails, all the time!"

This is a pretty important quote. Your server will fail. Even at AWS. This is why it is so important to have proper resiliency. If your server or AZ is offline your database should still be available.

In RDS this is configurable. Choose the option Multi-AZ DB Instance during the generation of the database. This makes your database available in multiple AZs.

You can also configure this after creating your database.

## Availability and durability

### Deployment options Info

The deployment options below are limited to those supported by the engine you selected above.

Multi-AZ DB Cluster - new

Creates a DB cluster with a primary DB instance and two readable standby DB instances, with each DB instance in a different Availability Zone (AZ). Provides high availability, data redundancy and increases capacity to serve read workloads.

Multi-AZ DB instance

Creates a primary DB instance and a standby DB instance in a different AZ. Provides high availability and data redundancy, but the standby DB instance doesn't support connections for read workloads.

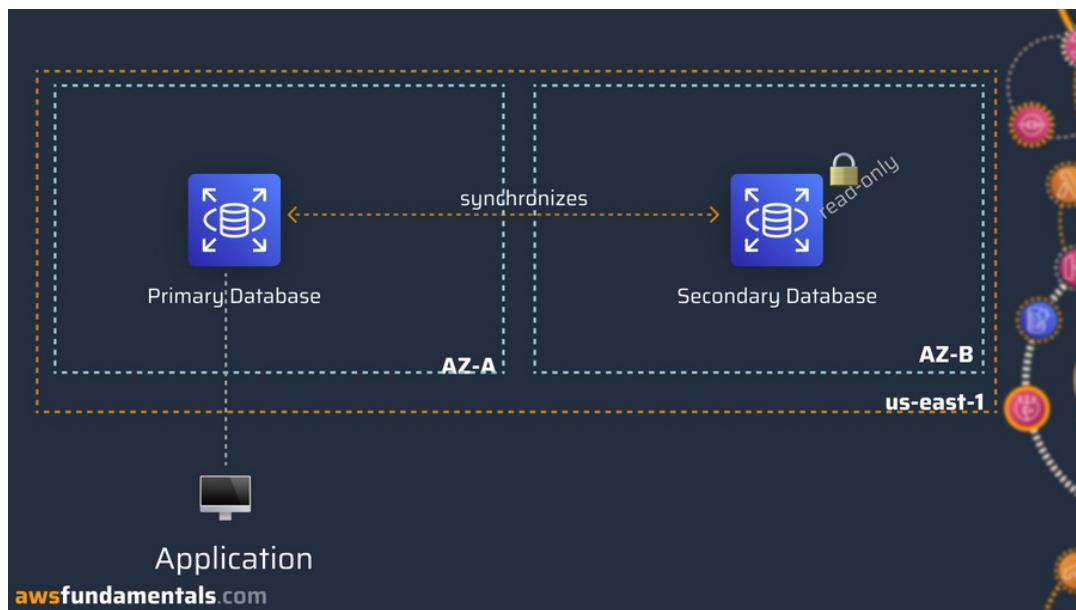
Single DB instance

Creates a single DB instance with no standby DB instances.

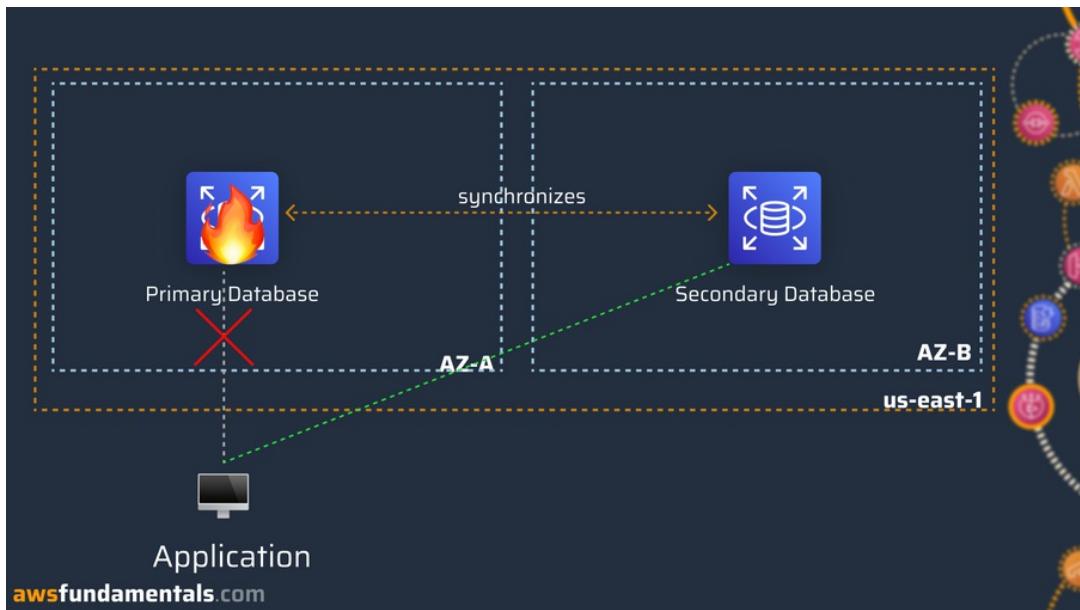
### Multi-AZ Creates a Primary and a Secondary Database Which Will Be Synchronized Constantly

But what happens in the background if you activate Multi-AZ?

RDS creates a second database for you. Say your primary database is in AZ A. Once you activate multi-AZ a secondary database, for example, AZ B will be provisioned. This database synchronizes with your primary database. The secondary database will be read-only.



Once your primary database has issues and goes offline, AWS guarantees that your connection string automatically goes to your secondary database with reading & writing access. This is without any data loss or manual intervention. All of that is managed by AWS.



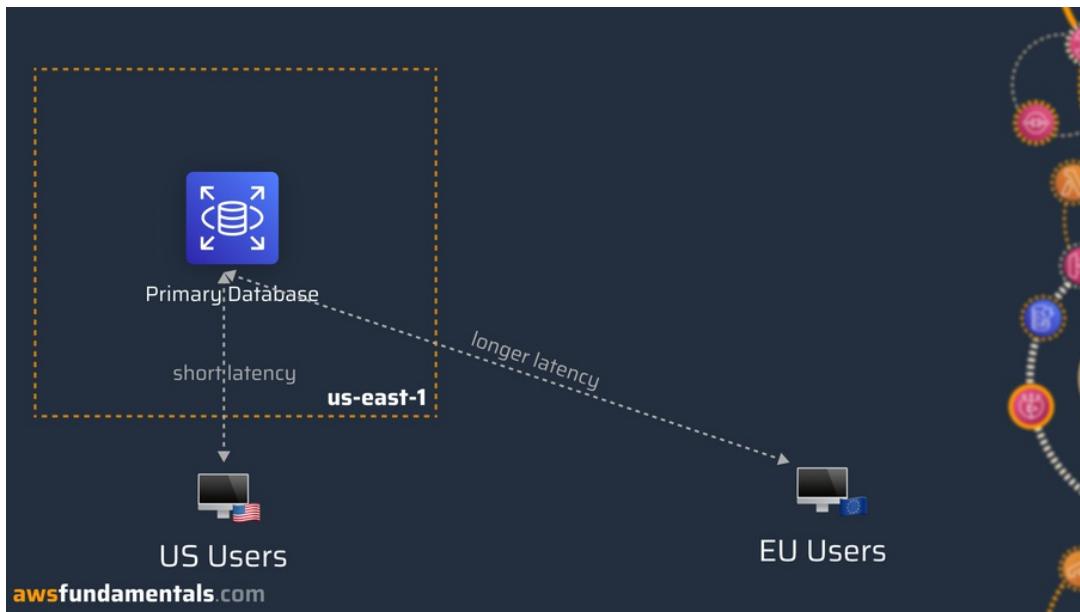
Achieving this behavior a few years back was a huge effort. In AWS it is simply clicking a button.

The behavior and startup times are dependent on the database engine you use. The best thing to do is as **always to test things!** Don't rely on things to be working out. Instead, test it out. AWS has a good comparison table to see how different engines behave and how to choose the right one.

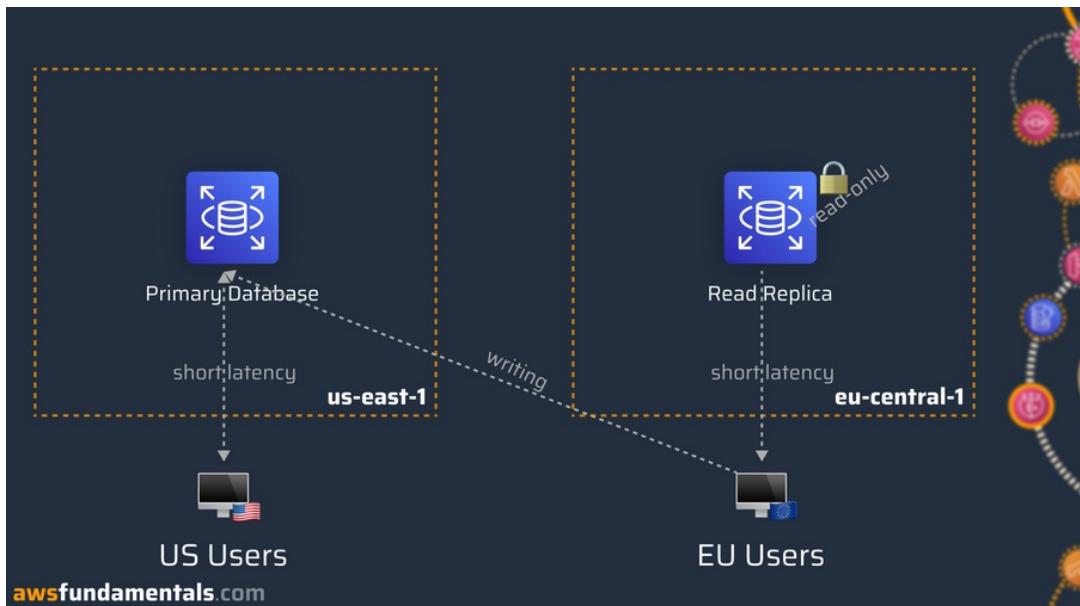
### **Read Replicas Allow You to Increase Performance for Read-Heavy Workloads Across Regions**

Another great functionality of RDS is the usage of Read Replicas. A read replica is a second database that only enables read access. You often use read replicas for read-heavy applications to increase performance. Read Replicas can be in the same region or you can create one in another region.

Let's say you have an application where most users are in the US. Your application will live in `us-east-1`. All users from other countries like Europe will experience a longer latency. This is not ideal.



To increase the performance you can create a read replicate in `eu-central-1`. This enables a much shorter latency to the database for reading workloads. Writing will still happen in `us-east-1`.



You can do this in the console by selecting your database and clicking on `Create Read Replica`.

The screenshot shows the AWS RDS Databases management interface. At the top, there's a navigation bar with 'RDS' and 'Databases'. Below it is a search bar labeled 'Filter by databases'. A table lists databases, with 'database-1' selected. To the right of the table are dropdown menus for 'Region & AZ' and 'Size'. A context menu is open over 'database-1', listing actions such as 'Actions', 'Delete', 'Restore from S3', 'Create database', 'Set up EC2 connection', 'Create read replica', 'Create Aurora read replica', and 'Promote'. The 'Create read replica' option is currently selected.

Often you separate not only by users but also by applications. Your reporting and business intelligence software should use a read replica.

Each read replica has its own endpoint. For example:

- primary: `mydb.123456789012.us-east-1.rds.amazonaws.com`
- read replica: `myreadreplica.123456789012.us-east-1.rds.amazonaws.com`

## Automate Backups with RDS

With backups, you can restore your database back to a certain point in time. In case you introduced a bug, deleted entries on accident, or any other incident occurred that resulted in corrupted or lost data.

RDS supports automated backups. These will constantly create snapshots of your data.

### Backup

**Enable automated backups**  
Creates a point-in-time snapshot of your database

**Backup retention period** Info  
The number of days (1-35) for which automatic backups are kept.

7 ▼ days

**Backup window** Info  
The daily time range (in UTC) during which RDS takes automated backups.

Choose a window

No preference

You can configure your backup in the RDS console. You configure a backup window which is a time when a backup takes place.

The backup window is optional. If skipped, RDS will fall back to default times for each region.

Important to keep in mind: the backup process will **increase the load** on your database while it is created. It's a good practice to configure backup windows for times when your application is not at peak load, e.g. at night times instead of during business hours.

Retention times refer to the length of time that automatic backups are stored. The defaults are:

- One day if you created the DB from an API or CLI
- Seven days if you've created it from the console.

You can easily restore your backup either via API or via the console. Just choose a backup and promote it to your current data.

### **Encrypt Your Data Directly on the Server**

You can encrypt your data within RDS. If you activate encryption your data will be **encrypted at rest**. That means your data is encrypted on the actual hardware. It is encrypted with the industry standard of AES-256.

#### **Encryption**

**Enable encryption**  
Choose to encrypt the given instance. Master key IDs and aliases appear in the list after they have been created using the AWS Key Management Service console. [Info](#)

AWS KMS key [Info](#)  
 ▾

RDS activates this by default for every database creation. You should keep this activated.

The default encryption key is stored in KMS. You can also provide your own keys or provide a customer key. With that only, your customer can decrypt the data. All logs, backups, and snapshots are encrypted. A read replica of your encrypted database also needs to be encrypted.

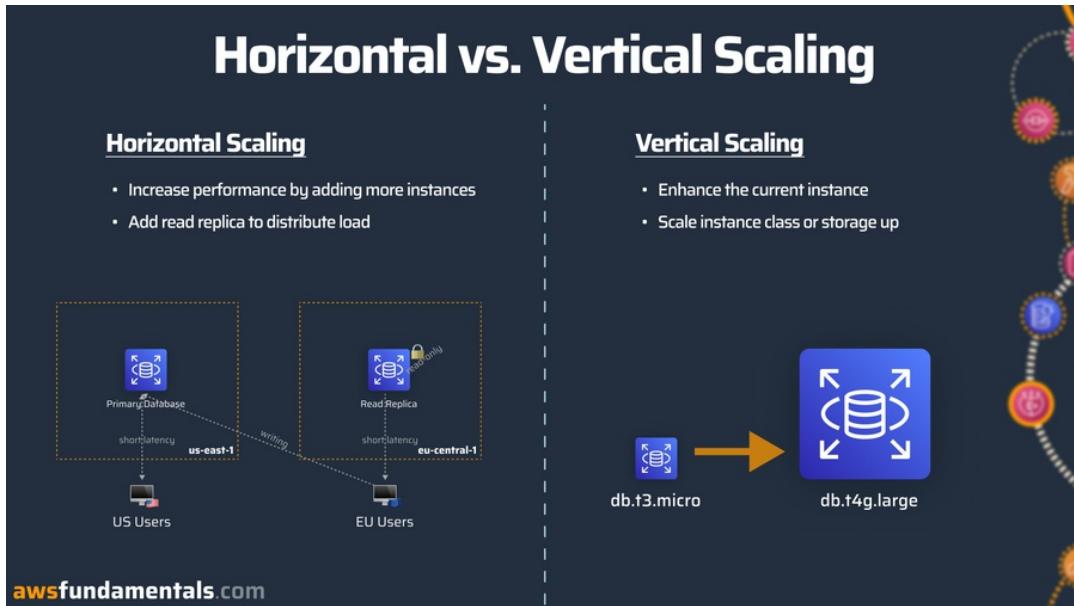
### **RDS Supports Scaling Its Storage Automatically but Scaling the Instance Class Requires Manual Work**

Scaling up and down your application is what makes RDS powerful. This can either happen at predictable peak times or at unpredictable events.

- **Upscaling** - means your database requires **more** resources like CPU or storage.
- **Downscaling** - means that your database needs **fewer** resources to save costs.

You can configure auto-scaling policies with RDS. With these policies, you are able to automatically scale up and down.

#### Horizontal Scaling Adds More Instances While Vertical Scaling Enhances the Current Instance



#### Horizontal Scaling Adds More Instances

Instead of having one database instance, you would have two or three for example. Adding **read replicas** is one of the most straightforward ways of horizontal scaling. This was already discussed in the read replica chapter. The main point is: If your application only needs to scale for reading reasons add read replicas. You will find that this happens often.

Another way to increase the performance by scaling horizontally is by adding an RDS proxy. An RDS Proxy pools database connections and lets you share connections. This is especially important if you have serverless workers like Lambda Functions.

#### Vertical Scaling Enhances the Current Database Instance

This results in more memory, more CPU, or more storage.

**Auto Scaling** allows you to scale up and down automatically, based on a policy. A policy can be based on a metric like **free storage**.

A common example is: if there is only 10% free storage → Scale up

Auto-scaling is only possible for **storage** not for the **actual instance class**.

You can scale the instance class with a manual solution. For example, you could use EventBridge & Lambda to scale it up. But there are also downtimes involved so be careful about that.

### RDS Is Priced on the Instance Class & How Long the Instance Runs

It's important to understand that RDS runs an instance 24/7 so it doesn't scale to zero. Even if no requests are coming in you will still pay for having the database. You can check out the pricing calculator to get an accurate estimate of your database.

The main price factors are:

- Instance class: db.t3.micro
- Number of nodes: 1
- Pricing mode (On-Demand vs. Reserved): You can reserve instances or pay on-demand
- Amount of storage: How much storage is available
- Backup: How many backups were taken

### Use Cases from the Real World

There are tons of examples of use cases for SQL databases. Everywhere, where you need to save data and persist data is suitable for a SQL database. Let's see three example use cases you will see a lot in the industry.

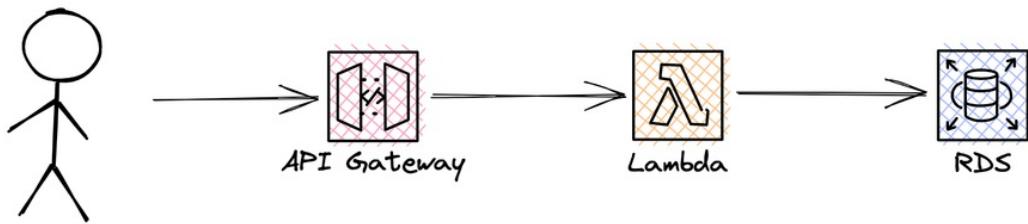
#### Use Case 1: Building Web and Mobile Applications

Building web and mobile applications is a common use case for RDS. Let's say you develop customer management software. In your RDS database, you can save all your customer data like:

- Names
- Addresses
- Telephone Numbers

RDS makes it easy to set up a database and store this data in the cloud. A typical application could exist out of the following services:

- **API Gateway** - exposing your business logic
- **Lambda** - executing business logic on API requests
- **RDS** - persisting data



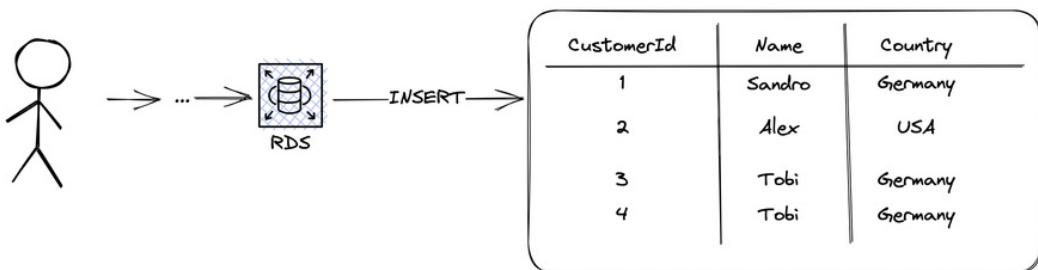
This architecture allows you to build a redundant, resilient, and high-performance application without having to worry about the underlying infrastructure.

In addition, RDS allows you to scale up your database to handle peak loads and automatically creates backups to ensure the availability of your data. This makes it an ideal choice for building web and mobile applications that need to store and persist data.

#### Use Case 2: Analytical Use Cases like OLAP Operations

SQL databases are often compared to NoSQL databases like DynamoDB. One of the main advantages of SQL is Online Analytical Processing (OLAP) transactions. This means grouping data together. For example, with basic aggregates such as sums or averages.

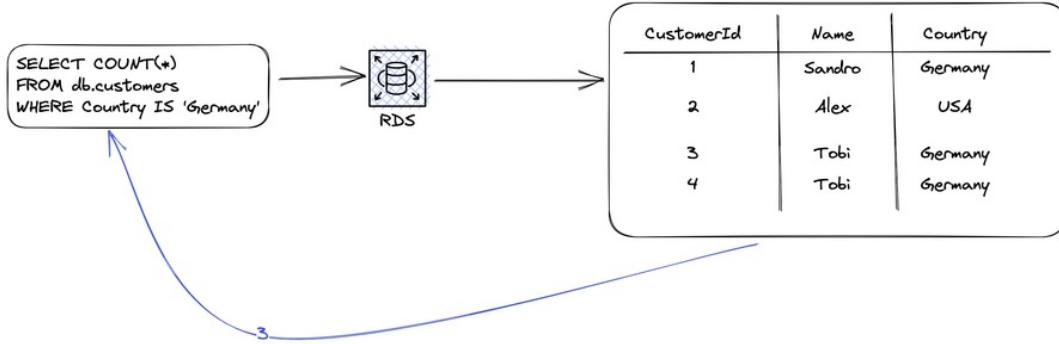
Let's take our customer management software as an example again. This is what your data looks like:



You want to know how many of your users are living in Germany. By using SQL statements you

can group your data. The corresponding SQL statement would look like this:

```
SELECT COUNT(*) FROM db.customers WHERE Country IS 'Germany'  
# Result: 2
```



SQL databases are fast with such operations. You describe the result you want to see and the database figures out how to get you the results. This is a typical OLAP operation.

This is not possible with NoSQL databases. NoSQL databases have other benefits, which we will highlight in their own chapter.

### Use Case 3: Moving on-Premise Databases to Managed Databases

One common use case for RDS is getting the benefit of the fact that it is a managed service. Many companies have their whole databases running on bare metal servers or virtual machines. This includes a lot of overhead. There are dedicated engineers that take care of:

- Backing up databases
- Updating the underlying operating system
- Keeping track of licensing issues
- Scaling databases manually

Many of these tasks can be automated and lifted off to AWS by using RDS. A very common project is to move on-premise workload (workload that is not on the cloud) to the cloud.

Many on-premise databases also have really expensive licenses. Another step that is often involved here is to get rid of the expensive licensing costs. Engines like Aurora or Postgres are often used for that since they are free.

### Tips for the Real World

- **Use Multi-AZ** - your database or Availability Zone will fail. It is best practice to have a second standby database always available.
- **Activate managed backups** - with RDS it is really easy to activate backups. Make use of them. Not having backups can be a huge business risk.
- **Enable Read Replicase** - especially for read-only workloads this is a must. You don't want to overload your database with read-intensive workloads.
- **Activate Autoscaling** - create auto-scaling policies to scale your storage automatically.
- **Understand connection pooling** - especially for serverless applications this is a must. SQL databases and serverless applications can have issues with too many connections. Use connection pools or connection APIs if you use a serverless compute layer.

## Final Words

RDS is one of the core services in AWS. You won't find a larger organization that operates in the cloud and is not using RDS.

It gives you the ability to build SQL databases very easily. Be aware of the costs because RDS can get quite expensive. Scaling databases is possible with RDS but it is not built-in like with DynamoDB.



Amazon **DynamoDB**

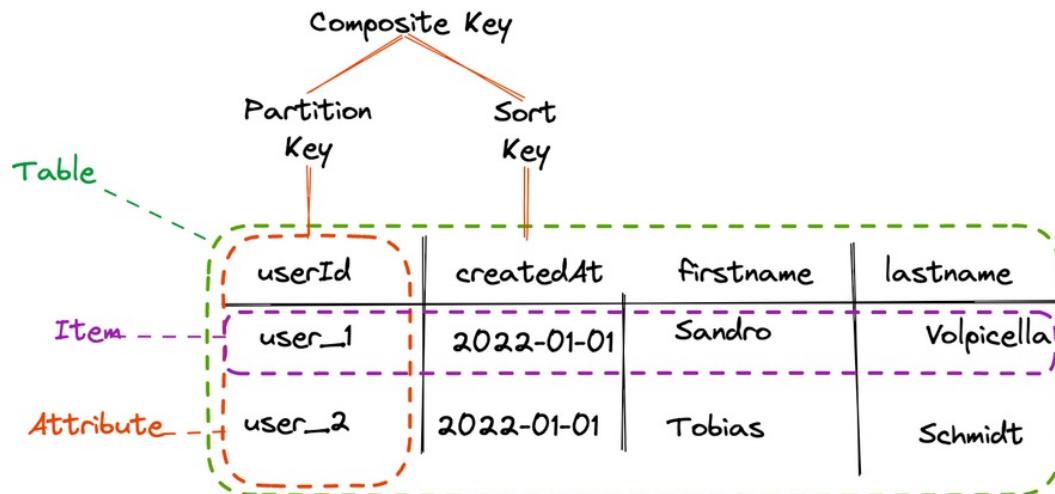
# Building Highly-Scalable Applications in a True Serverless Way With DynamoDB

## Introduction

DynamoDB is a fully-managed NoSQL database that is able to handle any scale. Additionally, it offers great features to integrate natively with other services. As it's not your common NoSQL storage but comes with a long list of unique points, it's important to understand its internals beforehand.

## Understanding DynamoDB's Key Concepts

DynamoDB's internals are built around the following major concepts: **tables**, **items**, **attributes**, and **types**. Let's dive into those.



Term	Definition
<b>Table</b>	A table is a collection of items. A table in DynamoDB stores many items and is similar to a table in a SQL database. In our example, it is the whole user table.
<b>Item</b>	An item is one row of a table. For example, one user can be one item. One item can have several attributes like last name, first name, address, etc.
<b>Attributes</b>	An attribute is a field within an item. It is like a key in a JSON document. An attribute is described as a type like a string, number, etc. One attribute can also be another whole JSON document.

Term	Definition
<b>Type</b>	DynamoDB has different types of attributes. These are: - Number - String - Binary - Boolean - Null - List: ["Apple", "Banana"] - Map: Another document / JSON - Sets: List with one data type, e.g. only numbers. We'll look at them in more detail.
<b>Primary Key</b>	A primary key <b>uniquely</b> defines an attribute within a table. There are two sorts of primary keys in DynamoDB: 1. <b>Primary Keys</b> : A single key identifying the item. E.g. <code>userId</code> 2. <b>Composite Keys</b> : A composite key is composed of two keys. A <b>partition and a sort key</b> . The composition of the two keys <b>identifies your item uniquely</b> . Not the partition key alone but only the composition of both. A sort key sorts your query automatically. This can be very handy. For a user table, this can be for example <code>userId</code> and <code>createdAt</code> . We will see that in more detail later on.

### **DynamoDB Is a NoSQL Database. It Saves Your Data in a JSON Format and Doesn't Enforce a Schema**

DynamoDB is **not a SQL** database, but a NoSQL database. NoSQL has different definitions. It is often referred to as non-SQL or non-relational. But don't take the no in non-SQL too literally, as there's only no explicit schema. In an SQL database (e.g. RDS - Aurora) you need to define a schema. Your data needs to adhere to that schema. Let's take the example from the RDS chapter once again:

CustomerId	Name	Country
1	Sandro	Germany
2	Alex	USA
3	Tobi	Germany
4	Tobi	Germany



Each customer in this database needs to have these fields. You cannot add additional fields.

In DynamoDB this is not enforced during the writing of the data. That means you can also add

other fields to the table that weren't defined before.

CustomerId	Name	Country	Address	Email
1	Sandro	Germany	null	s@s.de
2	Alex	USA	{street}	null
3	Tobi	Germany	null	null
4	Tobi	Germany	null	null

 DynamoDB

In this example, we also added the address and the email. The data is not normalized in this case. Meaning that you will often see a lot of null values in DynamoDB tables.

We still have an implicit schema as your application still expects your data in a certain format. It's not random and it often can't be changed easily. This is especially true for DynamoDB and its access pattern requirements, as we'll explore later on.

DynamoDB is a **key-value** database. It stores data differently than SQL databases

Items within DynamoDB are documents and follow the JSON syntax. Let's have a look at one of our uses in the database.

This is how we see a user in a normal JSON view.

```
{
  "userId": "user_1",
  "createdAt": "2022-08-20",
  "firstname": "Sandro",
  "lastname": "Volpicella"
}
```

DynamoDB internally maps each JSON key (for example `userId`) to a datatype, e.g. a string or number. We refer to this as the **DynamoDB JSON**.

The DynamoDB JSON looks like the following code block.

```
{
  "userId": {
    "S": "user_1"
  },
  "createdAt": {
```

```

    "S": "2022-08-20"
},
"firstname": {
    "S": "Sandro"
},
"lastname": {
    "S": "Volpicella"
}
}

```

In DynamoDB the key `"S"` defines the type, in that case, String. We will dive into that a bit more in the section on data types.

If you work with DynamoDB you will often have so-called marshallers in place. They return you the "normal" JSON object like the first one.

But it is still important to understand how DynamoDB stores your data.

### No Schema

One of the main differences between SQL to NoSQL databases is the schema.

SQL enforces you to stick to a pre-defined schema. You need to add certain attributes. And you are only allowed attributes that are present in the schema.

NoSQL works differently. You can write anything to your database. The only attribute that needs to exist is the primary key.

All other data can be written to the database. It is common practice to have attributes only available for a certain set of your data. This will result in many empty fields and that is fine.

Developers coming from a strong SQL background often have issues with that. The data is **not normalized** like it is in SQL.

userId	createdAt	firstname	lastname	subscription
user_1	2022-01-01	Sandro	Volpicella	null
user_2	2022-01-01	Tobias	Schmidt	sub_1

Let's see an example. Users in our database can have a subscription. Only pro users have the

attribute subscription. All free users don't have this attribute.

`user_2` has the subscription `sub_1`. `user_1` doesn't.

### NoSQL Doesn't Mean Non-Relational. Your Data Always Has Relations

A word about non-relational. NoSQL often refers to a database not having relations. What this actually means is of a technical nature. The AWS documentation for example states the following:

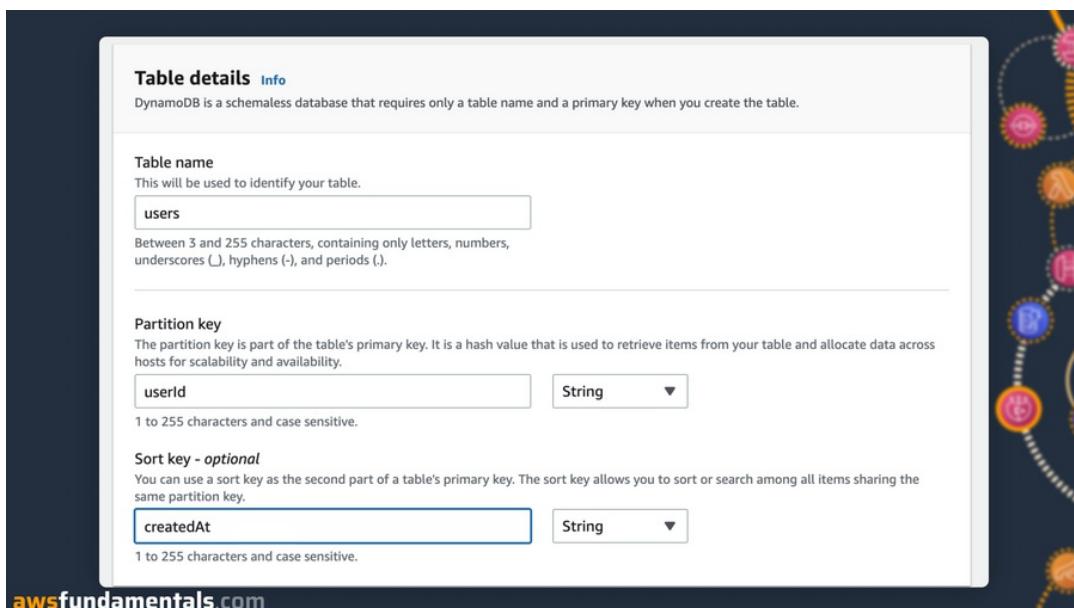
NoSQL is a term used to describe nonrelational database systems that are highly available

While the technology **is nonrelational** your data still has relations. Unlike SQL you cannot do Joins across tables. But your data still has relations.

Your users do have relations to other tables. A user table and an order table are still connected. Users and Subscriptions are also connected. Keep this in mind while setting up your database. We'll see different access patterns later. These will be important to create based on your relations.

### Creating a Table

Go to the DynamoDB service in your Management Console and click on **Create Table**.



Let's use the following keys:

- Partition Key: `userId`
- Sort Key: `createdAt`

This table allows you to follow all examples in this chapter.

## **Primary Keys Identify an Item Uniquely. They Can Be a Simple or a Composite Key**

Primary Keys identify one item in your database uniquely. This concept is similar to a SQL database. DynamoDB comes with different options for primary keys: **simple** and **composite**.

### **Simple Primary Key - Solely Relying on the Partition Key**

A simple primary key is one attribute that identifies the item in the table. In a user table, this could be the `userId`.

Once you add a user with the `userId user_1`, you cannot add another one with this ID. The primary key refers to a partition key in that case.

A **Partition Key** defines how DynamoDB stores your data. DynamoDB uses partitions to save your data. Check out this guide if you want to know more about that. This is not crucial for starting out with DynamoDB.

### **Composite Primary Key - A Combination of Partition and Sort Key**

Contrary to simple keys, composite keys are built of two parts: the **partition** and the **sort key**.

As the name suggests, the sort key enables sorting operations via queries. In our user example, you would get all users in sorted order by the `createdAt` date.

In our defined table the primary key exists out of both a partition key and a sort key:

- Partition Key: `userId`
- Sort Key: `createdAt`

This allows having users with the same `userId` but different `createdAt` times.

	userId	createdAt	subscription
	user_1	2022-01-01	{ "customer_id": { "S": "cus_1" }, "plan_id": { "S": "plan_1" } }
	user_1	2022-01-02	

## Composition within Sort Keys

It's possible to take this one step further by introducing compositions within sort keys. We could combine the team and timestamp in our example like the following:

```
[team] # [timestamp]
```

It doesn't seem natural at the first glance but offers powerful query mechanisms. We could use the above key as our sort key in the user table.

The following operations can be applied to sort keys:

- Equal to (=)
- Less than or equal to (<=)
- Less than (<)
- Greater than (>)

By having more data within your sort key you can fulfill even more query requirements.

Let's add three more users:

	userId	createdAt
	user_2	companya#2020-08-21
	user_2	companya#2020-08-20
	user_2	companyb#2020-08-20

We've added some meta information to the `createdAt` field.

```
[COMPANY] # [DATE]
```

We can do several things now.

1. Query for `user_2`
2. Query for `user_2` and `createdAt` begins with `company`

The first query will show us all `user_2`. The second query allows us to select only a subset of the data

The screenshot shows the AWS DynamoDB console interface. At the top, there's a dropdown menu labeled "Scan/Query items". Below it, a "Scan/query a table or index" section with tabs for "Scan" and "Query". The "Query" tab is selected, and the table name "Users" is chosen. Under "userId (Partition key)", the value "user\_2" is entered. Under "createdAt (Sort key)", the filter "Begins with" is set to "company". A "Sort descending" checkbox is unchecked. At the bottom of this section are "Run" and "Reset" buttons. Below this, a status message says "Completed" with a green checkmark and "Read capacity units consumed: 0.5". The main area displays the results under "Items returned (2)". Each item has a checkbox, the "userId" value "user\_2", and the "createdAt" value "company#2020-08-20" or "company#2020-08-21". There are also "Actions" and "Create item" buttons, and navigation controls for the results.

userId	createdAt
user_2	company#2020-08-20
user_2	company#2020-08-21

While this access pattern is not necessarily useful for our example, thinking about access patterns while designing your keys is a huge benefit as you can't change them afterward.

You can add indexes later which will have their own keys. But the actual primary key **cannot be changed**.

### Exploring the Different Data Types: Scalar, Document, or Set

Each attribute in DynamoDB has an associated type. All types can be categorized into three categories **Scalar**, **Document**, and **Set**.

#### Scalar Types Are Strings, Numbers, Booleans, or Null Values

```

{
  "userId": {
    "S": "123"
  },
  "age": {
    "N": "28"
  },
  "proUser": {
    "B": true
  },
  "subscription": {
    "NULL": true
  }
}

```

**String** ————— "S": "123"  
**Number** ————— "N": "28"  
**Boolean** ————— "B": true  
**Null** ————— "NULL": true

**Scalar** data types represent exactly one value.

Data type	DynamoDB Representation
<b>String</b>	S
<b>Number</b>	N
<b>Binary</b>	B
<b>Boolean</b>	BOOL
<b>Null</b>	NULL

Take our user table as an example. You can see the `userId` is a string (`S`) while `age` is a number (`N`).

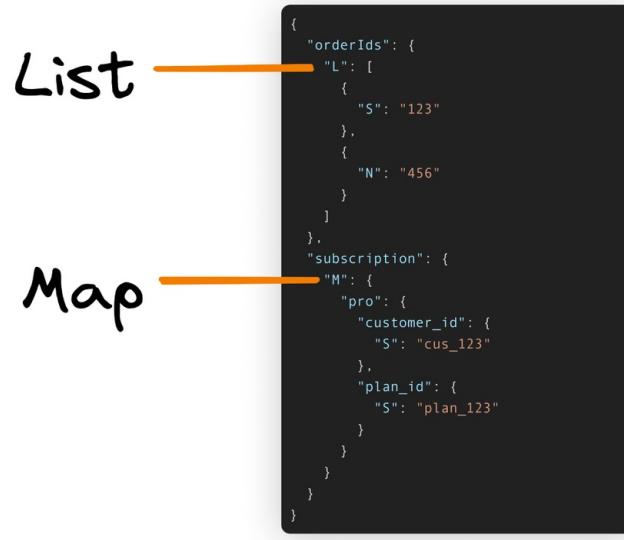
## DynamoDB JSON

```
{  
  "userId": {  
    "S": "user_1"  
  },  
  "age": {  
    "N": "28"  
  },  
  "createdAt": {  
    "S": "2022-08-20"  
  },  
  "firstname": {  
    "S": "Sandro"  
  },  
  "lastname": {  
    "S": "Volpicella"  
  }  
}
```

## JSON

```
{  
  "userId": "user_1",  
  "age": 28,  
  "createdAt": "2022-08-20",  
  "firstname": "Sandro",  
  "lastname": "Volpicella",  
}
```

## Document Types



**Map**

**Document datatypes** can either be a List or a Map. A list is an array structure while a map is like a JSON object.

Data type	DynamoDB Representation
<b>List</b>	L
<b>Map</b>	M

A list is a simple array of **similar or different** scalar types. For example, for our user example, we could have a list of all `orderIds`. This list can contain numbers but it can also contain strings.

## DynamoDB JSON

```
{  
  "userId": {  
    "S": "user_1"  
  },  
  "createdAt": {  
    "S": "2022-08-20"  
  },  
  "firstname": {  
    "S": "Sandro"  
  },  
  "lastname": {  
    "S": "Volpicella"  
  },  
  "orderIds": {  
    "L": [  
      {  
        "S": "123"  
      },  
      {  
        "N": "456"  
      }  
    ]  
  }  
}
```

## JSON

```
{  
  "userId": "user_1",  
  "createdAt": "2022-08-20",  
  "firstname": "Sandro",  
  "lastname": "Volpicella",  
  "orderIds": ["123", 456]  
}
```

Each item in a list can have a different **scalar type**. So the list could also look like that: [123,

```
"ORDER", NULL].
```

The second type of the Document category is a **Map**. The map is a JSON document as an attribute. A common example is to have a subscription object in your user model.

## DynamoDB JSON

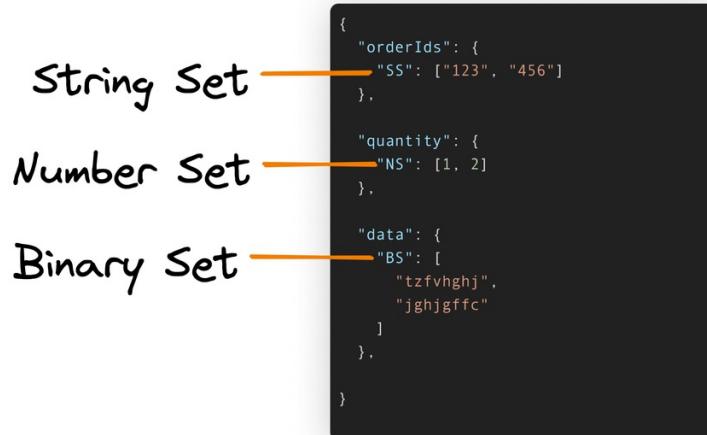
```
{
  "userId": {
    "S": "user_1"
  },
  "createdAt": {
    "S": "2022-08-20"
  },
  "firstname": {
    "S": "Sandro"
  },
  "lastname": {
    "S": "Volpicella"
  },
  "subscription": {
    "M": {
      "customer_id": {
        "S": "cus_123"
      },
      "plan_id": {
        "S": "plan_123"
      }
    }
  }
}
```

## JSON

```
{  
    "userId": "user_1",  
    "createdAt": "2022-08-20",  
    "firstname": "Sandro",  
    "lastname": "Volpicella",  
    "subscription": {  
        "customer_id": "cus_123",  
        "plan_id": "plan_123"  
    }  
}
```

You can see that the user now has an attribute `subscription` with a containing object. This object has two scalar types `customer_id` and `plan_id`.

### Set Types Are Lists of the Scalar Type String or Number



**Set** datatypes represent a list of the **same scalar value**. There are three different set types out there.

Data type	DynamoDB Representation
<b>String Set</b>	SS
<b>Number Set</b>	NS

Data type	DynamoDB Representation
Binary Set	BS

While a normal list can have different types mixed (like a number, string, and NULL). A **Set** type can only have one type. If we take our `orderIds` as an example, this would be a **String Set**.

### DynamoDB JSON

```
{
  "userId": {
    "S": "user_1"
  },
  "createdAt": {
    "S": "2022-08-20"
  },
  "firstname": {
    "S": "Sandro"
  },
  "lastname": {
    "S": "Volpicella"
  },
  "orderIds": {
    "SS": [ "123", "456" ]
  }
}
```

### JSON

```
{
  "userId": "user_1",
  "createdAt": "2022-08-20",
  "firstname": "Sandro",
  "lastname": "Volpicella",
  "orderIds": [ "123", "456", "789" ]
}
```

Sets can also be of the type binary (base 64 encoded) or number sets.

## You Can Use Scans or Queries to Retrieve Items. Queries Are Cheaper and Faster

A large part of using DynamoDB is **reading & writing data**. For most use cases, this maps down to the following operations: **Query & Scan**.



### Scan - A Slow Iteration through the Whole Database

A scan goes through the **whole database** and retrieves **a list of items**. Depending on the number of items in your database this list doesn't have to be complete. Your data is also structured into different pages. This is called pagination.

Completed Read capacity units consumed: 2

**Items returned (5)**

	userId	createdAt	age	orderIds	orders	subscription
<input type="checkbox"/>	user_2	companya#2020-08-20	28	[ {"S": "1"}, {"S": "2"} ]	{ "hijklhijkl": true }	{ "customer_id": "cus_1" }
<input type="checkbox"/>	user_2	companya#2020-08-21				
<input type="checkbox"/>	user_2	companyb#2020-08-20				
<input type="checkbox"/>	user_1	2022-01-01			{ "customer_id": "cus_1" }	{ "customer_id": "cus_1" }
<input type="checkbox"/>	user_1	2022-01-02				

Since we only have 5 items in the table we see all items in the table. One scan can retrieve up to 1 MB of data.

Scans are expensive, as you'll pay for all the iterated items. Scans are slow because they will iterate the whole table. Always think about a query first, or better having your access patterns defined before creating your table, and only consider a scan as the very last option.

### Query - Retrieving Items Based on Your Access Patterns

A query is **much cheaper** because you only pay for the items retrieved. For example, if you have the `userId` you can use this ID to query your database. Let's do that in the console.

The screenshot shows the AWS DynamoDB Query interface. At the top, there's a dropdown menu labeled "Scan/Query items". Below it, a sub-menu says "Scan/query a table or index" with tabs for "Scan" and "Query" (which is selected). A dropdown menu shows "Users". Underneath, there are fields for "userId (Partition key)" containing "user\_1" and "createdAt (Sort key)" with "Equal to" selected and an input field "Enter sort key value". There's also a checkbox for "Sort descending". A "Filters" section is collapsed. At the bottom are "Run" and "Reset" buttons.

**Completed** Read capacity units consumed: 0.5

**Items returned (2)**

	userId	createdAt	subscription
<input type="checkbox"/>	user_1	2022-01-01	{"customer_id": {"S": "cus_1"}, "plan_id": {"S": "plan_1"}}
<input type="checkbox"/>	user_1	2022-01-02	

### Batch Operations - Multiple Operations In A Single Command

In most AWS APIs it is possible to do operations in a batch. Instead of a dedicated API call for every item, you would group many times into one batch call - e.g. submitting updates for 10 different items.

DynamoDB offers you the following batch operations:

- `BatchGetItem`
- `BatchWriteItem`
- `BatchExecuteStatement`

We won't go into detail about these. If you need to retrieve or write a high load always checkout if a batch operation is available.

### Scans As A Last Resort

As said before, but as another reminder: **scans are the last option**. Thoughtful access patterns, a good primary key design, and queries are the way to go.

## Extending Your Query Capabilities with Indexes

You're not locked to your initial query capabilities after creating your table. DynamoDB allows you to create indexes as additional pairs of keys. Let's define another table `Orders` to explore this feature in detail.

Partition Key			
userId	createdAt	firstname	lastname
user_1	2022-01-01	Sandro	Volpicella
user_2	2022-01-01	Tobias	Schmidt

We have a simple primary key set to `orderId`. We can now query each item by its identifier.

**But what if we want to get all orders from a specific user?**

Without an index, we'd need to scan all orders and filter the result by the `userId`. We'd then also pay for three items even if we only want to retrieve two. It's insignificant in our example but escalates quickly with increasing table sizes.

## Understanding Global and Local Secondary Indexes

With another index that settles on the `userId`, we can fulfill our query requirements. But let us quickly dive into DynamoDB's different types of secondary indexes: **global (GSI)** and **local (LSI)**. Both come with different features and requirements regarding their creation.

Global Secondary Index (GSI)			
GSI Primary Key			
orderId	productId	quantity	user
order_1	product_2	2	user_1
order_2	product_2	40	user_1
order_3	product_1	1	user_2

Local Secondary Index (LSI)			
LSI Partition Key & Table Partition Key		LSI Sort Key	
orderId	productId	quantity	user
order_1	product_2	2	user_1
order_2	product_2	40	user_1
order_3	product_1	1	user_2

- Global Secondary Indexes (GSI)** Global Secondary Indexes allow you to create a different primary key that is independent of your original one. It is called global because a query on that index will check **all data** from the table. This index lives in its own partition space.
- Local Secondary Indexes (LSI)** A Local Secondary Index can only be created **during table generation**. It is called local because all data resides within the same partition. With an LSI you need to reuse the **Partition Key of your table**. You can only define another Sort Key.

**A Small Dive Into How Partitions Work In DynamoDB** In DynamoDB, a table is divided into multiple partitions, and each partition is stored on a different server. When an item is added to the table, it is assigned to a partition based on the partition key value. All items with the same partition key value are stored in the same partition and are therefore stored on the same server. This allows DynamoDB to distribute the data across multiple servers, which helps to scale the table as the size of the data grows. If you're interested in more detail about partitions check out this amazing article by Alex DeBrie.

### Creating Indexes

Our challenge of querying **orders for a specific user can be solved by a global secondary index**. But why a global and not a local secondary index? There are two main reasons:

- We want to define a **new primary key** (`userId`). LSIs only allow you to add sort keys.
- You can generate LSIs only **during table generation**. The table is already created.

Go to your table → Indexes and click on **Create Index**

Name	Status	Partition key	Sort key	Read capacity	Write capacity	Projected attributes	Size	Item count
No global secondary indexes Global secondary indexes allow you to perform queries on attributes that are not part of the table's primary key.								

Doing this will open a small guide for creating an index:

**Index details** [Info](#)

<b>Partition key</b>	<b>Data type</b>
<input type="text" value="userId"/>	String ▾
1 to 255 characters.	
<b>Sort key - optional</b>	<b>Data type</b>
<input type="text" value="Enter the sort key name"/>	String ▾
1 to 255 characters.	
<b>Index name</b>	
<input type="text" value="byUser"/>	
Between 3 and 255 characters. Only A-Z, a-z, 0-9, underscore characters, hyphens, and periods allowed.	

Your partition key for this index needs to be an existing attribute of the table.

We want to query all orders by users. For that, we choose the `userId` as the partition key. We don't define a sort key.

The name of our index will be **byUser**. It is common practice to name the indexes like that:  
`by<ENTITY _ WE _ WANT _ TO _ QUERY>` → `byUser`.

If we want to have an index to query by the product we could create one called `byProduct`.

After we click on *Create* DynamoDB creates the index. For smaller tables, this is pretty fast. For larger tables, this can take some minutes.

In the background, DynamoDB creates its own data structure. With the next index, we can query the table more efficiently.

Without an index (or with the default one) the table looks like this:

Table PK

orderId	productId	quantity	user
order_1	product_2	2	user_1
order_2	product_2	40	user_1
order_3	product_1	1	user_2

With the new data structure, your orders table looks like that now:

byUser PK

user	orderId	productId	quantity
user_1	order_1	product_2	2
user_1	order_2	product_2	40
user_2	order_3	product_1	1

You see that your data for the `byUser` index is now accessible by the `userId` attribute.

Let's test out the query in the console. Head over to your table. Next to the query, you can now see a dropdown menu. Select the index `byUser`.

Completed Read capacity units consumed: 0.5

	orderId	product_id	quantity	userId
<input type="checkbox"/>	order_2	product_2	2	user_1
<input type="checkbox"/>	order_28	product_2	2	user_1
<input type="checkbox"/>	order_1	product_2	2	user_1

The query field changed from `orderId` to `userId`. If you enter a user id like `user_1` you will see all items connected to this user.

### Thinking About Access Patterns First

By thinking about your access patterns upfront, you can choose a primary key that supports the access patterns of your application and helps to ensure that your table is performant and scalable.

You don't need to overthink it from the beginning but put in a few thoughts about your query requirements and maybe how to achieve evenly distributed partition keys.

Most importantly: get your hands dirty fast and simply build applications. It is always possible to do migrations and set up new tables.

### Adding, Removing, or Updating Items in DynamoDB

In this section, we will use the CLI for the first time. We felt the need to explain mutating items in more detail since some concepts are hard to grasp.

## Expression Attributes - Mutating Items In DynamoDB

If you want to insert data into DynamoDB you pass **Expression Names & Expression Values**. DynamoDB uses them to ensure you don't use any reserved variables like `BACKUP`, `ANY`, or `TYPE`. There are more than 570 of these keywords. Expression attributes are placeholders for the key and value of your data. Let's see an example with the CLI. We want to update one item in our database.

```
aws dynamodb update-item \
  --table-name Orders \
  --key '{"#pk":{"S":"pk"}}' \
  --update-expression "SET #quantity = :quantity"
  --expression-attribute-names '{
    "#pk": "orderId",
    "#quantity": "quantity",
  }' \
  --expression-attribute-values '{
    ":pk": "order_1",
    ":quantity": "3",
  }'
```

The expression attribute names start with a `#` and the expression attribute values with `:`.

- The key definition is applied via `{ "#pk": { "S": "pk" } }`. `#pk` is mapped to `orderId` in the block of `--expression-attribute-names`.
- The same thing applied to `#quantity`.

We map the values in the same way only in the block `--expression-attribute-values`. Remember that **expression attributes** are like variables.

- Expressions Names start with `#`
- Expressions Values start with `:`

## Inserting Items

First, let's insert items into DynamoDB. There are many ways to insert data into DynamoDB. You can use the web console, the AWS API, or the CLI. We will use the CLI.

For inserting items you need to define the primary key. All other attributes are optional since

DynamoDB is schemaless.

Let's insert a new order.

```
aws dynamodb put-item \
--table-name Orders \
--key '{ "#pk": { "S":":pk" } }' \
--item file://item.json \
--expression-attribute-names '{ "#pk": "orderId" }' \
--expression-attribute-values '{ ":pk": "order_1" }'
```

We've split the key definition and the actual content of the entry (found in `items.json`).

```
{
    "orderId": { "S": "order_1" },
    "userId": { "S": "user_1" },
    "productId": { "S": "product_2" },
    "quantity": { "N": "2" }
}
```

## Removing Items

Removing items in DynamoDB only requires the primary key:

```
aws dynamodb delete-item \
--table-name Orders \
--key '{ "#pk": { "S":":pk" } }' \
--expression-attribute-names '{ "#pk": "orderId" }' \
--expression-attribute-values '{ ":pk": "order_1" }'
```

## Updating Items

If you want to update items in DynamoDB you always need to pass an **Update Expression**.

Update expressions have their own syntax. The main four functions are:

<b>Set</b>	Adds one or several attributes to an item
<b>Remove</b>	Removes one or more attributes from an item
<b>Add</b>	Updating numbers or Sets.
<b>Delete</b>	Removes one or more elements from a set.

Let's start with an example that will update `quantity` in our table :

```
aws dynamodb update-item \  
  --table-name Orders \  
  --key '{ \"orderId\": { \"S\":\"order_1\" } }' \  
  --update-expression "SET quantity = 3"
```

Update expressions are very powerful as they help to avoid race conditions (concurrent processes trying to update the same item and by that causing lost updates) by only updating a subset of fields of your entry.

### DynamoDB's Capacity Modes: On-Demand for Unpredictable Traffic - Provisioned for Predictable Traffic

DynamoDB offers two different capacity modes: **On-Demand** and **Provisioned**. For **on-demand**, there's no need how many reads and writes you'll need per second, as it will scale immediately. **Provisioned capacity** requires you to know your traffic patterns, at a steady level (that can also be scaled via CloudWatch, but much slower) of available read and write capacity.

In general, these capacity modes define two things.

1. Your bill: on-demand capacity is **way more** expensive.
2. The possibility is that your request can get throttled. Throttling means your requests will be rejected by DynamoDB via a `ThrottlingException`.

### Read And Write Capacity Units

DynamoDB charges you based on **Read Capacity Units (RCU)** and **Write Capacity Units (WCUs)**.

One read capacity unit refers to **one strongly consistent read** or **two eventually consistent reads** per second. This read can be for an item with a size of **up to 4 KB**. If the item has more than 4 KB you will consume more RCUs - e.g. 5 KB will consume 2 RCUs.

Additionally, DynamoDB allows you to use **eventual** or **strong consistent reads**. The latter avoids the possibility of reading outdated data.

Consistency Level	Description	Pro	Con
-------------------	-------------	-----	-----

<b>Consistency Level</b>	<b>Description</b>	<b>Pro</b>	<b>Con</b>
<b>Eventual consistent read</b>	It May not reflect very recent changes	- Fast - Cheap	- Not strong consistent
<b>Strong consistent read</b>	Reflects all changes	- Always consistent	- More expensive - Slower

A write capacity unit refers to one write per second for items up to 1 KB in size. For example, if you define 5 WCUs for your table you can write 5 items with a maximum size of 1 KB into your table **per second**. Dynobase offers a great Capacity Calculator for this.

#### **On-Demand Capacity For Unknown Inconsistent Traffic Patterns**

The On-Demand capacity mode doesn't require you to define any WCU or RCU. This is a good choice if you fulfill at least one of the following conditions:

- Traffic patterns are unknown and vary greatly
- You don't want to monitor and manage write & read access
- It's a table for an application under development, saving you upfront costs

On-Demand will cover almost any load, as the service limits are immense.

#### **Provisioned Capacity Predictable Traffic Patterns**

Provisioned capacity is up to 7 times cheaper than on-demand, but requires you to define RCUs and WCUs. Those will be billed, regardless if you actually use them.

Use this mode for predictable traffic. It doesn't need to be steady as you can scale RCUs and WCUs with auto-scaling policies.

#### **When to use what - A Summary To Remember**

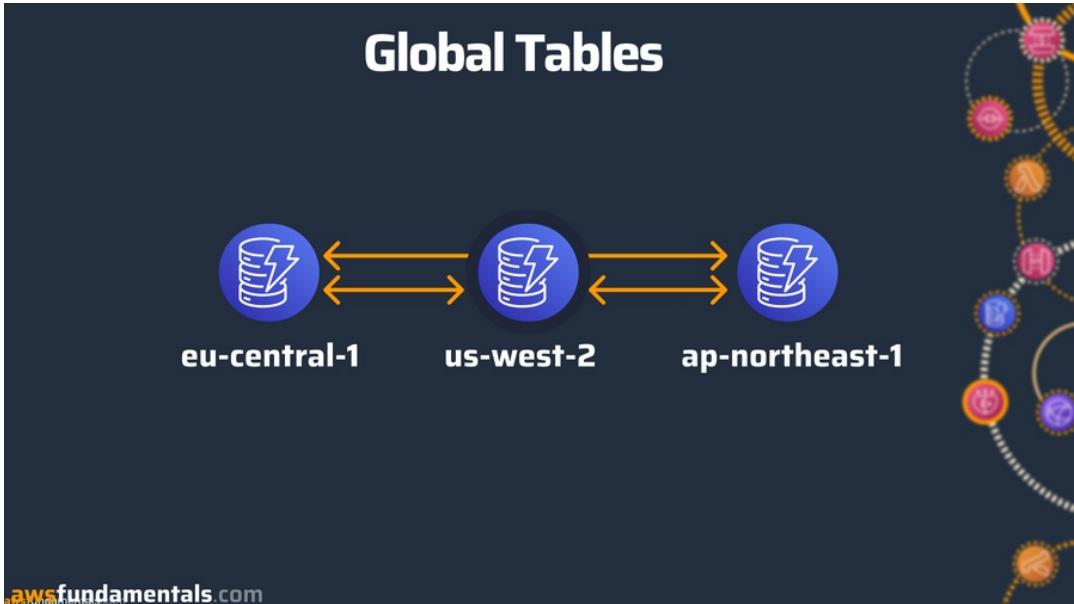
- **Variable, unpredictable traffic** → On-Demand
- **Variable, predictable traffic** → Provisioned with Auto Scaling
- **Steady, predictable traffic** → Provisioned with Reserved Capacity

**Our Suggestion:** Don't overthink this from the beginning. Use provisioned capacity with low

RCUs and WCUs until you reach the Free Tier limits (25 RCUs/WCUs per month). Afterward, chose on-demand.

### Global Tables Create Synchronized Tables Across Regions

With DynamoDB's global table feature, you can synchronize tables across regions easily, increasing resiliency and following the patterns of the Well-Architected Framework of AWS.



Data is not only backed up to another region but has also a **bi-directional** synchronization. Regardless of the write region, each region within the global table definition will receive all updates.

Even though this feature is managed, be careful. Synchronization takes time and concurrent modifications on the same items in different regions can cause integrity issues.

### Backing-Up Data With Its Built-in Feature Set

DynamoDB offers a fully-managed backup solution. Complicated processes of backing up or restoring data are a thing of the past.

<b>On-Demand Backups</b>	Trigger backups manually or via a scheduled event
<b>Continuous Backups</b>	Point in Time recovery. Your backup will be done automatically and you can restore data to the last 35 days.

<b>Exporting Backups to S3</b>	Export all of your data to S3. You can restore it by importing the backup to a new table.
--------------------------------	---

In general, DynamoDB differentiates if you use the **AWS Backup** service or if you use the direct **backup functionality of DynamoDB**.

#### **AWS Backup - The Global Backup Service By AWS**

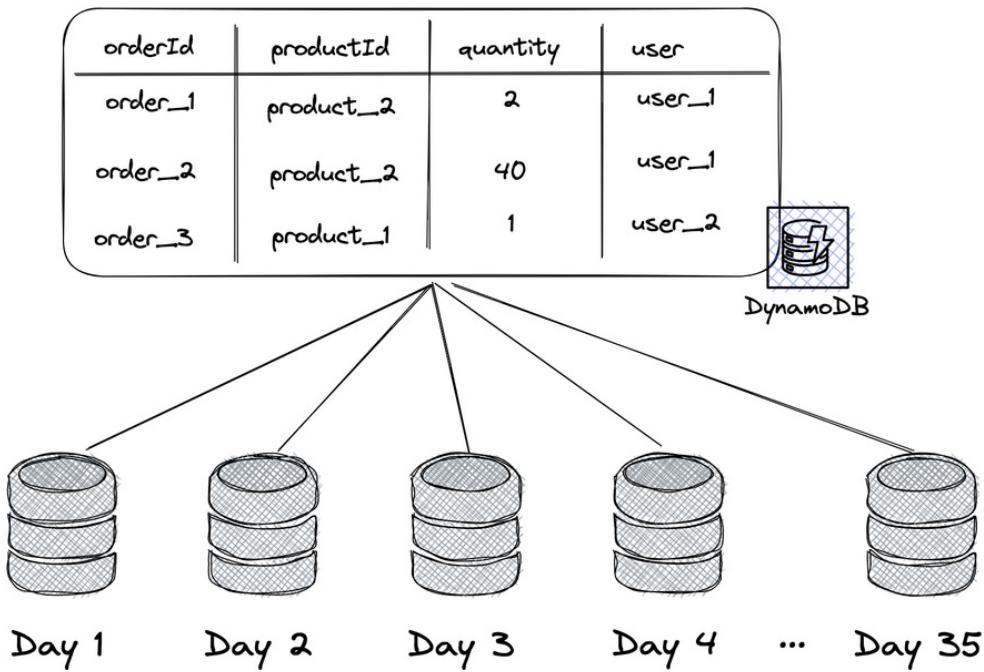
AWS Backup is AWS's global backup service, which does not only cover DynamoDB tables. Its feature set includes amongst others:

- Scheduled backups
- Cross-account, cross-region copying
- Cold storage tiering
- Encryption

We only focus on DynamoDB's built-in backup functionality.

#### **DynamoDB Backup - The Built-In Backup Mechanism of DynamoDB**

DynamoDB's integrated backup functionality is capable of **Continuous, On-Demand, and Point-In-Time backups**. Tables with enabled Point-in-Time allow you to reset it to any point in the past of the last 35 days.



Point-In-Time recovery can be activated by going to your database table, to the tab **Backup**, and ticking the box **Enable point-in-time-recovery**.

DynamoDB > Tables > Orders > Edit point-in-time recovery settings

Edit point-in-time recovery settings

**Point-in-time recovery (PITR)** Info

Point-in-time recovery provides continuous backups of your DynamoDB data for 35 days to help you protect against accidental write or deletes. Additional charges apply. See [Amazon DynamoDB pricing](#)

Enable point-in-time-recovery

Cancel **Save changes**

### Restoring Point in Time Backups

Restoring is a quick and easy task. Go to your backup console, click on **Restore** and specify the time you want to restore to and a table name. Remember that restores will always be done **into a new table**. Therefore it's important to enable your application to switch between tables easily.

**Restore settings**

**Name of restored table**  
This name will identify your restored table.

Between 3 and 255 characters in length. Only A-Z, a-z, 0-9, underscore characters, hyphens, and periods allowed.

**Point-in-time recovery**  
Specify the date and time in the last 35 days from which you would like to restore data.

Latest - August 21, 2022, 12:30:27 (UTC+02:00)

Specify date and time (UTC+02:00)

Any date and time after August 20, 2022, 17:11:32 (UTC+02:00)

**Secondary indexes**

Restore the entire table  
Your restored table will include all local and global secondary indexes.

Restore the table without secondary indexes  
Your restored table will exclude all local and global secondary indexes. Restoring this way can be faster and more cost efficient.

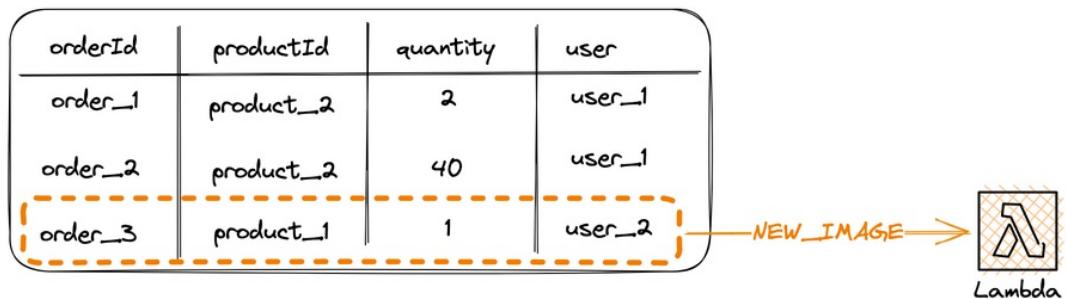
## On-Demand Backups - The Cheaper Alternative

On-demand backups can be done manually in the console tab (**Backup > Create Backup**) or you can trigger them for example with an EventBridge rule and a Lambda function. Each backups will also be saved in DynamoDB.

The screenshot shows the AWS DynamoDB console with the 'Backups' tab selected. At the top, there's a section for 'Point-in-time recovery (PITR)' with status 'Enabled', earliest restore point on August 20, 2022, and latest restore point on August 21, 2022. Below this is a 'Backups (0)' section with a 'Create backup' button highlighted. A dropdown menu for 'Create on-demand backup' and 'Schedule backups' is open. The main area shows a table with columns: Name, Status, Creation, ARN, and Size. A message at the bottom says 'No backups' and 'Create a backup to save your data.'

Starting with activating **Point in Time Recovery** (Continuous Backups), as it's a safe and easy choice that doesn't require additional steps.

### DynamoDB Streams To Trigger Lambda Functions On Changes



With DynamoDB streams, you can invoke Lambda functions for item operations in DynamoDB. As an example, we want to send a confirmation email to the user when a new order is saved.

You can activate streams in the DynamoDB console by going to the tab **Exports and Streams**.

The screenshot shows the AWS DynamoDB console interface. The top navigation bar has tabs: Overview, Indexes, Monitor, Global tables, Backups, and Exports and streams (which is highlighted with a blue border). Below the tabs, there's a section titled "Exports to S3 (0) [Info](#)". It says "Showing all export jobs from the last 90 days." There's a search bar with "Find exports" placeholder text and a pagination area with "1". A large button labeled "Export to S3" is prominent. Below this is a table header with columns: Export ARN, Destination S3 bucket, Status, and Start time (UTC+02:00). The body of the table says "No exports".  
  
The next section is titled "Amazon Kinesis data stream details". It explains that Amazon Kinesis Data Streams captures item-level changes in your table and replicates them to a Kinesis data stream. It includes an "Enable" button and a status indicator "Disabled".  
  
The final section is titled "DynamoDB stream details". It explains that capturing item-level changes in your table and pushing them to a DynamoDB stream allows you to access change information through the DynamoDB Streams API. It includes an "Enable" button and a status indicator "Disabled".

Once you enable streams you have the option of which data you want to pass to your Lambda function.

**DynamoDB stream details**

Capture item-level changes in your table, and push the changes to a DynamoDB stream. You then can access the change information through the DynamoDB Streams API.

**View type**

Choose which versions of the changed items you would like to push to the DynamoDB stream.

**Key attributes only**  
Only the key attributes of the changed item.

**New image**  
The entire item as it appears after it was changed.

**Old image**  
The entire item as it appears before it was changed.

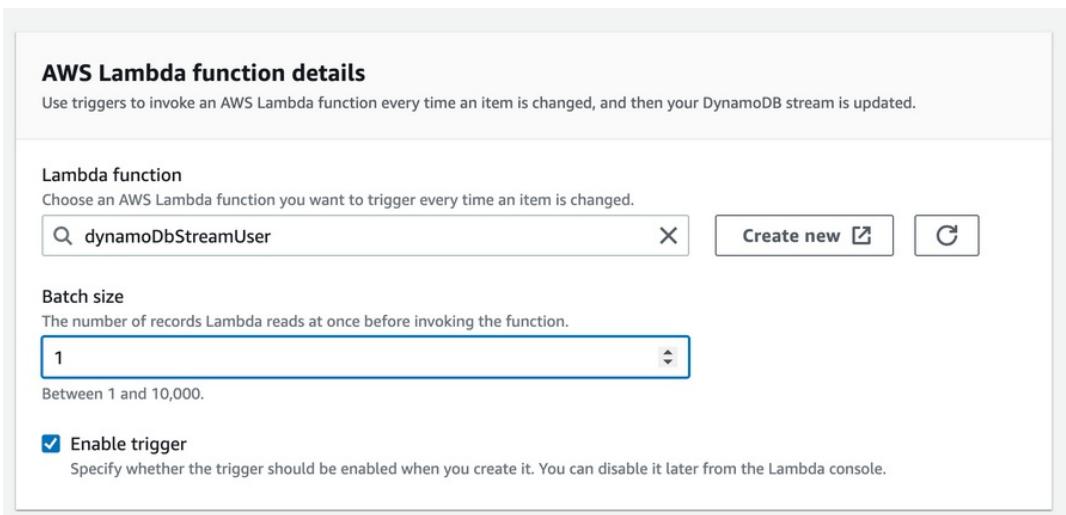
**New and old images**  
Both the new and old images of the changed item.

[Cancel](#) [Enable stream](#)

There are different options that define what will actually be passed to your function.

Type	Description	For example when inserting a new user
<b>Key attributes only</b>	Only the key of the changed item	<code>userId</code>
<b>New Image</b>	The whole new inserted item after it was changed.	The whole user object
<b>Old Image</b>	The image before it was changed.	For adding a new user that doesn't exist because it is defined as "before it was changed".
<b>New and old images</b>	Both the whole new object after it was changed and the object before it was changed.	For inserting a new user this would be a completely new user.

After enabling streams, we need to connect a Lambda function via **Create Trigger**.



You can define a **Batch size** that reduces the costs. For our example, that is not needed.

Remember that your Lambda function needs to have permission to consume this stream. This can be configured within your function's configuration tab under Permissions > Role. Simply add a new policy that allows Lambda to work with the streams of our table:

```
{  
  "document": {  
    "Version": "2012-10-17",  
    "Statement": [  
      {  
        "Effect": "Allow",  
        "Action": [  
          "dynamodb:DescribeStream",  
          "dynamodb:GetRecords",  
          "dynamodb:GetShardIterator",  
          "dynamodb>ListStreams"  
        ],  
        "Resource": "arn:aws:dynamodb:<REGION>:  
<ACCOUNT_ID>:table/Orders/stream/*"  
      }  
    ]  
  }  
}
```

After you activated the stream you will now see an incoming event for every item you've added

to DynamoDB. The incoming payload will contain the whole new entry if we selected `NEW_IMAGE` at our stream's configuration.

```
{
  "Records": [
    {
      "dynamodb": {
        "ApproximateCreationDateTime": 1667111662,
        "Keys": {
          "orderId": {
            "S": "oder_3"
          }
        },
        "NewImage": {
          "orderId": {
            "S": "oder_3"
          }
        },
        "StreamViewType": "NEW_IMAGE"
      }
    }
  ]
}
```

### Time-To-Live To Automatically Expire Items

By adding a Time-To-Live (TTL) attribute, DynamoDB can automatically delete items for you. The TTL attribute needs to be an epoch timestamp and will set the expiry date of the item. DynamoDB removes expired entries **within 48 hours**, so this is **not a real-time operation**.

TTL can be enabled within the **Additional Settings** tab.

## Enable Time to Live (TTL) Info

### TTL settings

#### TTL attribute name

The name of the attribute that will be stored in the TTL timestamp.

Between 1 and 255 characters.

### Preview

Confirm that your TTL attribute and values are working properly by specifying a date and time, and reviewing a sample of the items that will be deleted by then. Note that preview may show only some of the relevant items.

#### Simulated date and time

Specify the date and time to simulate which items would be expired.

August 22, 2022, 12:58:47 (UTC+02:00)

ⓘ Enabling TTL can take up to one hour to be applied across all partitions. You will not be able to make additional TTL changes until this update is complete.

If you've never worked with epoch, there are online calculators that can help you. Let's use them to calculate an epoch time for the 22nd of August: 1661166039. Now, let's add this to one of our entries.

<span> ⓘ Completed</span> Read capacity units consumed: 2				
<b>Items returned (1)</b>		<input type="button" value="C"/>	<input type="button" value="Actions ▾"/>	<input type="button" value="Create item"/>
userId	createdAt	firstname	lastname	ttl (TTL)
123	2022-08-20	Sandro	Volpicella	<u>1661166039</u>

We can now see that the field indicates that Time-To-Live is activated as it's extended by (TTL) and the actual values are underlined, offering a hover with information:

UTC	X
22 August 2022 11:00:39 UTC	
Local	
22 August 2022 13:00:39 CEST	
Region (Frankfurt)	
22 August 2022 13:00:39 CEST	
:cella	16611 <input type="button" value=""/>

If you want to check which data will be removed at what time, you can go back to your TTL settings and select **Run Preview**.

**Preview Time to Live** X

Confirm that your TTL attribute and values are working properly by specifying a date and time, and reviewing a sample of the items that will be deleted by then. Note that preview may show only some of the relevant items.

**TTL attribute name**  
The name of the attribute that will be stored in the TTL timestamp.  
  
Between 1 and 255 characters.

**Simulated date and time**  
Specify the date and time to simulate which items would be expired.  
 (Epoch time value: 1661166240)

Items to be deleted (1)				
<input type="text" value="Find items"/>				
userId	createdAt	firstname	lastname	ttl (TTL)
123	2022-08-20	Sandro	Volpicella	1661166039

TTL is a really life-save for getting rid of unused and temporary data.

### DynamoDB Encrypts Data at Rest on the Server

DynamoDB encrypts all your data at rest by default. This means your data on the actual server is encrypted. By default, it uses a KMS key managed by AWS but you can choose to use your own key.

## Use Cases from the Real World

DynamoDB can be used for every application that needs to store data. Let's see three common examples.

### Use Case 1: Building a High-Performance and Scalable Application

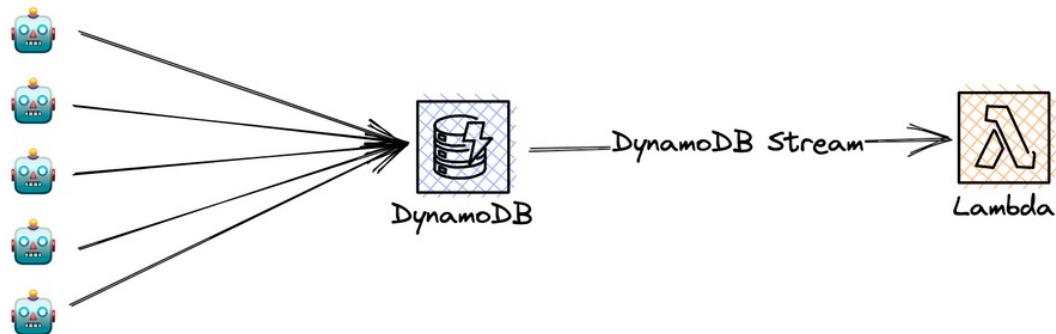
A very common use case for DynamoDB is building high-performant, scalable applications. NoSQL databases are really good at performing at a high load. With DynamoDB you don't need to take care of any scaling issues if your indexes are correctly set.

Let's take a social media platform as an example. DynamoDB allows you to store user data and activity logs. Allowing millions of users to interact with the database without any downtime.

Peak traffic is met automatically and you don't have to take care of any scaling policies due to the nature of usage-based pricing.

### Use Case 2: Handling Large Amounts of Data for IoT Sensors

Another use case for DynamoDB is storing and processing a large amount of data. DynamoDB is very well suited for such use cases since it is optimized for OLTP (Online Transactional Processing) workloads. DynamoDB uses multiple partitions (based on your keys) to access your data fast.



This makes it a great choice for a database for the Internet of Things (IoT) applications. IoT applications include a large number of sensors that are constantly sending data to a database. The number of sensors can go into the millions. DynamoDB's streaming capability makes it a great choice for processing data in near real-time and sending it to further analytics services.

### **Use Case 3: Powering Serverless Applications**

DynamoDB's pricing is completely based on your usage. This makes it an amazing choice for getting started with serverless applications. If your application doesn't have many writes or read requests the database will be free.

Once your application takes off DynamoDB will scale with it. Its fully managed nature and support for horizontal scaling make it the ideal choice for powering serverless applications. You can start your serverless applications in a risk-free way without paying any more.

### **Tips for the Real World**

The following are some quick tips you should follow when working with DynamoDB:

- **Think about your access patterns first** - If you design your database don't start by thinking about your keys and attributes first. But start thinking about your access patterns. If you build a project management tool your access patterns could look like the following:  
⇒ Get users by user ID ⇒ Get users by project ⇒ Get projects by user Note down the access patterns and discuss them with your development team. Use the NoSQL workbench to model all Indexes.
- **Always prefer queries over scans** - always think about how you can leverage a query over a scan. Scans won't scale and are very expensive
- **Make use of global tables** - global tables allow you to create a single table that is replicated to multiple AWS regions. Use this feature right from the beginning
- **Activate point-in-time recovery** - it will allow you to restore your database to points in the last 30 days.
- **Monitor usages for your tables** - use common metrics in CloudWatch and build a dashboard for DynamoDB. There are automatic dashboards for DynamoDB already available. This will help you understand how your application behaves.

### **Final Words**

DynamoDB is one of the most amazing services within AWS. Learning DynamoDB gives you superpowers for building scalable, and high-performant applications. The service is easy to start with, but not easy to master as its concepts are often hard to grasp.



Amazon **S3**

# S3 Is a Secure and Highly Available Object Storage

## Introduction

Amazon Simple Storage Service is one of the most used and known services at AWS. With S3, you can store and retrieve any amount of data at any time. S3 is not a database, but a **BLOB** (**B**inary **L**arge **O**bject **F**iles) storage for saving huge amounts of unstructured data. S3 is also one of the safest places in the world to keep data due to its exceptionally high durability.

## Buckets and Objects Are at the Core of S3

A **bucket** is like a folder on your computer where you can put files in. Only on S3 with unlimited storage and an unlimited number of files.

A bucket name needs to be **globally unique** as your bucket will receive a unique URL with the following pattern: `<name>.s3.<region>.amazonaws.com`



There are many configuration options and features for your bucket, including:

- Encryption
- Enabling public access
- Versioning of objects
- Building event-driven systems based on object operations

## **Storage Classes - Save Money for Longer Retrieval Times**

Amazon offers different storage classes depending on your availability, retrieval, and durability needs. The goal of changing a storage class is to save money. It is always a trade-off for other factors such as:

- How long does S3 need to return objects
- How durable is your data
- How long does it take to retrieve your data

Let's have a look at an overview of all storage classes:

<b>Storage Class</b>	<b>Definition</b>	<b>Price</b>	<b>Retrieval Time</b>
<b>Standard</b>	The default for new objects & buckets. Active & Frequently accessed data.	Middle	Fast
<b>Intelligent Tiering</b>	Automatic cost savings by analyzing your access patterns.	Good	Depends on the access pattern
<b>Standard - Infrequent Access</b>	Data that is accessed infrequently on a monthly basis	Low	Middle to Fast
<b>One Zone - Infrequent Access</b>	Data that is accessed infrequently <b>and</b> that is only saved in one availability zone.	Very low	Middle to Fast
<b>Glacier Instant Retrieval</b>	Data that is accessed very infrequently but has still a low retrieval time (compared to other glacier alternatives)	Low	Middle to Fast

<b>Storage Class</b>	<b>Definition</b>	<b>Price</b>	<b>Retrieval Time</b>
<b>Glacier Flexible Retrieval</b>	Long-term, low-cost storage. Mainly used for archives or database backups. Retrieval options are very long.	Low	Slow
<b>Glacier Deep Archive</b>	The cheapest storage class out there. Only for long-term data that is accessed almost never.	Very Low	Very slow - hours
<b>Outposts</b>	Outposts are on-premise servers with the installed software. S3 can also be used in your data center	Depends	Depends

These are a lot of storage classes! The good thing is you don't have to know them by heart (only if you do a certification). The main thing you need to understand is that there is **no free lunch**. This means that there is always a trade-off. If you pay less for a storage class it will come at the cost of retrieval times.

If you don't have a large amount of data or a high S3 bill, don't even think about optimizing early and just stick to the standard class.

Once your bill starts to get higher it often makes sense to start with the Intelligent Tiering class. In our experience, this works out very well and you don't have to manage anything by yourself. AWS analyses your access patterns and puts the objects into different storage classes.

### **Use Glacier for Long-Living and Infrequently Accessed Data Like Database Backups**

If you save backups or archives in S3 it makes sense to choose a Glacier storage class. Glacier has three different classes depending on your retrieval time

1. Instant Retrieval
2. Flexible Retrieval
3. Deep Archive

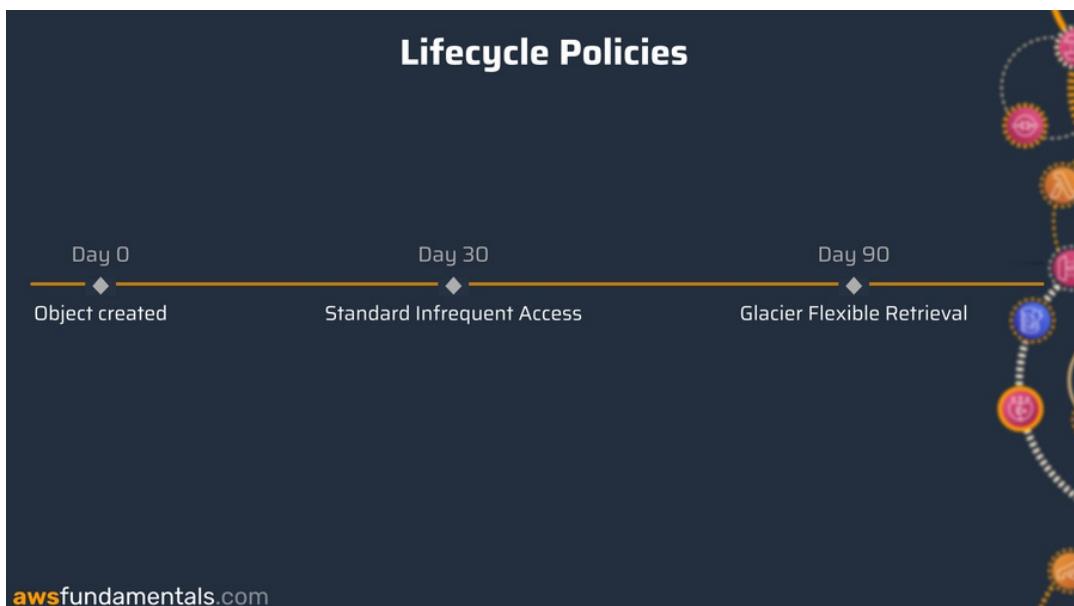
Deep Archive is a great choice for long-term backups as it's cheap but can take a few hours to

restore. If your database crashed and you need your data available **immediately** this is not a good option. Instant Retrieval or Flexible Retrieval would be a much better option.

As always: start simple and don't over-optimize prematurely. If you start optimizing think about all trade-offs out there.

### Lifecycle Policies Send Objects Automatically to a Different Storage Class

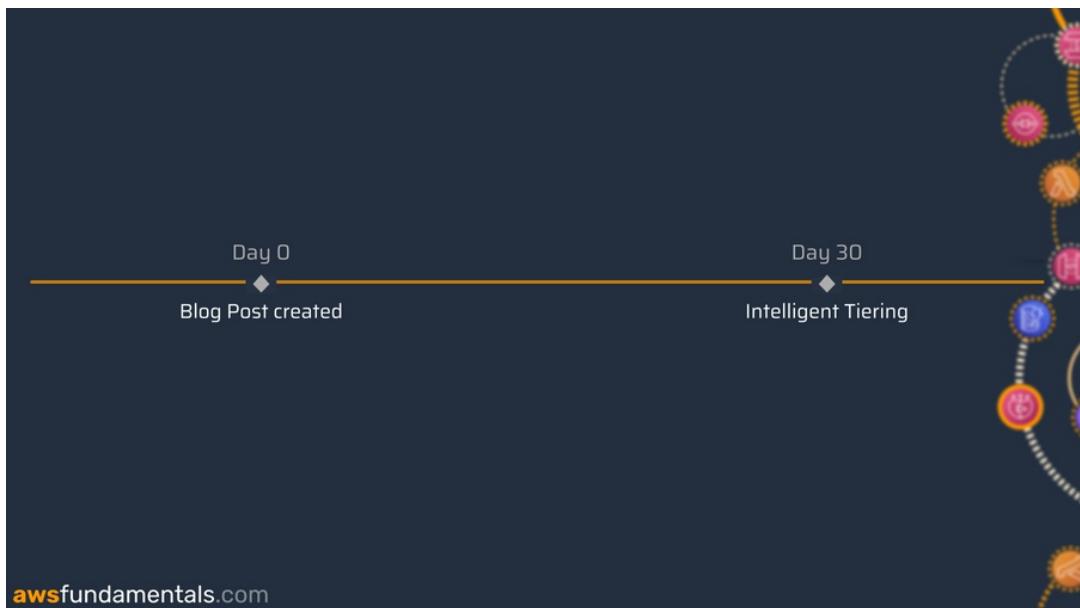
Lifecycle policies let you send data to a **different storage class after a defined time**. You can also remove objects after a certain time with policies.



Let's take the example of audio generation for our blogs. The main access on a blog post often happens in the first few days or weeks after publishing. At that time the retrieval time should be fast. The MP3 file in the bucket should stay in the **standard storage class**.

After the post has lost most of its visibility, it can live in a cheaper storage class. This is what a lifecycle policy can take care of. It would be even better to use **Intelligent Tiering** for this case. AWS can figure out the access pattern and moves it across storage classes. If traffic spikes again, e.g. due to search engine indexing, intelligent tiering can also take care of moving it up in the storage class hierarchy.

To enable this use case we define a lifecycle policy that moves all objects to Intelligent Tiering after 30 days of creation.



### Creating Lifecycle Rules in the Management Console

Let's take steps to create our first lifecycle policy via the management console by jumping to our bucket's management tab and selecting **Create Lifecycle rule**.

There we can define the name of the rule, which objects should be taken into consideration (e.g. we only want objects of a certain pre- or suffix), and the action that should be taken.

#### Lifecycle rule configuration

Lifecycle rule name  
AudioBlogMove  
Up to 255 characters

Choose a rule scope  
 Limit the scope of this rule using one or more filters  
 Apply to all objects in the bucket

**⚠️ Apply to all objects in the bucket**

If you want the rule to apply to specific objects, you must use a filter to identify those objects. Choose "Limit the scope of this rule using one or more filters". [Learn more](#)

I acknowledge that this rule will apply to all objects in the bucket.

**Lifecycle rule actions**

Choose the actions you want this rule to perform. Per-request fees apply. [Learn more](#) or see [Amazon S3 pricing](#)

- Move current versions of objects between storage classes
- Move noncurrent versions of objects between storage classes
- Expire current versions of objects
- Permanently delete noncurrent versions of objects
- Delete expired object delete markers or incomplete multipart uploads

These actions are not supported when filtering by object tags or object size.

**Transition current versions of objects between storage classes**

Choose transitions to move current versions of objects between storage classes based on your use case scenario and performance access requirements. These transitions start from when the objects are created and are consecutively applied. [Learn more](#)

Choose storage class transitions	Days after object creation	Remove
Intelligent-Tiering	30	<a href="#">Remove</a>

[Add transition](#)

**Review transition and expiration actions**

Current version actions	Noncurrent versions actions
Day 0 <ul style="list-style-type: none"> <li>• Objects uploaded</li> </ul> ↓ Day 30 <ul style="list-style-type: none"> <li>• Objects move to Intelligent-Tiering</li> </ul>	Day 0 No actions defined.

The last card will show you a summary of what will happen on what day.

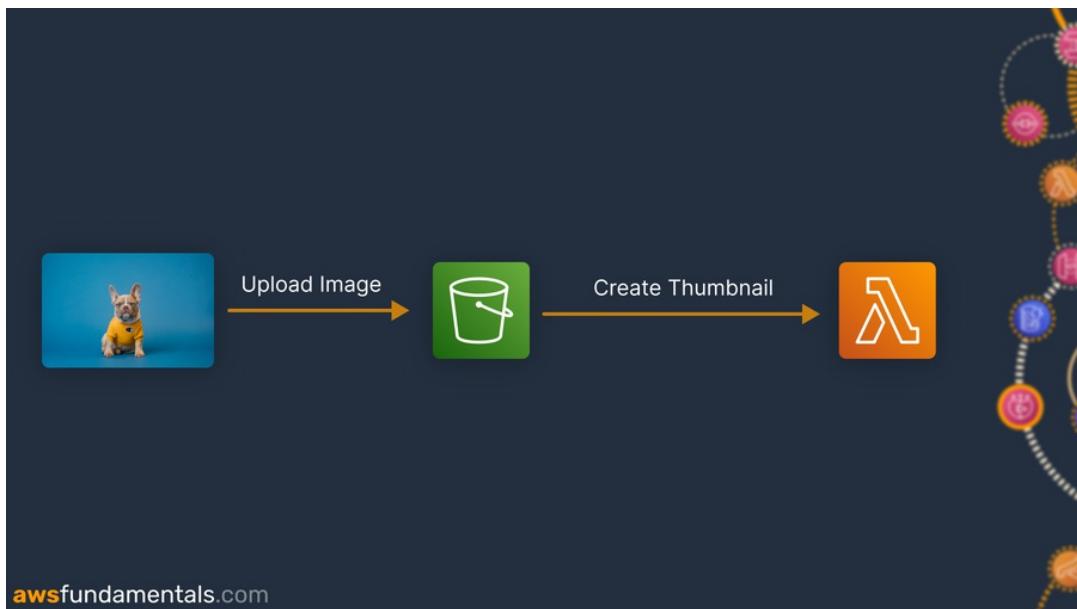
### Event Notifications - Send Events on S3 Changes to SQS or Lambda

Building event-driven systems are a **huge** part of AWS. There are whole applications that are built on top of S3's notifications engine.

One good example is image optimization. Let's take our blogging platform as an example. Users upload pictures as covers. The covers need to have different shapes and sizes. For the web, it is important to optimize the formats for different touch points.

And that's where S3 event notifications can step in. After an object was created in a bucket, it

can send an event to a Lambda function to take care of image optimization or creating thumbnails in different sizes.



#### Create Event Notifications in the Management Console

Let's create our first event notification by going to our bucket's permission tab and selecting **Event Notifications**.

We have to select which S3 operations should trigger our event, e.g. all object creations.

## Event types

Specify at least one event for which you want to receive notifications. For each group, you can choose an event type for all events, or you can choose one or more individual events.

### Object creation

- All object create events  
s3:ObjectCreated:\*

- Put  
s3:ObjectCreated:Put
- Post  
s3:ObjectCreated:Post
- Copy  
s3:ObjectCreated:Copy
- Multipart upload completed  
s3:ObjectCreated:CompleteMultipartUpload

### Object removal

- All object removal events  
s3:ObjectRemoved:\*

- Permanently deleted  
s3:ObjectRemoved:Delete
- Delete marker created  
s3:ObjectRemoved:DeleteMarkerCreated

Next, we have to select our destination. There are different choices:

1. Lambda Function → Simple code execution
2. SNS Topic → Send to more receivers (also personal ones like email or in-app)
3. SQS Queue → You need to handle retries, errors, etc.

For our example, we will choose Lambda. You can create a simple Node.js function in another tab which will just log the received input for now.

```
exports.handler = async (event) => {
    console.info('EVENT\n' + JSON.stringify(event, null, 2));
};
```

Now go back to your S3 tab and choose this Lambda function from the console

## Destination

**Before you begin** Before Amazon S3 can publish messages to a destination, you must grant the Amazon S3 principal the necessary permissions to call the relevant API to publish messages to an SNS topic, an SQS queue, or a Lambda function. [Learn more](#)

**Destination** Choose a destination to publish the event. [Learn more](#)

**Lambda function** Run a Lambda function script based on S3 events.

**SNS topic** Send notifications to email, SMS, or an HTTP endpoint.

**SQS queue** Send notifications to an SQS queue to be read by a server.

**Specify Lambda function**

**Choose from your Lambda functions**

**Enter Lambda function ARN**

**Lambda function**

Thumbnail ▾

**Cancel** **Save changes**

After saving our new event notifications, any uploads to our bucket will trigger our function. By jumping to our functions CloudWatch log stream, we'll see something like this:

```
{  
  "Records": [  
    {  
      "eventVersion": "2.1",  
      "eventSource": "aws:s3",  
      "awsRegion": "eu-central-1",  
      "eventTime": "2022-08-25T11:58:21.625Z",  
      "eventName": "ObjectCreated:Put",  
      "userIdentity": {  
        "principalId": "123"  
      },  
      "requestParameters": {  
        "sourceIPAddress": "123"  
      },  
      "responseElements": {  
        "x-amz-request-id": "123",  
        "ETag": "123"  
      }  
    }  
  ]  
}
```

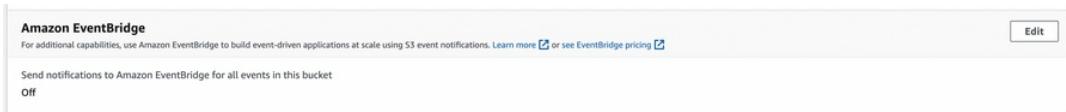
```

    "x-amz-id-2": "123"
},
"s3": {
    "s3SchemaVersion": "1.0",
    "configurationId": "thumbnail",
    "bucket": {
        "name": "audioblogexample",
        "ownerIdentity": {
            "principalId": "123"
        },
        "arn": "arn:aws:s3:::audioblogexample"
    },
    "object": {
        "key": "OG_How+to+build+Event-Driven+Systems+on+AWS.png",
        "size": 129497,
        "eTag": "b474be08285b3d1d5bf48dbcbd9b1478",
        "sequencer": "00630763DD8E8C5663"
    }
}
]
}

```

You can now execute functions on the input!

For the more advanced use cases: you can also send your data to EventBridge first and route data based on your message body.



## Server Access Logging - Log All Access to Your Bucket

Understanding what is going on in your S3 bucket can be quite challenging. Especially with bigger applications and a lot of traffic.

**Server Access Logging** allows you to understand all access and API calls to your S3 bucket. You can activate Server Access Logging in the S3 properties.

The screenshot shows the 'Server access logging' configuration page for an S3 bucket. At the top, there's a note about enabling server access logging: 'Log requests for access to your bucket. [Learn more](#)' with a blue link icon. Below this, there are two radio button options: 'Disable' (unchecked) and 'Enable' (checked). A warning message in a box states: 'Bucket policy will be updated. When you enable server access logging, the S3 console automatically updates your bucket policy to include access to the S3 log delivery group.' In the 'Target bucket' section, the URL 's3://audioblogloggingbucket' is entered into a text input field, and a 'Browse S3' button is next to it. Below the input field is a placeholder text 'Format: s3://bucket/prefix'. At the bottom right are 'Cancel' and 'Save changes' buttons.

S3 saves the logs in another S3 bucket you need to provide. The logging itself is **free**, but the **storage is charged**. Since S3 storage is pretty cheap it's recommended to always activate the logging feature unless you expect an enormous amount of operations.

### CloudTrail Data Events Give You a Granular View of API Calls in Your Bucket

Another way of logging your S3 access is CloudTrail data events. This service shows you a fine-grained view of your S3 data usage.

AWS CloudTrail is Amazon's own auditing service. CloudTrail captures internal API calls in your AWS Account. S3 offers the capability of capturing data events. This is different from normal CloudTrail API access.

API events in CloudTrail are for example the creation (`CreateBucket`) or deletion (`DeleteBucket`) of a bucket. Data events are resource operations like retrieving (`GetObject`) or deleting (`DeleteObject`) an object.

For applications that use S3 a lot, many logs can occur. Make sure to calculate this with your pricing costs because it can be very expensive.

### Bucket Policies Grant Permissions to Your Bucket and Objects

Bucket policies enable you to grant access permissions to your bucket and all objects inside of it. Policies look like typical IAM policies.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "CloudFrontAccess",
      "Effect": "Allow",
      "Principal": {
        "AWS": ["arn:aws:iam::111122223333:user/user_1"]
      },
      "Action": ["s3>ListObjects", "s3:GetObject"],
      "Resource": "arn:aws:s3:::DOC-EXAMPLE-BUCKET/*"
    }
  ]
}
```

Those resource-based policies are set on the resource itself. In our example, we're granting the list objects permission as well as the action to retrieve objects from the bucket. We allow this for a specific IAM user via the `Principal` field.

As learned previously in the IAM chapter, there are a lot more things you can do like adding conditions.

### **Batch Operations Lets You save Money by Uploading Objects in Batches**

As with other services, S3 also offers batch operations that allow you to cluster operations into a single API request. This will result in lower costs as each S3 operation (above the Free Tier contingent) is charged.

Take this into account if your customer doesn't need the result of an object upload immediately and if you have a high number of uploads.

AWS makes use of batching itself many times in its internal services. For example, in EventBridge. If you use Schema Discovery, AWS batches many requests. You can see that because you need to wait a couple of minutes until the new schema is available.

## Object Locks - Make Sure Your Data Hasn't Been Tampered With

Object locks allow you to save objects compliantly. The acronym for objects lock is **WORM** and it refers to **Write-Once-Read-Many**. Object locks are used to ensure your uploaded data wasn't tampered with by any system or developers. This is a very good fit for important log messages or backups.

Object locks have two different retention modes:

1. **Compliance** - no deletes or overwrites are possible
2. **Governance** - overwrites or deletes are only possible with specific permissions

## Keeping a Change History of Your Object with Versioning

Often, you don't want to override your data but you want to generate a new version of your S3 object. S3 allows you to version your objects. With that, you can keep older versions and you can roll back to a previous state.

Let's go back to our example of creating audio podcasts of your blog posts. With versioning, you can listen to older versions of your posts. This is also helpful if you just accidentally upload the wrong data.

Versioning in S3 is **fully managed**. You don't need to do anything other than activate the feature. No strings attached (except for maybe additional storage costs).

### Activate Versioning in the Management Console

Versioning can be activated in the Management tab of your S3 bucket.

**Bucket Versioning**

Versioning is a means of keeping multiple variants of an object in the same bucket. You can use versioning to preserve, retrieve, and restore every version of every object stored in your Amazon S3 bucket. With versioning, you can easily recover from both unintended user actions and application failures. [Learn more](#)

**Edit**

---

**Bucket Versioning**

Disabled

**Multi-factor authentication (MFA) delete**

An additional layer of security that requires multi-factor authentication for changing Bucket Versioning settings and permanently deleting object versions. To modify MFA delete settings, use the AWS CLI, AWS SDK, or the Amazon S3 REST API. [Learn more](#)

Disabled

By enabling it you can upload new versions of existing objects and see different versions of them in the console by toggling on the switch for **Show Versions**.

Objects (5)						
Objects are the fundamental entities stored in Amazon S3. You can use <a href="#">Amazon S3 inventory</a> to get a list of all objects in your bucket. For others to access your objects, you'll need to explicitly grant them permissions. <a href="#">Learn more</a>						
	Name	Type	Version ID	Last modified	Size	Storage class
<input type="checkbox"/>	FileA.png	png	3i_me3UrzNzTmALnlje3ArAtgoReOeh6	August 25, 2022, 16:31:52 (UTC+02:00)	76.8 KB	Standard
<input type="checkbox"/>	Header_.png	png	7QfMCVrNohH7a0om0DKGkXAdhoYfn_b9	August 25, 2022, 16:31:42 (UTC+02:00)	76.8 KB	Standard
<input type="checkbox"/>	Header_How to build Event-Driven Systems on AWS.png	png	KtxJksVfqm8oLAx6X_aJXnf8AjDYzo0A	August 25, 2022, 16:31:17 (UTC+02:00)	105.0 KB	Standard
<input type="checkbox"/>	Header_How to build Event-Driven Systems on AWS.png	png	null	August 25, 2022, 13:57:29 (UTC+02:00)	105.0 KB	Standard
<input type="checkbox"/>	OG_How to build Event-Driven Systems on AWS.png	png	null	August 25, 2022, 13:58:22 (UTC+02:00)	126.5 KB	Standard

Here you can see that **FileA** and **Header\_...** both have two different versions.

### Restore Old Versions by Downloading the Version and Uploading It Again

To restore the item to a previous version, download the object you want to restore and upload it again. This is not super intuitive but it is the official explanation by AWS. You can also delete the newer version of the file to have the previous available again.

### Making Your Bucket Public

Your S3 bucket is by default **not public**. Therefore, the option for blocking public access is **activated**. You can modify the option and allow your S3 bucket to be public.

#### Why Would You Want That?

There are many scenarios where it makes sense to open up your S3 bucket to the public. One example we had in the introduction already is the **hosting of static pages**. Your website should be public. Only then everybody can access it.

Another example is our audio blogs. Users should be able to listen to podcast versions of blog posts. If the users are not authenticating they need to have public access. Thus, the bucket needs to be public. You can adjust that by deactivating **Block all public access**.

You need to be careful with this of course. You don't want to open up the wrong bucket (like all your CloudTrail logs) for the public. People **will** find it.

## Pricing in S3 Is Based on Storage and API Calls

Pricing in S3 is mainly based on, but not limited to, the storage you need and the number of API calls.

Let's have a look at all the things you'll be charged for:

1. **Storage:** The prices vary a lot depending on the storage class you are using. For a standard storage class for example you pay around \$ 0.023 per GB. The pricing is tiered which means the more you use the smaller the GB fee will be but that doesn't matter that much.
2. **Requests against your buckets:** Requests are charged extra. They are also dependent on the storage class. Storage classes with an infrequent tier are much more expensive than a standard storage class. For example, for the Standard Storage class, you pay \$ 0.005 / per 1,000 requests. In the Standard Infrequent storage class, you pay \$ 0.01 / per 1,000 requests. This is much higher!
3. **Data Transfer:** One thing that is often forgotten in a cloud price calculation is the price attached to data transfer. For data transfer, it always depends if the traffic coming from the same AZ, region, CloudFront, or the internet. For example, traffic coming into S3 from the internet is free while traffic coming from S3 to the internet costs about \$ 0.09 per GB. Outgoing traffic via CloudFront is also free. This is important to consider when designing a whole cloud architecture.
4. **Advanced Features:** There are a couple of advanced features that can cost money. One example of that is Intelligent Tiering. Intelligent Tiering is the storage class that analyses your data access patterns and automatically moves your data based on the pattern in the correct storage class. The scanning can cost around \$ 0.002 / GB. If you have an application with much data and you think about switching the storage classes you need to take this into account. You can calculate if it is cheaper to get the data scanned or by leaving it in the same storage class. There are many more such nuances out there when doing cost optimization with S3.

## The Wide Fields Of Different Use Cases For S3

Amazon's flagship storage service finds its use in almost any application. We'll have a deeper look into three use cases where S3 plays a major role: automatically generating audio descriptions from text and hosting static websites.

### Use Case 1: Saving Audio Files

Imagine running a blogging platform where people can upload their blog posts. You want to offer a service where you take the text as input and transform it into a podcast format.

This diagram shows an example of architecture:



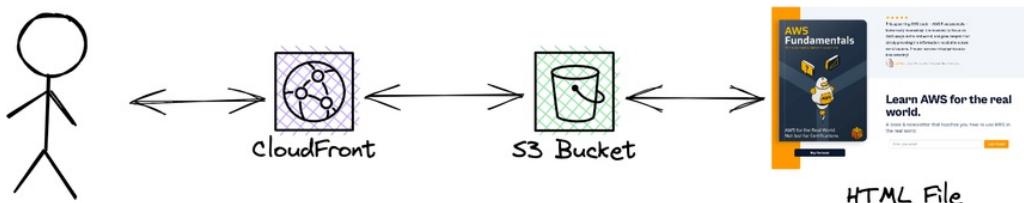
After you insert your data into DynamoDB, Lambda gets the event from a DynamoDB stream. Polly generates the text into audio. The file will then be stored in an S3 bucket.

On the frontend, you can simply embed the link to the MP3 file and make it public.

### Use Case 2: Hosting Static Websites

Another common use case is the hosting of static websites. You can combine S3 and CloudFront to serve your content over AWS's edge network. This network consists of many small servers around the world. This brings the content closer to your users.

S3 acts here as the storage and CloudFront serve the file all around the world.



If you want to host a static website in S3 you can make a bucket public, upload your HTML files, and enable static web hosting.

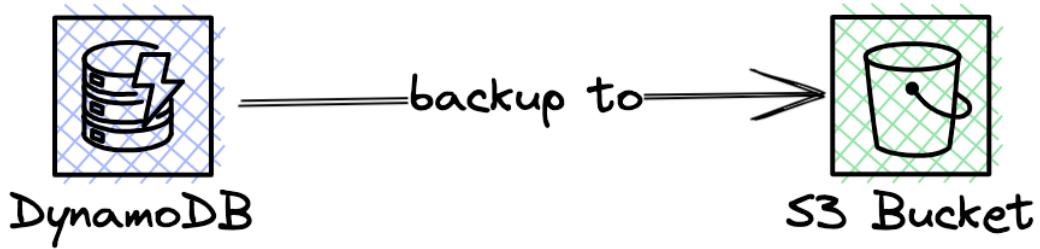
You can host websites on S3 in five steps:

1. Create Bucket
2. Upload HTML files
3. Allow public access
4. Add a bucket policy that allows `s3:GetObject` on all files

## 5. Enable website static hosting

### Use Case 3: Data Backups

Amazon S3 is an ideal solution for backing up data due to its durability and scalability. It's designed to provide 99.99999999% durability.



With its different storage classes, you can save backups really cheaply (e.g. with S3 Glacier). You can automate the process of backing up data to S3 by using AWS SDKs and the API of AWS.

### Tips for the Real World

- Enable versioning for your S3 buckets. With versioning, you are safe against **accidental deletions and overrides**.
- Configure **S3 object lifecycle policies** to archive or delete objects after a certain period of time.
- Make use of **Intelligent Tiering** if you don't have detailed specifications and access patterns
- Be sure to **block public access** (unless it is really needed).
- Use S3 **Cross-Region Replication** to store objects in several regions.
- Only activate **CloudTrail data logs** if really needed. They can get **really expensive**.

## **Final Words**

S3 is one of the most used AWS services in the world. Even non-cloud developers know what S3 is for. But most of the engineers out there have no idea about all the neat things you can do with it. By using event notifications, lifecycle rules, and proper storage classes you can build amazing applications & architectures for a very low price.

Digging deep into S3 is fun but is often not required. AWS is giving us the opportunity to use its full ability of storage classes without the need to know all details of it by activating Intelligent Tiering.



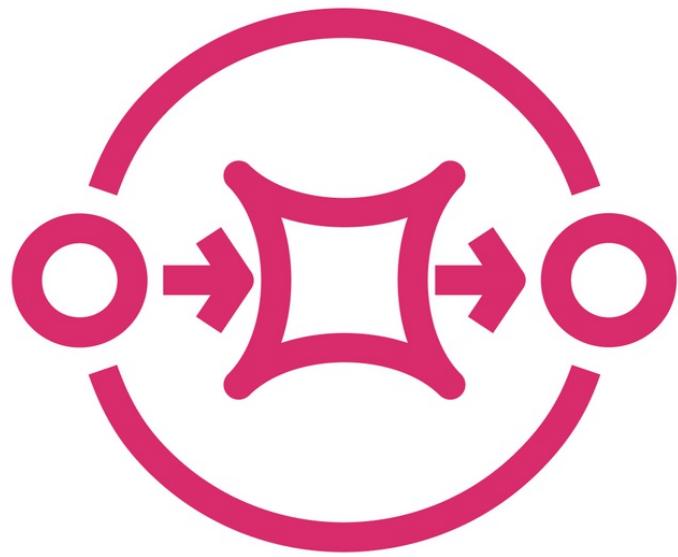
# Messaging

Messaging services are the next important cluster of services that are important for building web applications. They provide a way to send and receive messages between different components of the application, allowing them to communicate and work together.

This can be useful in situations where components need to be decoupled, such as for handling tasks that may take a long time to complete or for sending notifications to different parts of the application. Additionally, they help to make systems very resilient as they allow them to respond to incidents with reprocessing of events.

We'll have a look at three fundamental messaging services: SQS, SNS, and EventBridge. Amazon Simple Queue Service (**SQS**) is a fully managed message queuing service that makes it easy to decouple and scale microservices, distributed systems, and serverless applications. Amazon Simple Notification Service (**SNS**) is a fully managed messaging service that enables you to send messages to one or more destinations, such as email, SMS, SQS queues, and HTTP/S endpoints. **EventBridge** is a fully managed event bus service that makes it easy to connect applications together without coupling them tightly.

All three services are highly available, highly scalable, and fully managed, which makes it easy to build highly available web applications. They also support many advanced features such as encryption, dead-letter queues, and filtering to help improve the reliability and scalability of messaging in web applications. They also allow you to route and trigger the events to the right resources, making it simple to build an event-driven architecture.



Amazon **SQS**

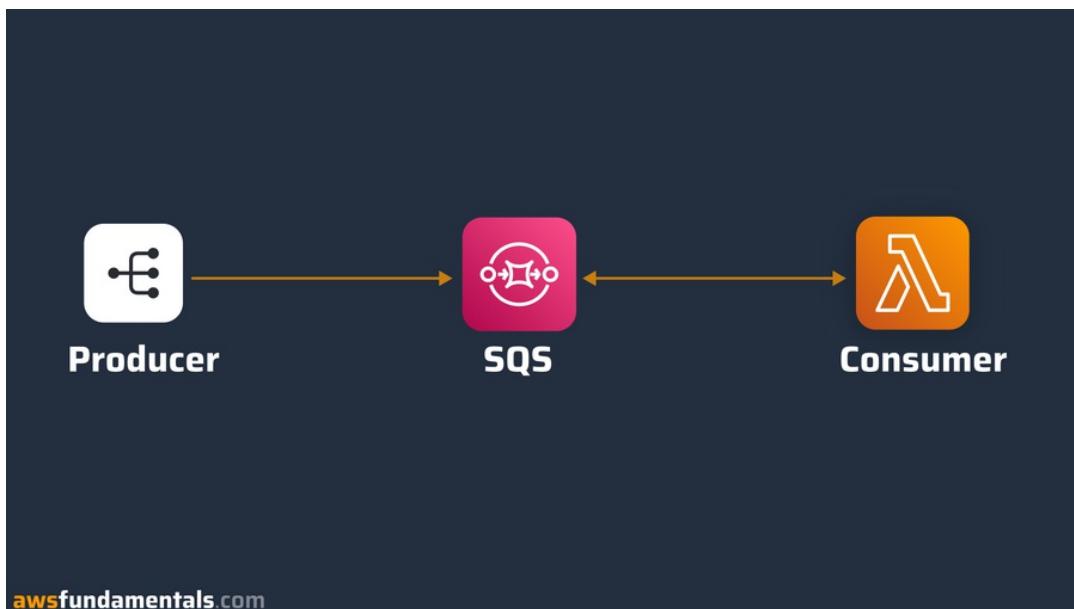
# Using Message Queues with SQS

## Introduction

Amazon Simple Queue Service (SQS) is a fully-managed message queue service by AWS. It allows you to build scalable and high-performant event-driven systems.

SQS decouples your architecture by introducing a message queue. Your message remains in the queue until a consumer picks it up and removes it.

There is almost no cloud application without SQS out there.

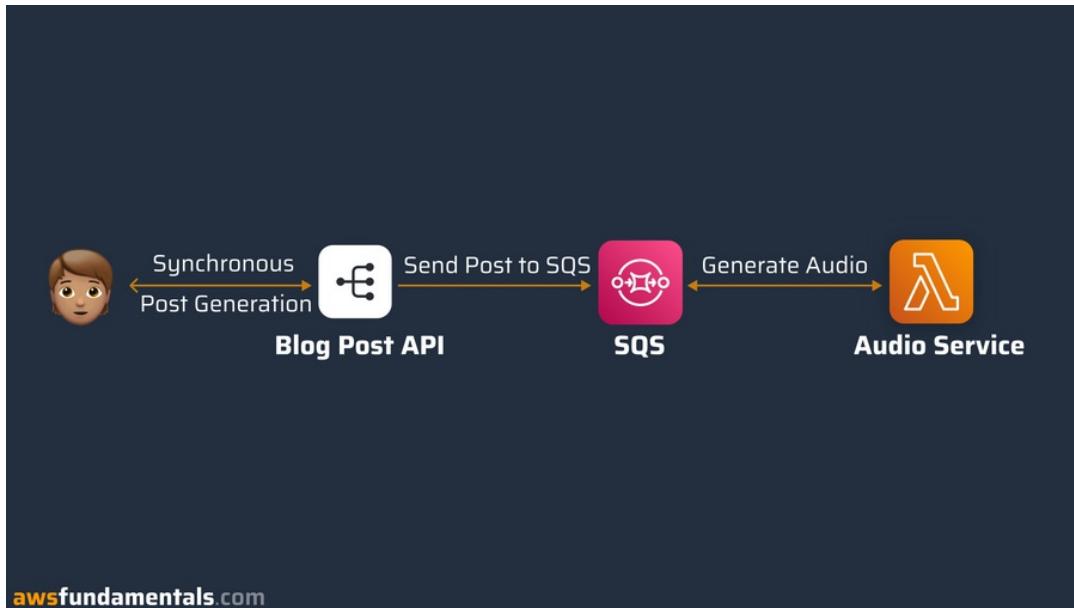


One amazing fact about SQS: It was the first service that was launched by AWS. Even before S3 and EC2. To quote the legend, Jeff Barr, here: *"We launched the Simple Queue Service in late 2004, Amazon S3 in early 2006, and Amazon EC2 later that summer."*

And one stunning statistic: at Amazon Prime Days 2021, SQS peaked at 47.7 million requests per second.

## SQS Is Poll-Based And Not Push Based

One common misconception about SQS is that it is a push-based system. The misconception often results from the tight integration with Lambda and SQS.



Let's take an example of our blogging platform. Every time a user creates a blog post we want to convert it to an audio version automatically.

In this architecture, it seems like SQS is pushing the message to the Lambda function. But actually, Lambda polls all the time for new messages. This is also one of the main cost points in SQS but later more in the cost chapter.

In AWS this is the **Event Source mapping** between SQS and Lambda. It runs in the background and looks for new SQS messages. If a new one arrives the Mapping sends it to Lambda.

So the main point to take with you: **SQS is poll-based, not push-based.**

To master and build with SQS you need to understand this fact. A push-based service would be the Simple Notification Service (SNS) which we will focus on later in the book.

**Poll-based is not a bad thing.** Indeed for many of the message queue functionalities, it is necessary. For example, for error handling and retries, it is often much easier to use poll-based queues.

### SQS Offers Standard and FIFO Queues

There are different types of queues in SQS. Standard, and FIFO queues.

The main difference is in the ordering of the messages. Standard queues don't order your messages. FIFO queues do order your messages. There are several implications with that.

We first need to understand the delivery methods in SQS and AWS in general. This is necessary to understand the differences in the queue types.

### **Exactly-Once Vs. at-Least-Once Delivery**

One of the default behaviors in cloud applications is **At-Least-Once Delivery**. At least once means that AWS guarantees you they deliver a message at least one time. It also means that consumers of an AWS SQS Standard queue can receive one single message multiple times.

This happens because of the nature of distributed cloud architecture. SQS saves your message on **several servers**. It can happen that one server is **unavailable**. If that happens during the deletion of that message, SQS sends the message again.

AWS also offers **Exactly-Once Delivery**. Exactly-once means that your consumer gets the message exactly one time. This is how many developers expect this system to work.

Exactly-once delivery is only possible with FIFO queues. FIFO queues can offer this delivery by saving a unique deduplication ID and checking if that was already processed. For standard queues, you need to build that yourself. **Take this seriously**. This is not a rare condition but happens a lot of times.

### **Idempotency Refers to Actions That Have the Same Result Even If They Were Executed Multiple Times**

For example, our audio blog generation is kind of idempotent. If we execute the call two times we will override the same file and the result doesn't change. It is **not idempotent** in terms of costs, since we call it two times the costs would be higher.

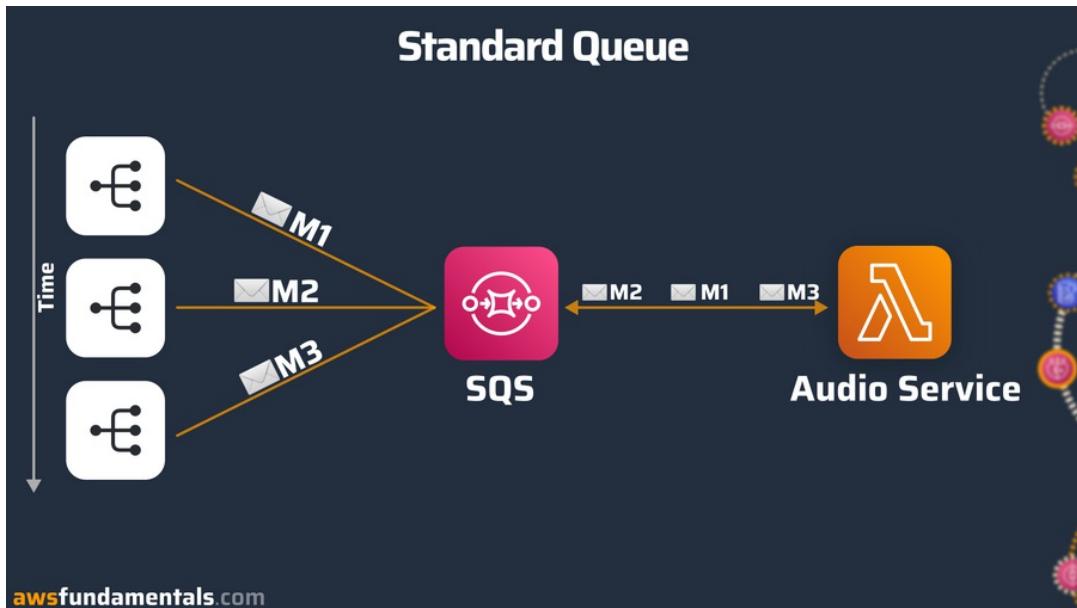
Sending a newsletter is a non-idempotent call. The result of calling the newsletter service two times is different compared to calling it only once.

To overcome this issue and make your calls idempotent you can do two things:

1. Save a unique attribute somewhere (e.g. DynamoDB) and check for every execution if the message was already processed
2. Make the call itself idempotent. For example, in audio blogs check if the generated file is already there

We won't get into details on idempotency here but please be aware of this topic if you start to implement something on AWS. Always consider that a request can appear twice.

## Use Standard Queues for Higher Throughput and High Limits

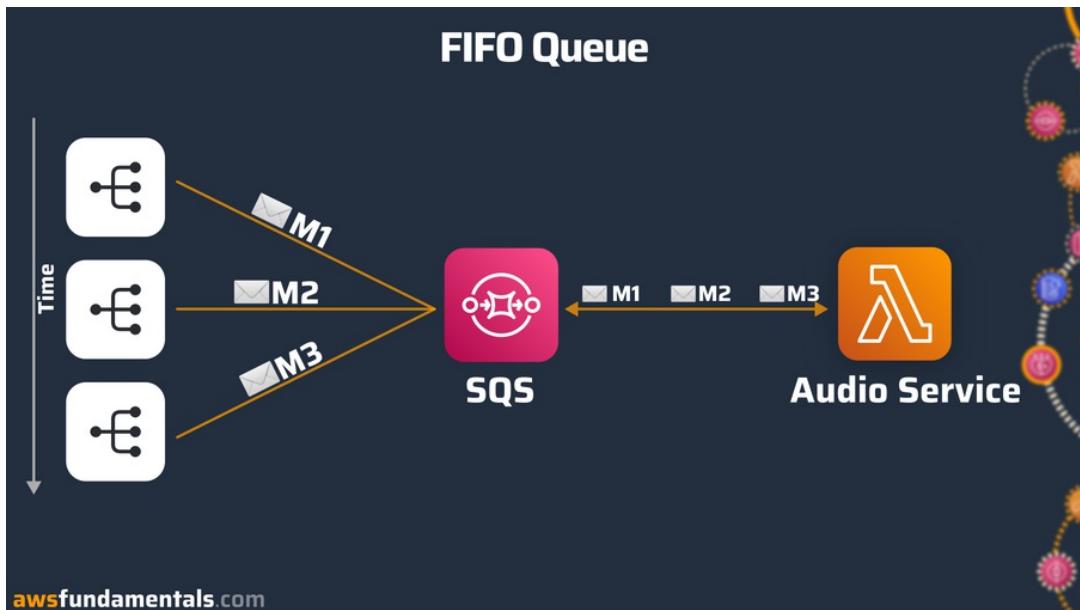


The **Standard Queue** in SQS can process a large number of messages. It follows the **at-least-once** delivery, meaning that calls can happen two or more times.

Standard Queues have much higher quotas compared to FIFO queues. For example, in standard queues, you can have 120,000 messages in flight. In FIFO queues only 20,000.

In Standard Queues, your message order does not prevail. That means a message that will be sent to the queue milliseconds after another message can be executed earlier. Keep this in mind when designing your architecture. Standard queues are also cheaper compared to FIFO queues.

## FIFO Queues Order Your Messages and Execute Them Exactly Once



FIFO refers to First-In, First-Out. This is a common principle in computer science. It indicates that the first message that will arrive in the queue is also the first message that will be picked up by the consumer. FIFO queues are keeping the order of your messages.

FIFO queues have another amazing benefit compared to standard queues. This is the **exactly-once delivery**.

With FIFO queues you don't need to handle idempotency by yourself. You can define a deduplication ID for each message. This is a unique ID that identifies your **message**. SQS checks if this ID was already processed in the last 5 minutes and rejects it if it was.

If you cannot provide a proper unique ID you can activate **Content-Based Deduplication**. This will generate an SHA-256 (a unique string) of your **message body**.

The approach to how FIFO handles idempotency can also serve as an idea of how to implement it yourself for other services or for standard queues.

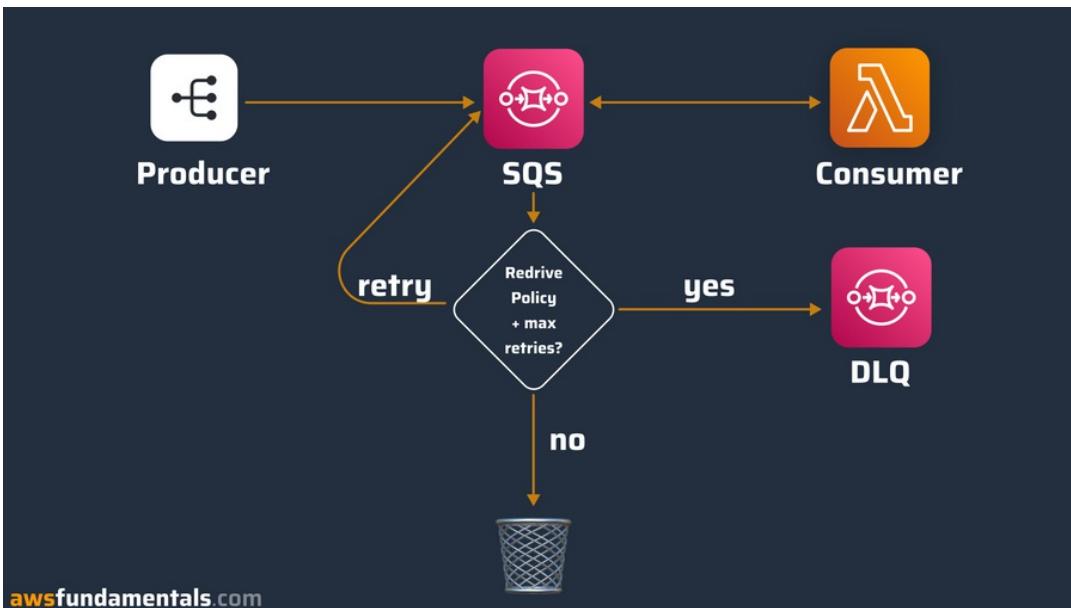
Understanding the difference between the queue types is crucial when working with SQS. You can't change the queue type after creating a queue. A major change is also within its quotas. We will focus on that in the chapter's limitations.

#### Why Shouldn't We Use Only FIFO Queues?

One of the main reasons here is cost & limitations. FIFO queues are not only more expensive per request. But FIFO queues also have much **tighter quotas**.

Batches can always only consist of 10 messages for both standard and FIFO-queues. However, standard queues have no limitation on `SendMessage` or `SendMessageBatch` API calls, whereas FIFO queues impose the limit of 3,000 messages per second: 300 requests per second via 10 batched messages. Only this limit can make a significant difference in terms of costs. If you're getting serious about developing in the cloud you need to learn how to implement idempotency in your applications. Don't use FIFO queues only for the feature of exactly-once processing.

### Use Dead-Letter-Queue for Handling Failures & Retries



A very important part of using message queues is failure handling & retries. You can attach Dead Letter Queues to your source message queue. Dead-Letter Queues (DLQ) allow you to save failed messages in another queue. You can execute a certain business logic only on these messages that failed.

#### Redrive Policies Define the Number of Retries

If you want to use DLQs you need to define a redrive policy. The redrive policy defines the number of retries. Once your queue retried the message for that threshold it sends the message to a DLQ.

#### Redriving Messages Back to the Original Queue

DLQs have the option of a redrive. Once you understood and fixed the bug you can send the data back to the original queue. SQS will process them again and you won't lose any data.

## Always Add a DLQ if You Use SQS

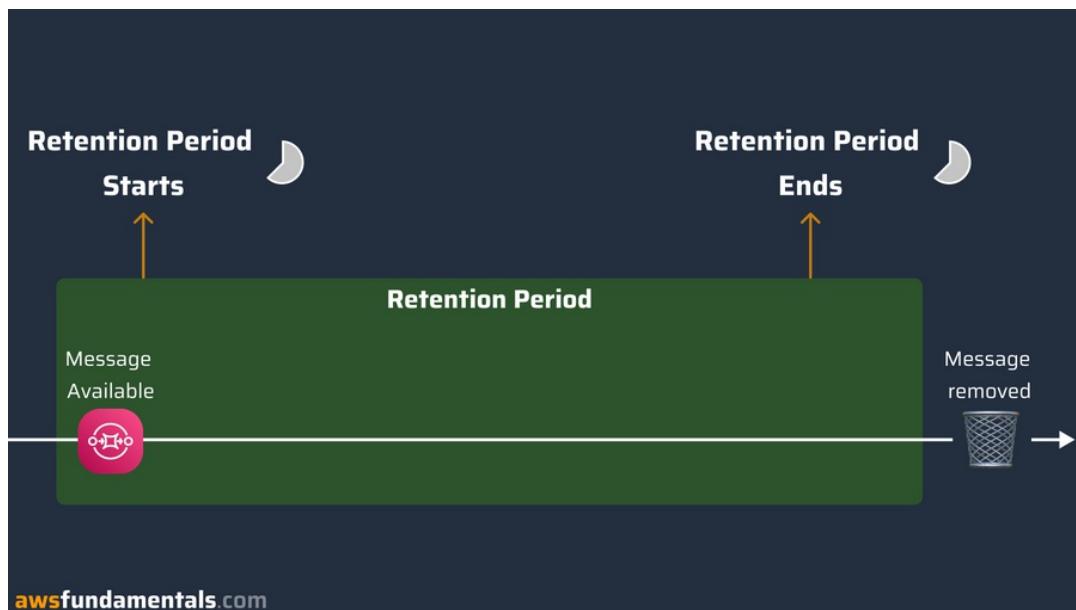
It is best practice to always add a DLQ to every SQS queue you create. Your consumer can always fail. Having a DLQ gives you insights into the reasons for failure. Typically you attach a CloudWatch alarm on your DLQ to get notified once there are messages in your DLQ. You can then go ahead and check the messages. This will give you better insights into the reasons the message failed.

If you introduce any bugs into your system the DLQ saves you from losing data.

## Configuring Polling Methods, Timeouts, and Delays

There are many parameters in SQS that are vital to understand. We'll go through the most important ones here and try to simplify them. We want to give you guidance on how to configure your queues best.

### The Retention Period Defines How Long SQS Holds Your Message in the Queue



The retention period describes how long a message remains in the queue. After that time the queue removes the message (or sends it to a DLQ). Your consumer needs to consume the message within that time and delete it.

The retention period can be as short as 1 minute and as long as 14 days. If a message reaches this threshold the queue will automatically remove it. Removal in SQS is depending on the redrive-policy. Either the queue will remove the message or send it to a DLQ. By default, the

retention period is 4 days.

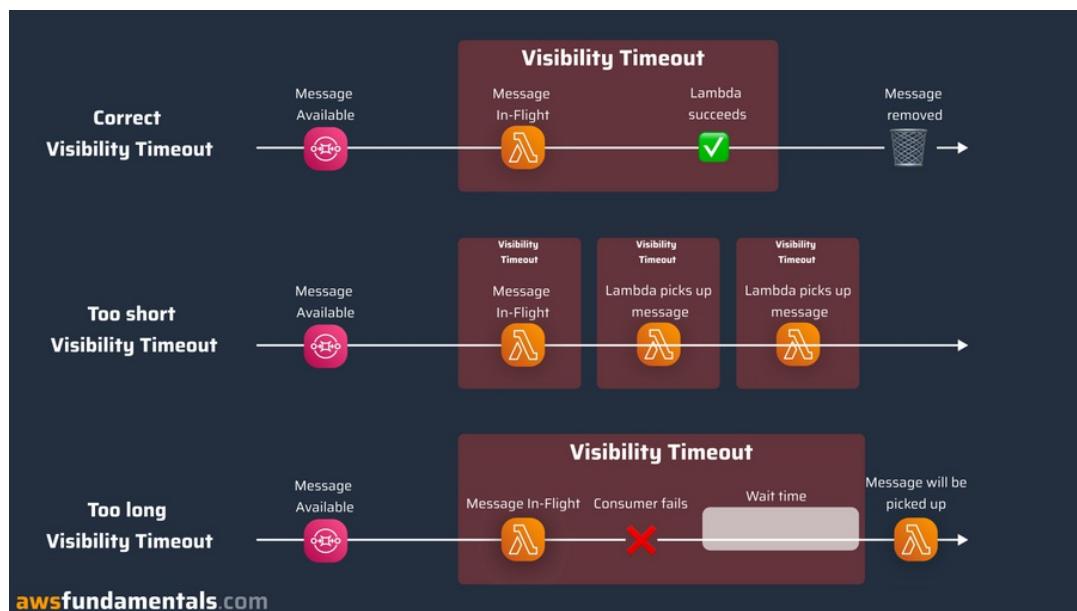
If we look at the picture above. We see that the retention period ends and no consumer picked up the message. This is often the case if you have busy consumers with a long polling interval.

### Hiding Messages From Other Consumers by Defining the Visibility Timeout

The visibility timeout defines how long your message is invisible. Once a consumer picks up a message the state of the message changes from `available` to `in-flight`. From that point on the **visibility timeout starts**.

By default, your visibility timeout is set to 30 seconds. You can set it between 0 seconds and 12 hours. Other consumers cannot pick up the same message while it is in the state `in-flight`.

Once the visibility timeout is over and the message was not deleted from the queue, it changes its state to `available` again.



Setting the correct visibility timeout is not always easy. It is important for your application to work correctly.

There are two main scenarios to consider:

1. **Visibility timeout too long** - your queue won't retry the message in a timely manner.
2. **Visibility timeout too short** - multiple consumers pick up the message

Let's see them in more detail. You can see all scenarios in the image above.

### **Visibility Timeout Is Too Short**

Imagine your consumer needs 1 minute to work on your message. After that, it removes the message from the queue. If your visibility timeout is only set to 30 seconds your message will be available too early. After 30 seconds your message changes the state from `in-flight` to `available`. Now it is visible to other consumers.

A second consumer will pick up your message and work with it. By that, your message execution is now duplicated. And even worse: after finishing the consumer is not even able to delete the message anymore, because the MessageHandle has expired. It retries until the death of the message by redrive policy.

### **Visibility Timeout Too Long**

Having a visibility timeout set too long can result in unnecessary waiting times. For example, you've set your visibility timeout to 10 minutes. Your consumer fails after 1 minute.

The message will still be invisible to other consumers for 9 more minutes. Your queue won't retry the message for the next 9 minutes. After 9 minutes the state changes from `in-flight` to `available`. Now a consumer can pick up the message again.

### **Set the Visibility Timeout rather too high than too low**

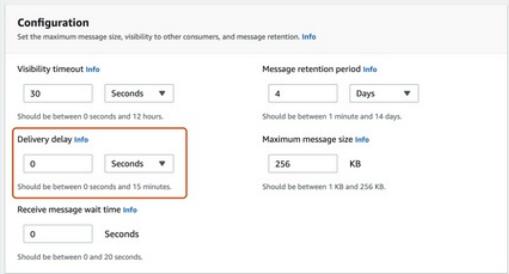
Setting the visibility timeout **too low** is in most cases **worse** than setting your timeout **too high**. But it all depends on your business context of how fast messages should be retried.

### **How should you set it?**

1. Start with some initial settings (like the default).
2. See how long your Lambda takes to finish the task
3. Set the timeout to that time + some extra time for longer executions
4. Understand how crucial it is that your retries are happening fast. Versus how bad is it if you retry only happens after a couple of minutes.

### **Delay Messages With Delivery Delay**

### Delay for the whole Queue



The screenshot shows the 'Configuration' section of an AWS SQS queue. It includes fields for visibility timeout (30 seconds), message retention period (4 days), maximum message size (256 KB), and delivery delay (0 seconds). The delivery delay field is highlighted with a red box.

### Delay per message



The screenshot shows the 'Send message' page. It has a 'Message body' input field and a 'Delivery delay info' field set to 10 seconds. The delivery delay field is also highlighted with a red box.

[awsfundamentals.com](https://awsfundamentals.com)

You can set up delays for your queue and your messages. Consumers can't pick up the messages for your defined time. Delays can have a maximum value of 15 minutes. By default, this is zero.

You can either set this value per queue or per message. Per queue means that every message in that queue will be delayed. You can also set this value per message. While sending this message you can define to delay it. In that case, only the **single message** will be delayed.

#### Long Polling and Short Polling Defines How Messages Are Polled

### Short Polling

- Receive message time = 0
- Queries a subset of SQS servers
- Large amount of empty responses
- Messages are received faster

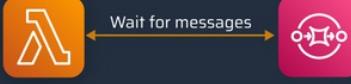
**When to use Short Polling?**  
You need the message immediately



### Long Polling

- Receive message time > 0
- Queries all servers and waits for available message
- Reduces the number of empty responses

**When to use Long Polling?**  
In all other cases



[awsfundamentals.com](https://awsfundamentals.com)

SQS is a polling-based system. This means that your consumer needs to poll the queue for new messages. SQS prices you on the number of requests.

A common scenario is that you use Lambda as your consumer. The event mapping is polling for new messages. If there are no new messages the SQS API is still called. The mapping receives an empty message (or better to say an empty list of messages). This API call incurs costs. The number of empty messages is the main metric we want to improve with different polling methods. To make your polling more efficient and reduce costs you can choose between **short polling** and **long polling**.

### **Long Polling Reduces the Number of Empty Messages**

Long polling reduces the number of empty messages. You can activate this by setting the **Receive Message Wait Time** to a larger value than 0. This can be set during queue generation or after the generation.

Long polling is querying all servers and waiting till a message is available. It will not query a subset of the servers but all servers. This makes long polling much more efficient. SQS is only sending a response if **at least one message is available** or the wait time has expired.

### **Short Polling Gives You Access to Your Message Immediately**

Short polling queries a subset of the available SQS servers. It doesn't wait for a minimum number of messages. But it will always return a response. Even if no message is available. This will incur costs.

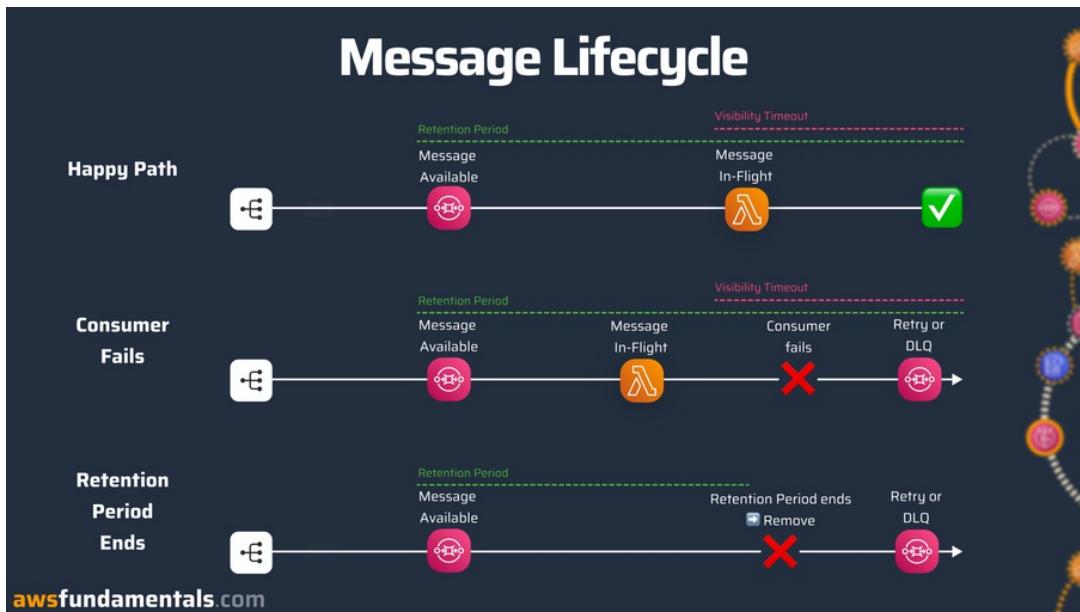
Short polling is activated by default because the receive message wait time is set to 0 .

With short polling, you will get your message much faster than with long polling. If this is a crucial factor use short polling. For all other cases use long polling to save costs.

### **Three Common Message Lifecycle Within SQS Are the Happy Path, Consumer Failures, and Too-Short Retention Periods**

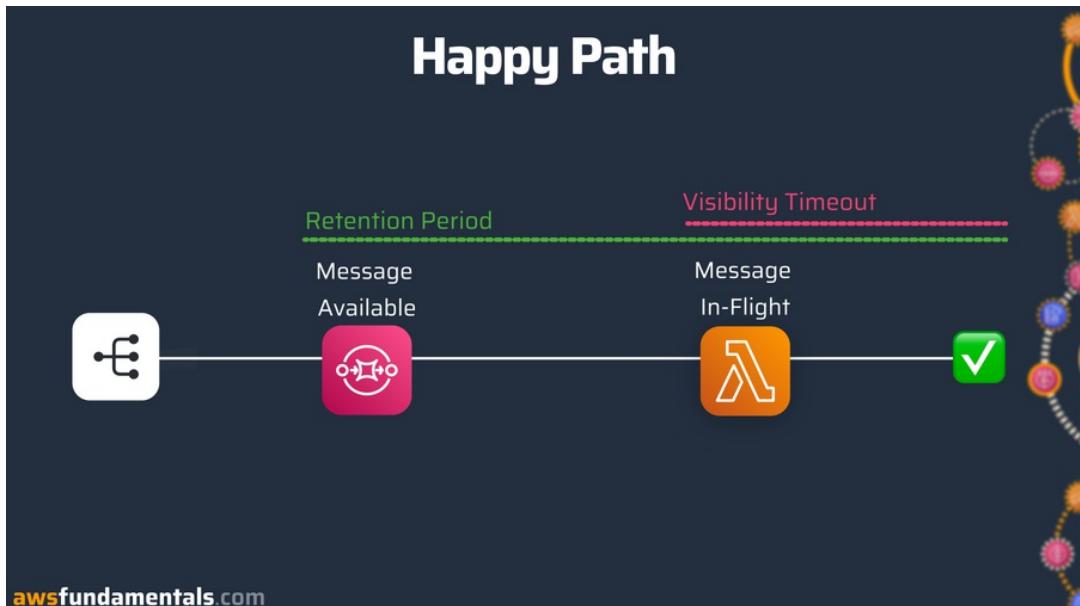
Let's now take a look at the whole picture. How is all of this coming together? There are many different scenarios with SQS messages. We will look at three really common scenarios.

1. The Happy Path: Everything works
2. Consumer fails
3. Retention period ends



### Happy Path

Let's see the happy path first.



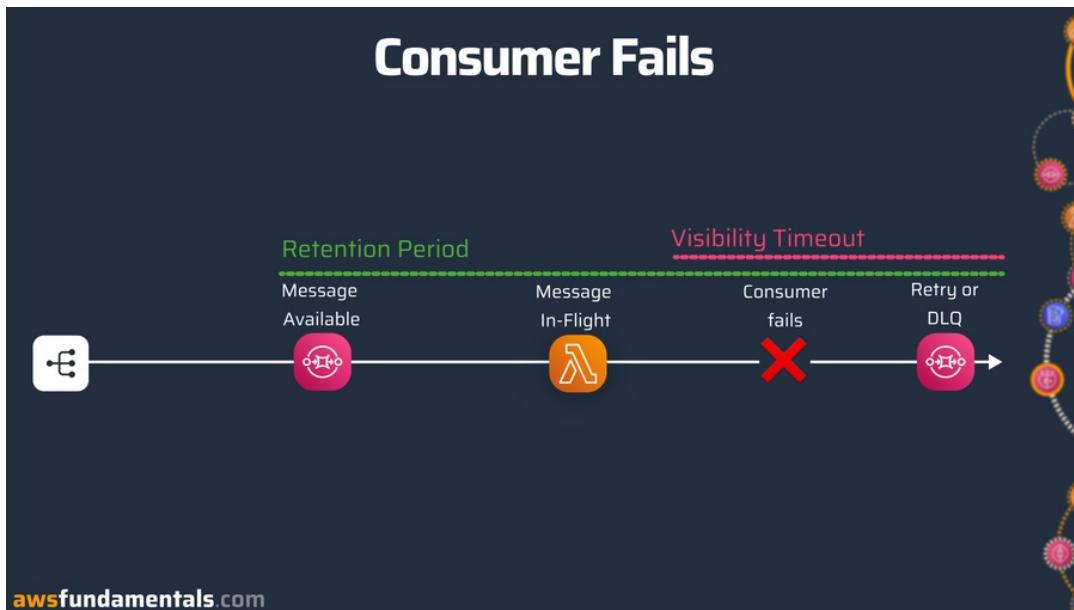
1. The Publisher sends a message to the Queue.
2. The message is now in the state **Message Available**.
3. The **retention period** time starts.

4. A lambda function polls for new messages and receives the message.
5. The message changes the state to **In-Flight**.
6. The **visibility timeout starts**.
7. The message is processed successfully.
8. The consumer removed the message from the queue.

Everything worked and the message is removed from the queue.

### Consumer Fails

The second scenario shows what happens when the consumer fails at executing the message.



1. Steps 1 - 6 are the same. The consumer picked up the message and the message is **in-flight**.
2. The consumer **fails** to work on the message → Lambda fails.
3. Lambda **doesn't** remove the message. It will reappear in the queue after the timeout expires.
4. The **Redrive Policy** will be used now. If one is defined it will be executed, if not the message will be deleted.
  - a. Redrive Policy Defined: Retry the message or move the message into DLQ.

- b. Redrive Policy not defined: Remove message.

#### Retention Period Ends



The third scenario shows what happens when the retention period ends. This is often due to misconfiguration of the queue.

1. The publisher sends a message to the Queue.
2. The message is now in the state `Message Available - Retention Period Starts.`
3. No consumer picks up the message.
4. Retention period ends.
5. Depending on the Redrive Policy the message will be removed or sent to the DLQ.

These are the three main lifecycles that happen in SQS. Of course, there are many more. But these three are very common to understand.

## **Encrypt Your Message on the Server Side**

You can activate server-side encryption in your queue to ensure that nobody else accesses your messages. You can either use SQS-owned keys or provide your own with KMS (Key Management Service). By using KMS you can also let your user provide their own keys.

Both encryption modes protect your messages using 256-bit AWS encryption. SQS encrypts your message once it receives it. If you use custom KMS keys make sure that all of your consumers are able to use these keys as well. If you haven't defined this your messages can't be decrypted. Remember: KMS adds additional costs.

## **SQS Quotas Like Batch Size, Message Size, and Messages In-Flight Are Important to Know**

Each service, account, and region have different quotas. There is a huge difference in quotas for the type of queue you use. Standard vs. FIFO queue.

Before you start designing applications in AWS make sure to understand quotas. Quotas often shape the architecture of your cloud application.

Limit	Standard	FIFO
<b>Message Size</b>	256 KB	256 KB
<b>Messages per queue</b>	$\infty$	$\infty$
<b>Messages In-Flight</b>	120 k	20 k
<b>Operations per second</b>	$\infty$	300 / s Batching: 3,000 / s
<b>Batch Size</b>	10,000	10

### **Message Size**

The default maximum message size is 256 KB. This is the same for both and is important to note.

If you have larger items consider saving data in a database or in S3.

### **Messages per Queue**

Messages per queue are infinite. This refers to **available messages**, not messages in flight.

### **Message In-Flight**

Messages In-Flight are messages already picked up by a consumer. Consumers are currently working on these messages. For standard queues, this limit is 120,000 messages and for FIFO queues the limit is 20,000. FIFO queues have an additional processing overhead to keep the order and keep the exactly-once processing.

### **Operations per Second**

There are no limits to the standard queue. For the FIFO queue, it is limited to about 300 API requests per second. You can batch your messages into one batch of 10 messages. If you do that you can work on up to 3,000 messages per second.

### **Batch Size**

The batch size for standard and FIFO queues is 10.

### **SQS Bills You on the Number of Requests**

SQS is a serverless service. That means you pay only for your usage.

One thing to be aware of is that your consumers are polling constantly. That means if you create many SQS queues with Lambda functions attached. The event source mappings will incur costs. Even if you don't send any messages to the queue.

You can reduce this number by activating long polling.

The pricing differs for standard queues and FIFO queues. FIFO queues are more expensive in general. The pricing is tiered, which means the higher your usage is the cheaper the price per request gets. The pricing also slightly varies per region.

Tier - Requests per month	Standard	FIFO
<b>1 Million</b>	Free	Free
<b>1 Million - 100 Billion</b>	\$0.40	\$0.50
<b>100 Billion - 200 Billion</b>	\$0.30	\$0.40
<b>&gt; 200 Billion</b>	\$0.24	\$0.35

It is free to get started with SQS. Always consider your consumers as well if you calculate the

pricing.

Especially if you don't use serverless consumers like Lambda functions. If you have a whole EC2 instance only for SQS it will cost more.

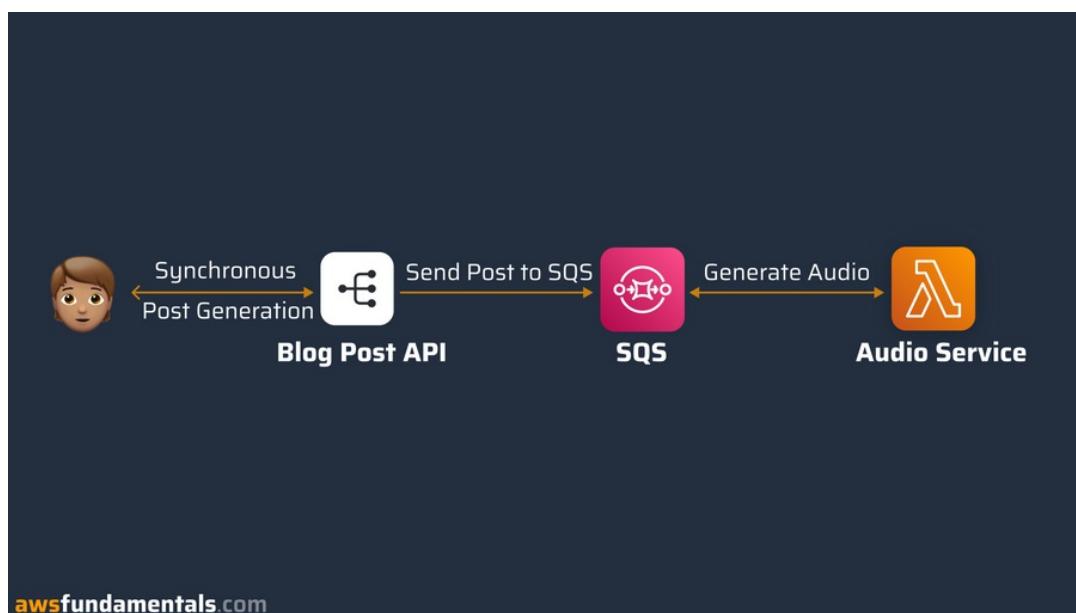
And make sure to think about if that makes sense.

### Use Cases - Build Asynchronous Tasks With SQS

Let's see three different use cases for SQS. Use cases in SQS are always around building event-driven systems & decoupling your systems. It's like a swiss army knife for building event-driven systems.

#### Use Case 1: Running Tasks Asynchronously (In the Background)

Imagine you have a blogging system. Each time your customer posts a blog post you want to transform this blog post into an audio version. We had this example already in the data chapter.



Generating audio can take a while, especially for good-quality and larger posts. If a user hits the publish button of the blog post you want to send a response as fast as possible. You shouldn't do the audio generation in this function. You can send the post to SQS and queue the task. By doing that you now decoupled your post generation from your audio generation.

If we would do everything in the Create Post API the end user needs to wait a long time until

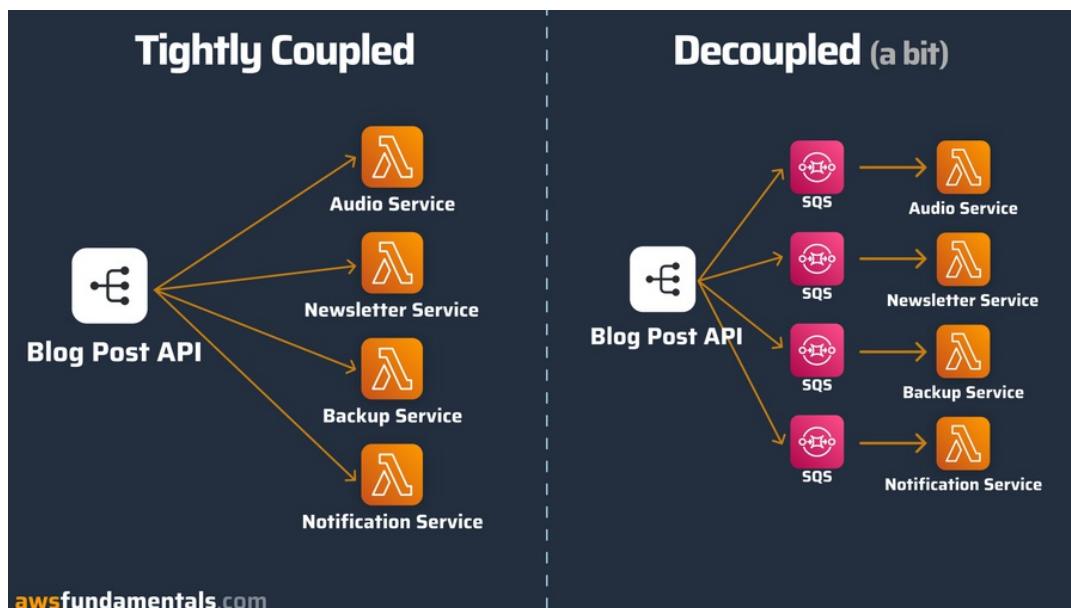
the blog post is generated. We don't want that. Instead, the performance should be fast.

### Use Case 2: Building Event-Driven Architectures

SQS enables you to build Event-Driven Architectures (EDA). In an EDA you have one producer of an event, often a user-facing API. There is also a consumer of an event, for example, a Lambda function. The message queue is in between and stores the event.

Building EDA has several benefits. It allows for more robust error handling and comes with basically free retry mechanisms. It is also much easier to store messages and replay them in case of bugs. You decouple your architecture. If you would do everything in one function (like the create post function) all functions would be tightly coupled.

Tightly coupled means that you need to touch many places to add a new service. It also means that the function always needs to wait for the slowest function.



Another common scenario is the failure of consumers. It can get quite complex if you need to handle failures if you run all your functions in one API.

Let's take an example of our Blog Post API.

In this example we have four different services that run after the API creates a post:

- Audio Service
- Newsletter Service

- Backup Service
- Notification Service

If one function encounters an error the API needs to handle that. By having message queues in between you can leave the error handling up to another function. The Blog Post API is then only responsible for creating the post. Nothing else.

In the image above it says **Decoupled (a bit)**. There are still some couplings in this architecture but for now, it serves as a good example. There are more ways to build event-driven systems on AWS but SQS is often the first starting point to build it.

#### Use Case 3: Buffer spikes in traffic

SQS can also be used for buffering spikes in traffic. If your application experiences a high load, SQS can be used to save messages and let your consumers decide when to work on the messages.

Let's see an example with the blogging platform again. If your application experiences a high load and multiple hundreds of people are publishing a blog post at once. Your consumer could be overloaded with requests. SQS can help you with that by saving the messages in a queue. Your consumer picks up the messages one-by-one if the consumer has the capacity to do that.

Error handling is also much easier in this setup since you can save all failed messages in another queue.

#### Tips for the Real World

Here is a quick list of recommendations. This is not a finished list but some things to get you started

- Cloud Engineering is an **iterative** approach. For lower workloads get started and see how your application and bill react
- Think about which **type of queue** (standard vs. FIFO) you should take **before** creating a queue. Keep in mind all limitations
- You should use **batching**. If you use batching make sure to understand how to handle failures. If an error in a batch happens the whole batch will be repeated. 2021 partial batch failure was launched
- Make sure to understand **error handling & retries**. Don't rely on it. Test it by actively

letting your clients fail. Trace your requests across DLQs and understand them properly.

- Use **long polling** to reduce costs
- **Always** use a DLQ
- Calculate your **Visibility Timeout**. Especially for high-throughput applications, you can follow AWS's recommendation.

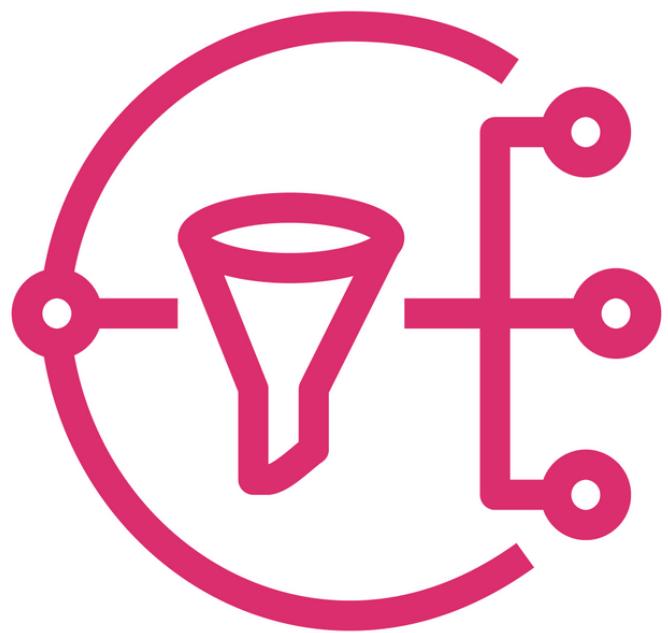
## Final Words

Mastering SQS will teach you a lot about asynchronous message handling in the cloud. SQS is also often the first service you use if you want to build up event-driven systems.

SQS gives you the power to handle millions of requests a second. The first million will be free of charge with the AWS Free Tier, so you can get started easily.

Any proper application you will see in the real world will have asynchronous tasks. And you will see SQS many times out there. So make sure to use the service and get your hands dirty.

In the next chapters, we will see more services to build event-driven systems.



Amazon **SNS**

# SNS to Build Highly-Scalable Pub/Sub Systems

## Introduction

Amazon Simple Notification Service (SNS) is a fully managed publish and subscribe service. A publisher sends messages to topics. Subscribers subscribe to topics. The topic is the distributor of messages in SNS. It forwards all messages to the subscribers.

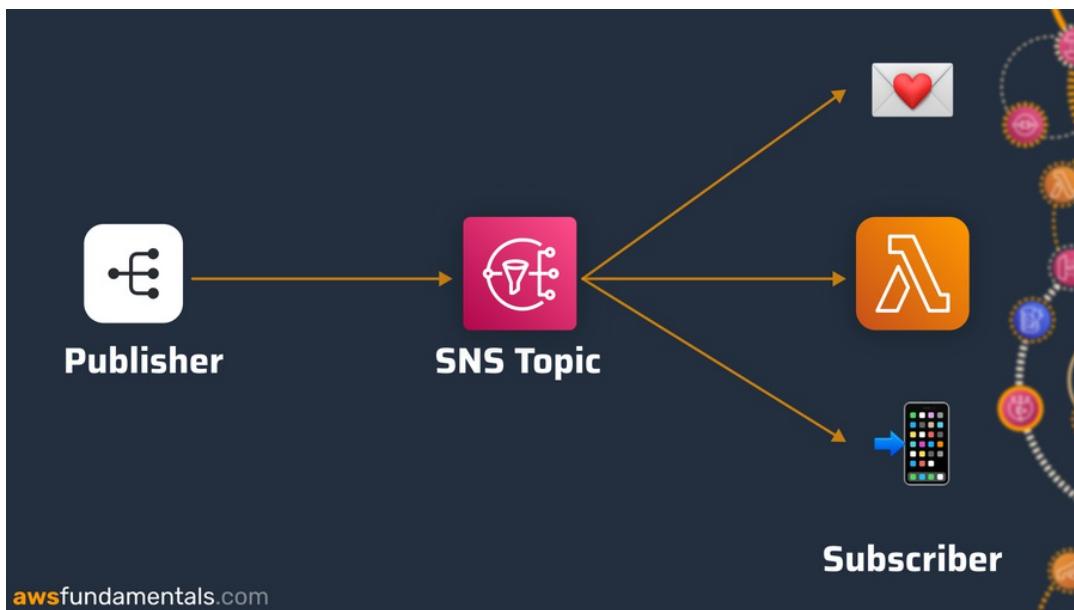
Subscribers can either be personal applications like smartphone notifications, emails, or SMS. A subscriber can also be another AWS Service.

SNS is ideal for high-throughput applications. There can be 12.5 million subscribers per topic.

## **SNS Is a Pub/sub-Service - Consumers Subscribe to Topics and Producers Publish Messages via Topics**

SNS is publishing messages to subscribers with a push-based model. Once a message comes in, SNS pushes out the message to its subscribers immediately.

SQS on the other hand is a poll-based mode. That means a message remains in a queue and consumers poll this message.



Consumers subscribe to topics. This is similar to you subscribing to a newsletter. Topics have a list of all subscribers. Publishers don't need to know about their consumers at all. They only need to know which topic to send their message to and that's it.

This decouples the systems a lot. In an SNS context, you have

1. **Publisher** - for example, your REST API.
2. **Topic** - a topic to where to send your messages and where subscribers can subscribe.
3. **Subscribers** - consumers that choose to receive the message.

### Destinations in SNS Are Either Application-Based or Person-Based

So far we understood that SNS can send messages to another AWS Service or to a personal device. Let's see in more detail which services are exactly supported.



SNS distinguishes the endpoints into **Application to Application** and **Application to Person**.

The picture above shows you all the available services. Let's go through them for each category

#### Application-To-Application

Application to Application sends messages to other applications and not to people. This is often used to further process messages. Either before sending it to a final customer or also to launch background tasks. It is pretty common to use a **fanout pattern** which we will look into in more detail later on.

The supported application services are:

1. **SQS** - feed your data into a queue
2. **Kinesis Firehose**- data stream to capture data
3. **Lambda** - invoke a function with a message
4. **HTTPS API** - call webhook APIs from SNS
5. **EventFork Pipelines** - this allows you to customize event handling

#### **Application-To-Person**

Application to Person endpoints sends messages to customers.

The supported services are:

1. **Chatbot** - Chatbot is an AWS service that allows you to create chat flows.
2. **SMS** - you can send an SMS to your customer
3. **In-App Notification** - you can also send notifications to your customer's mobile phone
4. **Email** - it is also possible to send emails to your customers
5. **Pager Duty** - there is an official integration with Pager Duty. You can get informed about CloudWatch alarms for example

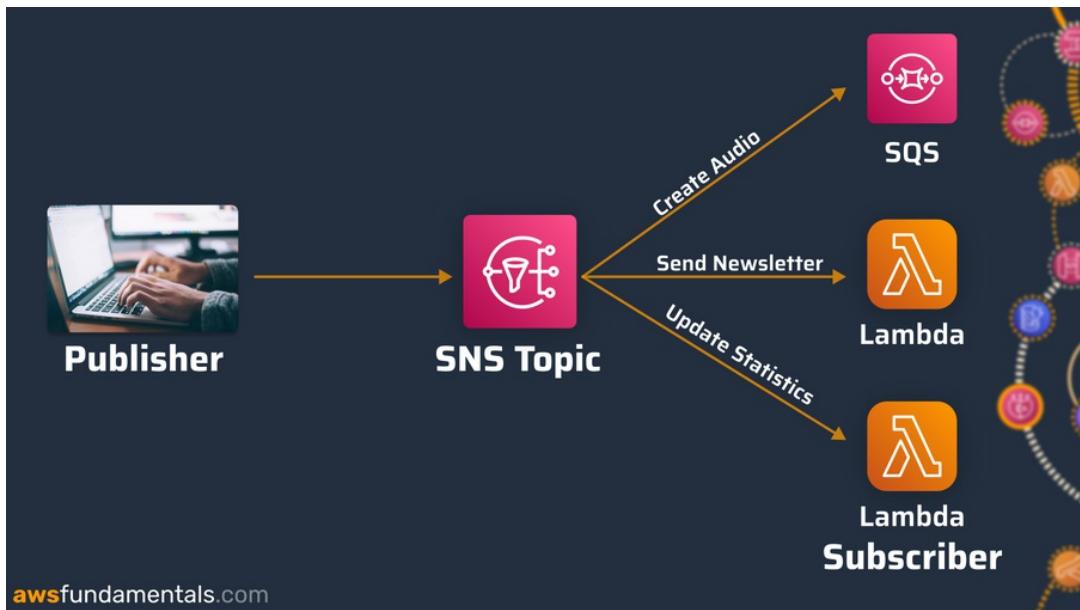
#### **The Fanout Pattern Allows Distributing One Message to a Variety of Services**

The fanout pattern is a very popular pattern in SNS. It is especially useful for building event-driven systems.

The fanout pattern is about event handling. SNS takes an incoming event and **fans it out** to many consumers. This is also a major difference in how SQS is handling events. SQS only has one consumer. Whereas SNS has many consumers.

This is the default behavior of SNS. You don't need to configure anything. The main intention of the fanout pattern is how it can be used in event-driven systems. Let's take a look at an example use case.

#### **Use Case Example - A Social Media Platform**



By using the fanout pattern you can decouple your architecture.

Imagine having a social media platform. Your users upload a post. Three actions should start after the upload happens:

1. Translate the post into several languages
2. Create an audio version of that post
3. Update statistics in the database

By subscribing your applications to this topic you can achieve this behavior.

SQS Will handle the audio creation. Two different Lambda functions handle the newsletter and updating of the statistics table. The fanout pattern allows you to decouple your architecture.

Your producer only needs to send the message to **one SNS topic**. SNS takes care of sending messages to all its subscribers.

### Secure Your Access to Topics with IAM Topic Policies

You can control access to your topics with AWS IAM by creating topic policies.

For example, you can restrict your topic to be only subscribed via HTTPS with this policy:

```
{
  "Statement": [ {
```

```
    "Sid": "OnlyHttps",
    "Effect": "Allow",
    "Principal": {
        "AWS": "111122223333"
    },
    "Action": ["sns:Subscribe"],
    "Resource": "arn:aws:sns:us-east-2:444455556666:MyTopic",
    "Condition": {
        "StringEquals": {
            "sns:Protocol": "https"
        }
    }
}
}
```

These statements follow a typical IAM statement. You have an `action` on a `resource` with `effects` and `conditions`.

In this case, the `condition` enforces the usage of only the HTTPS protocol.

## Encrypt Your Messages with SNS Encryption

You can also encrypt your data in SNS. You need to encrypt the message in two places, **in transit and at rest**.

**In-Transit** refers to the transportation layer. The HTTPS protocol and an SSL certificate encrypt your message in transit.

Encryption **at rest** refers to the actual encryption on the **disk on the server**. You can activate this in SNS.

SNS handles the whole encryption process on the server side. You can either use a key of AWS's own Key Service (KMS) or provide a custom one. This will encrypt all sensitive data in your message.

Be aware that some metadata is **not encrypted**:

- Topic metadata like names & attributes
- Message metadata like subjects, message ID, timestamp, and attributes

- Metrics for topics

## **Choose Your Topic Type - Standard for High Throughput or FIFO for Message Ordering**

In SNS you can choose two different types of topics.

- **Standard:** No message ordering, high throughput
- **FIFO:** Lower throughput but message ordering

### **Standard Topics Don't Preserve the Message Order**

A standard topic doesn't save your messaging order. That means SNS can send your messages in a different order than they came in.

Remember that SNS is a push-based service and not a poll-based service. SNS is pushing out the messages the second they arrive in SNS. Message ordering is only important if you send messages at the same time. In all other cases, this won't be an issue.

### **FIFO Topics Preserve the Message Order**

FIFO follows a **first-in, first-out** approach. It preserves the ordering of your incoming messages. There are many use cases where it is critical to keep the correct order. For example, if you want to send out notifications to your users in a special order like:

1. Order came in
2. Order is processed
3. Order is sent out

If the times between each message are low we'd recommend using a FIFO topic here.

To preserve the order you need to add a message group ID. The group ID orders all messages in that group. One example ID could be the user ID. SNS will order all messages belonging to this user ID.

### **Deduplication IDs to Ensure Each Message Is Only Sent Once**

SNS follows the **exactly-once** processing like how SQS behaves. This is due to the nature of a distributed cloud architecture. In our experience that happens far less with SNS. It is still

important to keep that in mind. The deduplication ID makes sure that SNS only sends your message one time.

### **Build Message Archives with Kinesis Firehose**

One best practice when working with SNS is to use a message archive. This archive saves all your incoming messages for a certain time.

This can be very useful in the following scenarios:

- You introduced a bug and need to repeat messages
- Using production-like messages in your development workflow
- Running analytics on your messages
- Checking compliance requirements

SNS doesn't have an in-built archive (unlike to EventBridge) but you can use **Kinesis Data Firehose** for that.

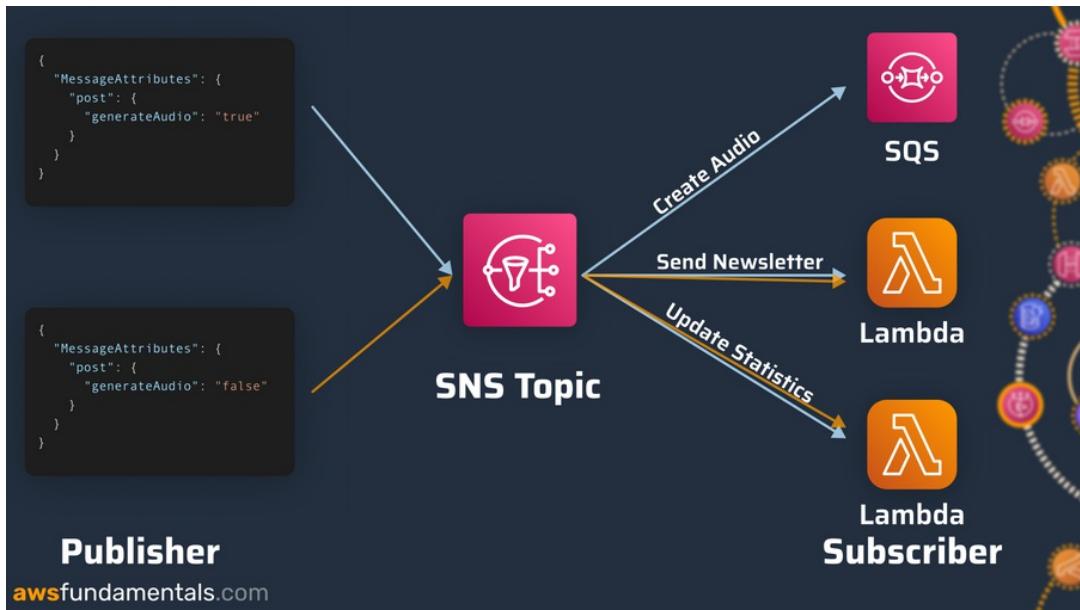
Kinesis allows you to save all incoming messages in S3 or Redshift. You can also build Lambda functions or Athena queries for the above use cases.

### **Message Filtering Sends Only a Subset of Messages to Subscribers**

Message filtering allows you to send only a subset of messages to subscribers. You can assign filter policies that check the message for certain attributes. If a message meets these policies, SNS sends it to the subscriber. This allows a flexible routing mechanism.

The filter policy can be both applied on **Message Attributes** and the **Message Body**.

Let's see an example:



In this example, we check for the field `generatedAudio` in the `post` object. This field is `true` for one message and `false` for the second message. Only the message with `generateAudio` set to `true` will be sent to the `Create Audio` queue.

### Delivery Retries Define How Retries & Error Handling Is Working for Server-Side Errors

For each delivery protocol, SNS defines a **delivery protocol**. With this protocol, you can define how retries are happening. Retries will only happen on **server-side errors**.

#### Understanding Server-Side Errors

Compared to SQS, SNS is **not aware** of errors happening **inside** of your Lambda function. SNS only cares about sending out your messages. That means once your message is out it is successful for SNS.

If your Lambda function fails to work on the message the delivery protocol **will not be aware of that**. SNS calls your Lambda function asynchronously. We highlight this because this is very often misunderstood.

A common server-side error is a missing IAM policy. If your topic isn't allowed to call a Lambda function a server-side error will occur.

Another error would be if the Lambda API is down but this doesn't happen very often (fortunately).

With the delivery protocol, you can then define retries. The current default policy looks like that:

```
{  
  "http": {  
    "defaultHealthyRetryPolicy": {  
      "numRetries": 3,  
      "numNoDelayRetries": 0,  
      "minDelayTarget": 20,  
      "maxDelayTarget": 20,  
      "numMinDelayRetries": 0,  
      "numMaxDelayRetries": 0,  
      "backoffFunction": "linear"  
    },  
    "disableSubscriptionOverrides": false  
  }  
}
```

You will define the following parameter:

- How many retries should happen?
- How many retries should happen without delay?
- What are the minimum and maximum delays?
- Which backoff function should be used in case multiple errors happen?

Retries are very powerful but keep in mind that they are only **used for server-side errors**. You still need to work on client-side errors. If your Lambda function fails, you need to take care of that as well. You will either use another SQS queue instead of a Lambda call. Or you will use Lambda Destinations and Dead Letter Queues for these cases.

### Dead Letter Queues save All Failed Messages

SNS also allows the usage of Dead Letter Queues (DLQ). DLQs receive all failed messages. A failed message is only failed if a **server-side error happened**. We introduced this in the last section.

It is always best practice to use DLQs. In a worst-case scenario (e.g. you removed an IAM policy on accident) your DLQ will save the failed message. Each failed message is available in

your DLQ. After fixing the bug you can go ahead and send the message again to the topic or subscriber.

This is pretty important for notifications. Imagine you want to send notifications to all your user's smartphones and it fails. The message will be lost if you don't use a DLQ.

One important detail here: **You cannot redrive your messages from DLQs back to SNS as you can in SQS.** Redrive functionality in DLQs is only available if there is a source queue available. Since you send your messages to your DLQ from SNS there is no source queue to redrive your messages to. You either need to replay them on SNS or send them to the consumer. This is already a bit advanced. But it is important to be aware of failure scenarios. In EventBridge you can use the Archive & Replay functionality which we will check in the next chapter.

Remember: **Use DLQs.**

### Pricing Is Based on the Number of Requests

SNS is a serverless service. You don't have any fees if you don't use it. Your pricing is completely usage-based.

1 Million SNS Requests	\$0.50
100k Notifications via HTTP	\$0.06
100k emails	\$2.00

**Free Tier:** Your free tier covers 1 million requests, 100k notifications, 100 SMS, and 1000 notifications via email **every month.**

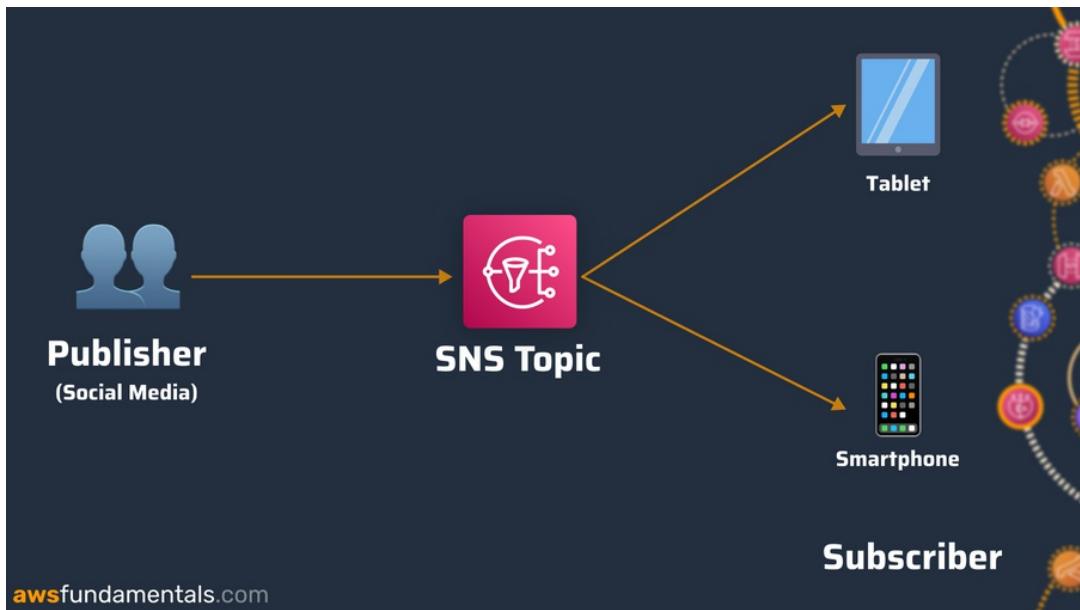
### Typical Use Cases Are CloudWatch Alarms and in-App Notifications

Let's look at some example use cases to understand this service even better.

#### Use Case 1: SNS Can Be Used to Send in-App Notifications

One standard use case is using SNS for sending In-App Notifications.

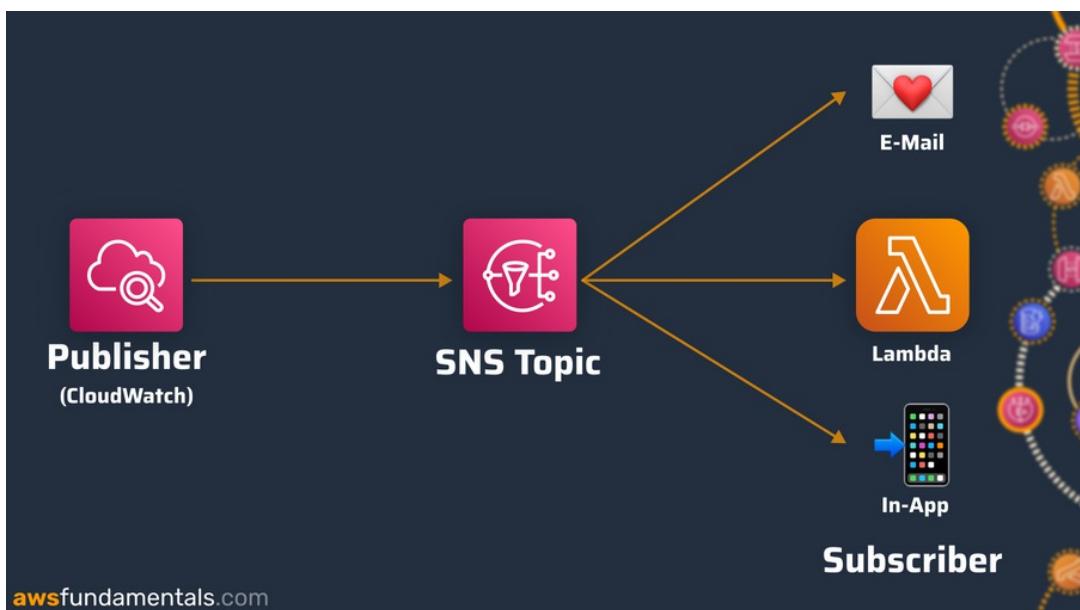
Most modern applications need to send In-App Notifications in different cases. A good example is a social media network. Once you get a comment or a like on your post the user should get informed in the app.



Different mobile devices (e.g. a tablet and a smartphone) can subscribe to this topic and SNS sends out the same notification directly to the end user's device.

#### Use Case 2: CloudWatch Alarms Trigger SNS Topics to Send Out Messages

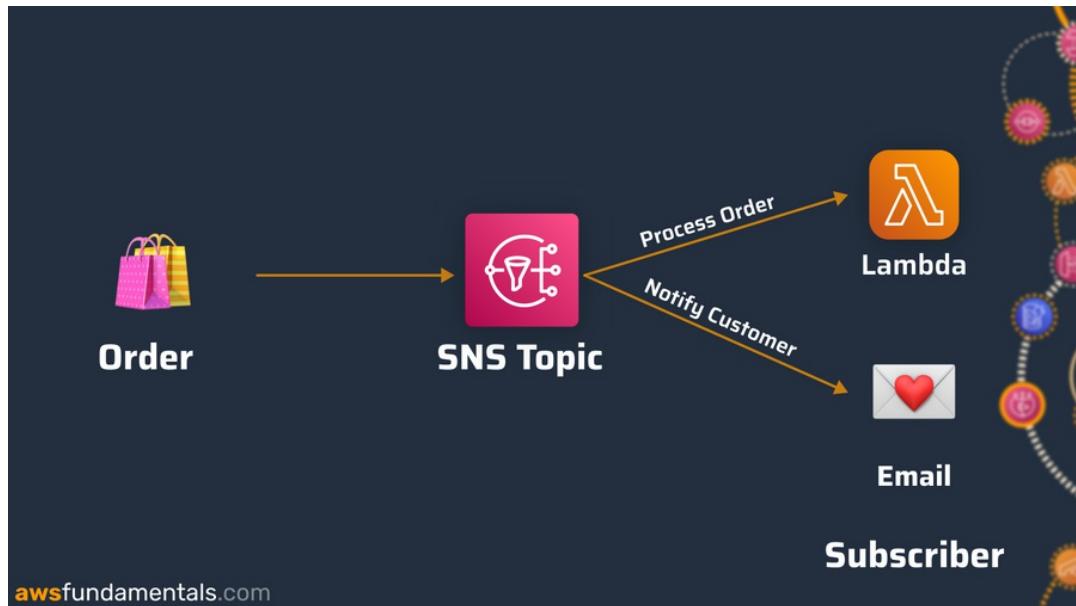
A pretty common example is the usage of CloudWatch alarms. CloudWatch is the monitoring and alerting service within AWS. Each log and metric about your services live in CloudWatch. With CloudWatch you can create alarms. One example alarm can be on failures of your Lambda function. Once it fails you should get informed.



Your CloudWatch alarm uses SNS to send an automated message to SNS once the alarm goes off. SNS takes the message and forwards it to your subscribers.

Your subscriber is often an email, SMS, or Lambda function. The Lambda function can execute code to send notifications to services like Slack, or MS Teams.

### Use Case 3: Notify Customers of an E-Commerce Application about Order Status



SNS can be used to send out real-time notifications. These notifications don't have to be only for mobile applications like phones or tablets. The notifications can also trigger different systems or combine both of them.

Imagine running an e-commerce store. Once an order comes in you can notify the end user about the order's status. At the same time, you can call a Lambda function that takes care of the next step of the order. Here you will use the fanout pattern to fan out the message to several services.

This could be used for anything from order tracking to order completion notifications.

#### Tips for the Real World

- Use SQS queues as consumers if you need to persist messages. You can't retry the failures **of consumers** in SNS.
- Utilize Kinesis Firehose to build **message archives**.

- Make sure to understand **error handling**. Test error handling by introducing errors and trying to fix & retry them.
- Use **DLQs** for server-side errors.
- Leverage SNS message **filtering** to send only a subset of messages to subscribers.
- Monitor your SNS topics and DLQs with **CloudWatch alarms**.

## Final Words

SNS is the second service we've introduced in the messaging chapter. SNS is an amazing service to build high-throughput, customer-facing applications. The ability to use application and personal endpoints are amazing in SNS.

The fanout pattern allows you to build event-driven systems. You can combine the strengths of SQS and SNS by using this pattern.

Yet, there are some things that are still hard to grasp for beginners. Especially with error handling and retries. Always keep in mind the async nature of AWS.

SNS is one of the basic services in AWS that you need to master.

The last chapter in the messaging category will be about **EventBridge**. EventBridge improves many things that we already saw in this chapter.

So let's dive in.



Amazon **EventBridge**

# Building an Event-Driven Architecture with AWS EventBridge

## Introduction

EventBridge is one of the newer services in AWS which launched back in 2019. Before the launch, the service existed with fewer features as CloudWatch Events.

EventBridge makes it easier to build Event-Driven architectures by integrating AWS services without code.

The general idea of EventBridge is that your event bus receives an event from an event producer. You can add rules to your event bus. The rule routes the incoming event to the defined consumer, e.g. Lambda, SQS, Step Functions, and many more.

## Event-Driven Architectures Decouple Producer and Consumer of Events

EventBridge is the ultimate service for implementing your event-driven architecture (EDA). Event-Driven Architecture itself is nothing new. But by integrating serverless services it got much easier to implement a proper EDA in.

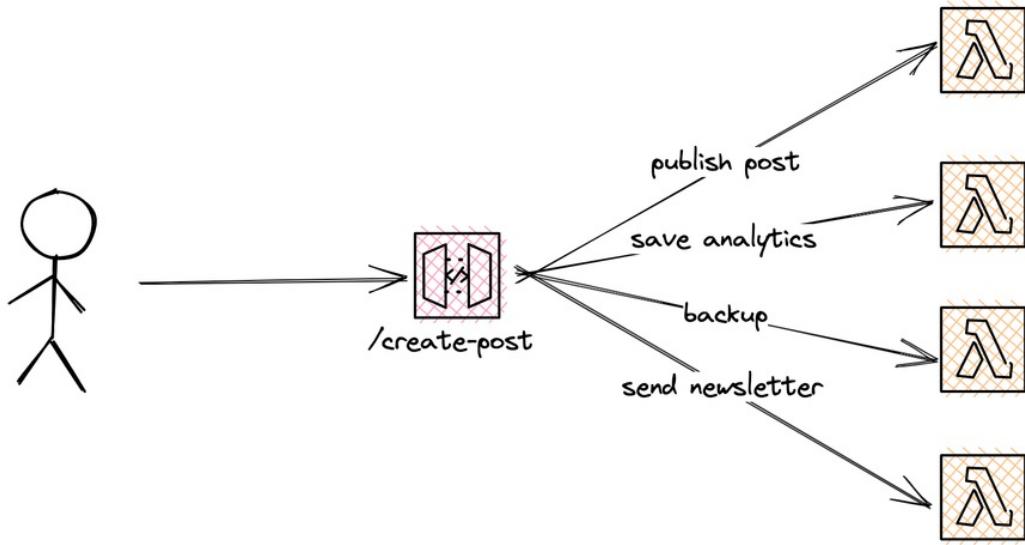
**Decoupling your producer and consumers** is one key factor in an EDA.

Before we see the benefits of an EDA let's have a look at the alternative. The alternative is having many functions or endpoints. We call this an API-driven architecture. This is how you normally start. By using this approach we quickly see the pain points.

For that example, we imagine we build a blogging platform. A user publishes a post and calls the `createPost` endpoint. This endpoint is doing many things:

1. Publish the post
2. Save Analytics
3. Make a backup of the post
4. Send a newsletter

All these different actions take place in the same endpoint, `createPost`.



There are several scenarios where this is not optimal.

First of all the coupling is high. All functions interact tightly with each other. If you want to change how the backup works you need to touch the `createPost` endpoint.

If you need to add or remove an action you also need to touch the endpoint itself. With a larger team, this can get very complex.

Once you introduce retries & error handling it gets much more complex. What happens if one of these services gets unavailable? Do we need to revert all changes? Should we retry it and let the user wait?

All these scenarios are not a good developer experience.

This is often the point where developers start looking for alternatives. Alternatives often are:

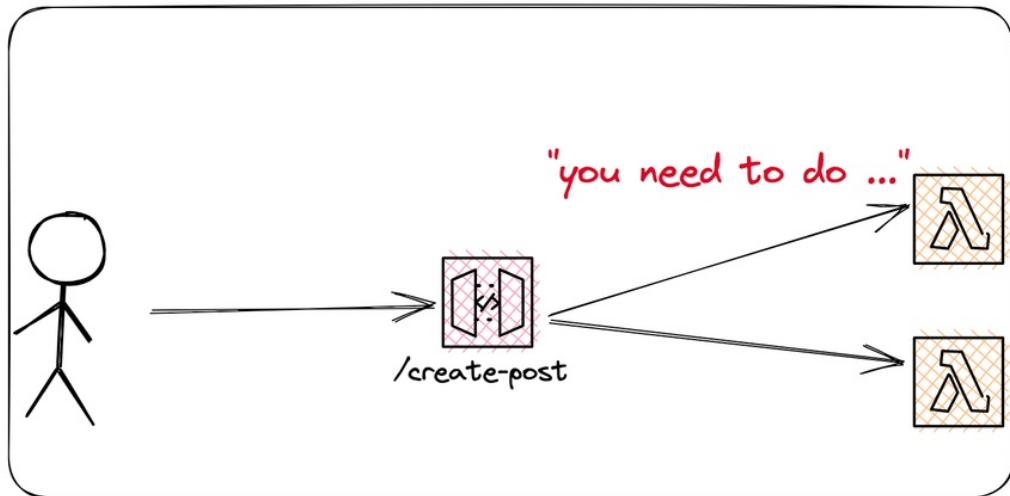
- Message queues like SQS
- Pub/Sub Services like SNS
- Event routes like EventBridge

With EventBridge, you'll get the most flexibility to decouple your architecture.

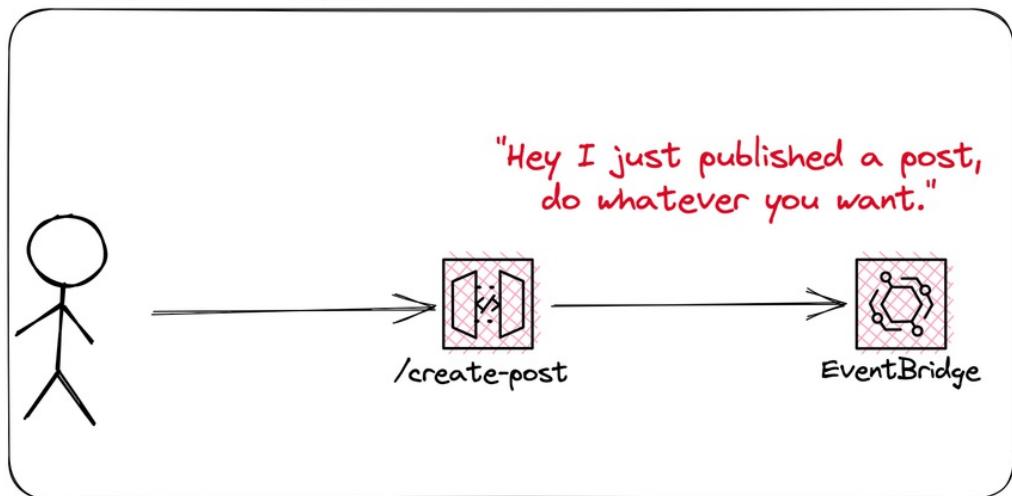
#### **EDA Decouples Your Architecture by Introducing Additional Components**

EDA helps you solve these issues. The main goal is to decouple the producer and consumer of events. EDA is solving that issue by introducing an additional component, the event router.

## API-Driven



## Event-Driven



The `createPost` endpoint would only create the event and send it to the router. This is where the responsibility of this endpoint ends.

The main difference between the API-driven, and the event-driven way is how communication happens. In the API-driven way, the API tells other consumers what to do:

- **"You need to back up the post".**
- **"You need to send the newsletters".**

In an event-driven way, the endpoint tells the router that something happened, without caring

what happens next: “***Hey I just published a post***”.

We are not directing other consumers to do things. We only tell the router what happened and it decides what happens next.

We will see the components above in much more detail soon. But to summarize, EDA has the following benefits:

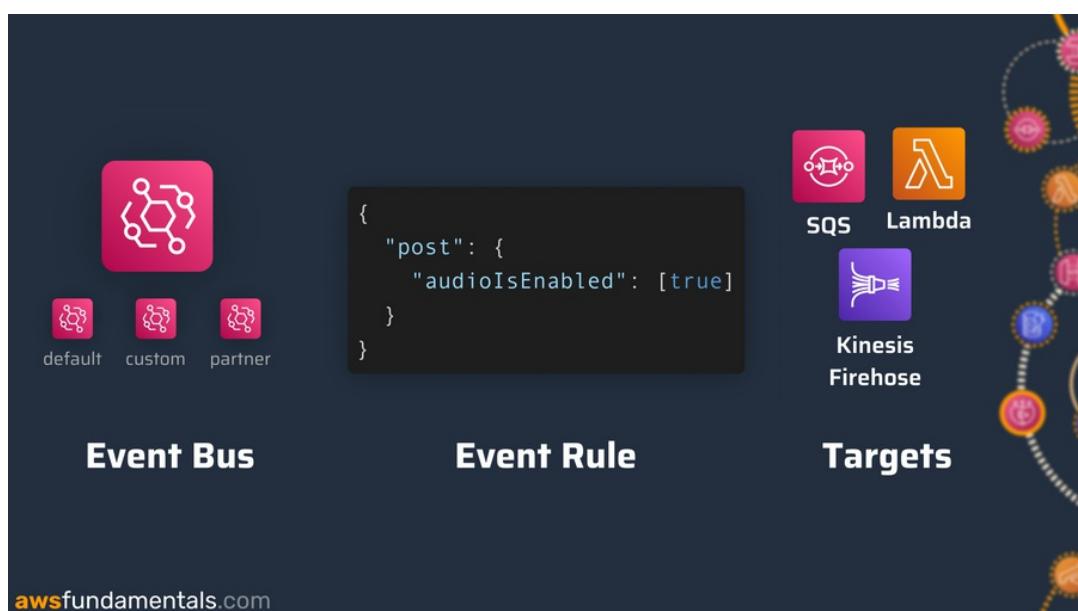
- The architecture is more **decoupled**.
- Tasks run in the **background** and the most crucial task (publishing the post) is executed fast.
- **Adding** services to events **is much easier** because they can subscribe to events.

More of these benefits will be much clearer after the whole chapter.

### Event Bus, Event Rules, and Targets Are the Main Components of EventBridge

EventBridge consists of three main components:

1. Event Bus
2. Event Rules
3. Targets



## The Event Bus Is the Receiver of All Events

First of all, there is the EventBus. The event bus is the receiver of all events. You always need to define the event bus name if you want to send events to EventBridge.

A payload submitted to EventBridge can look like that:

```
const params = {
  Entries: [
    {
      EventBusName: "MyEventBus",
      Detail: messageBody,
      DetailType: "postPublished",
      Source: "organization.api",
      Time: new Date(),
    },
  ],
};
```

The `EventBusName` needs to be present in each call to EventBridge. The event bus has a unique name across your AWS Account & Region.

There are three different main types of event buses.

### Default Bus

Each AWS account has a default event bus. AWS sends all internal AWS events to this event bus. For example, if you launch or remove an EC2 instance. Also, all scheduled events (CRON jobs) run on the default event bus. With the newly introduced (2022) EventBridge Scheduler that changes a bit. You don't need to run scheduled tasks on the default event bus anymore.

### Custom Bus

For custom events, you create a custom event bus.

### Partner Event Bus

EventBridge allows you to include Partner events. Partners are for example MongoDB, Auth0, or Zendesk. To be able to receive and act on these events you will create a partner event bus.

## Event Rules Define How to Route Events to Its Consumers

The second component is an **event rule**. This is one of the most powerful features of EventBridge.

The event rule describes the routing of events. Once an event arrives the rule sends it to the consumer if the event pattern matches.

AWS describes this as **filter patterns**. This is often quite confusing for new starters on AWS. You don't filter anything but you try to **match** on fields in the event.

Let's see an example. Assume you send the following event:

```
{  
  "post": {  
    "text": "This is my post",  
    "audioEnabled": true  
  }  
}
```

You can match everything that is present in the body. In this example, it makes sense to match the key `audioEnabled`. If this is `true` let's send it to a Lambda function. If it is `false` don't do anything.

The main benefit here is that all of that is a simple configuration. There is **no code** involved for the pure reason of forwarding the event. Less code is better.

EventBridge has different event patterns. Event patterns define the matching of the event. In the above example, our defined event pattern would look like that:

```
{  
  "post": {  
    "audioEnabled": [true]  
  }  
}
```

Do you see the array brackets (`[true]`)? That doesn't mean that the field `audioEnabled` is an actual array. This is the syntax of event patterns. You can also add more values into the brackets to match more values but that doesn't make sense in this example.

There are many more examples of event patterns in the official developer documentation.

## EventBridge Can Use a Variety of AWS Services as Targets

Targets are the **consumers of our events**. There are many AWS services supported to integrate with. Among them are:

- API Destinations
- API Gateway
- CloudWatch
- EventBus
- Lambda
- Kinesis
- SQS
- SNS

It is only possible to add 5 targets per rule, which is **not** a problem. Many people who start reading about EventBridge stop if they read about that quota.

This is not a problem due to the way we design our event rules and targets. In our case, one rule will only have one target. This is the subscription pattern. By using that the **target owns the rules**. We will dive into that a bit more later on.

EventBridge calls targets in an asynchronous way. That means EventBridge is not waiting till your consumer finishes. EventBridge only knows whether it delivered the message or not. The service has no information about the insides of your services. For example, it doesn't know if your Lambda threw an error or not. Keep that in mind.

## The Event Has the Properties of Detail, Detail-Type, and Source

The event itself is a JSON message. Look at this example event:

```
{  
    "version": "0",  
    "id": "6a7e8feb-b491-4cf7-a9f1-bf3703467718",  
    "detail-type": "postPublished",  
    "source": "organization.api",  
    "account": "111122223333",  
}
```

```
"time": "2017-12-22T18:43:48Z",
"region": "us-west-1",
"resources": [],
"detail": {
    "post": {
        "text": "This is my post",
        "audioEnabled": true
    }
}
}
```

There are **three important fields** if you think about your event message:

1. **detail** - Your event message
2. **detail-type** - A description of your event type
3. **source** - The source of your event

There are many more fields in the event like `version`, `account`, and `time`. The three stated above are the most important ones.

#### **Detail Is Your Actual Message**

In the `detail` section, you include your whole event message. We saw the example of a post a few times already.

```
{
    "detail": {
        "post": {
            "text": "This is my post",
            "audioEnabled": true
        }
    }
}
```

Your event rules will match in this `detail` section.

## Detail-Type Describes Your Event

`detail-type` is a string that defines your event. In our example, this is `postPublished`.

Your event rules are also matching on the `detail-type`. Detail types often follow a convention of noun + verb. For example:

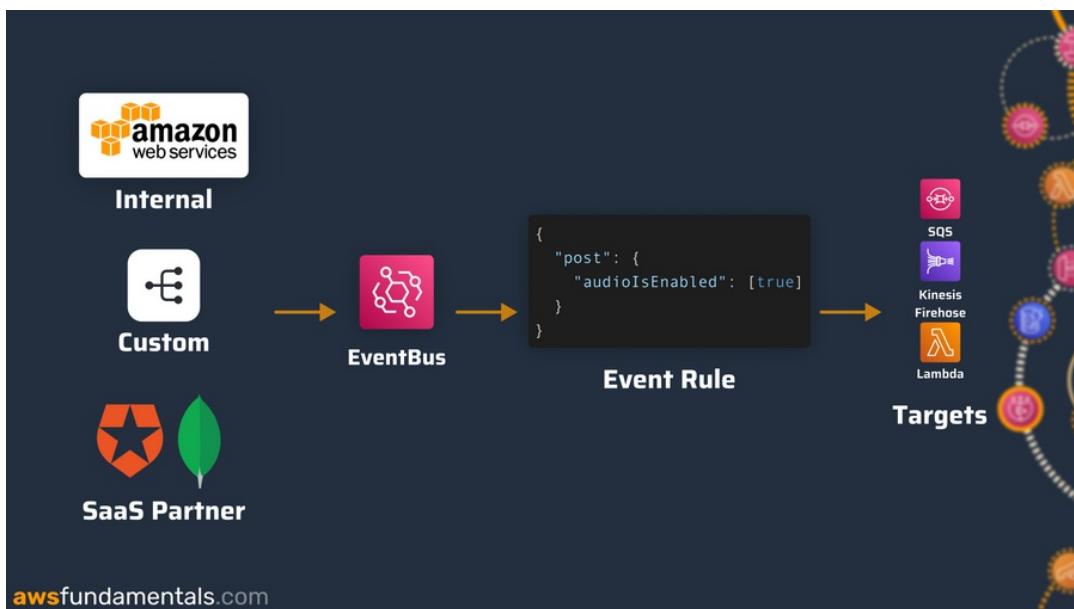
- Post published
- Order created
- Customer deleted

## Source Is Where the Event Comes From

Your `source` string describes where an event is coming from. For internal AWS events, this always starts with `aws..`

For custom events, you can define it. The best practice is to follow a dot notation (like Java package names). For example, you could do `organization.api`.

## Event Sources Are Either Internal AWS Events, Custom Events, or Partner Events



Once you integrate SaaS partners you will have their schemas available as well. This is useful to get started quickly. For example, if you integrate MongoDB you will see a schema of your MongoDB event. It shows you which attributes the event will have and what type they are.

## **Internal AWS Events Represent Actions within Your AWS Account**

EventBridge sends all internal AWS events to the **default event bus**. You can add rules for different internal actions.

- Do you need to know when an EC2 instance was created? Listen to the event

```
aws.ec2@EC2InstanceStateChangeNotification
```

- Do you want to know when an autoscaling policy was executed → Listen to events like

```
aws.autoscaling@EC2InstanceLaunchLifecycleAction
```

## **Use a Custom Event Bus for Your Application**

The second area where events are coming from is the custom event bus. Custom means that every application can send an event to an event bus. This makes EventBridge powerful, as you can create your own event bus and let your application send events to this bus.

Let's take our blogging platform again. Once we create a post we can send an event to the event bus with the following body:

```
{
  "post": {
    "content": "Hello World",
    "audioEnabled": false
  }
}
```

Our event bus checks all rules and if any matches it will forward the event.

## **SaaS Integration Receives Events from AWS Partners**

The last source of events is SaaS integrations. EventBridge has many partners with which you can connect EventBridge, as said before, amongst them are MongoDB, Autho, and Zendesk.

Looking at MongoDB: If your whole architecture is on AWS but your data lives on MongoDB you can connect these two.

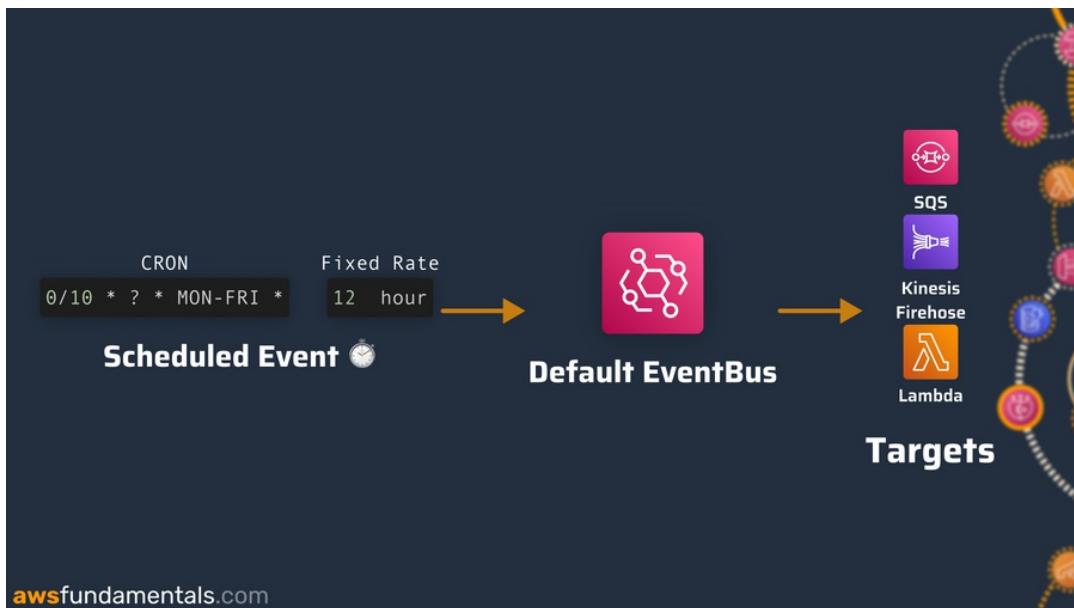
The MongoDB integration allows you for example to act on database triggers. Once you insert data in your database EventBridge receives a trigger from MongoDB. You can listen to this trigger and launch a Lambda function for example. This is like DynamoDB streams.

This makes it much easier to have a homogenous architecture.

## Schedule Tasks with the EventBridge Scheduler

One of the first functionalities in EventBridge was the usage of **scheduled events**. A scheduled event is either a one-time schedule or a recurring schedule (CRON job).

The EventBridge Scheduler was launched in late 2022. Until this point, your recurring jobs ran on the default event bus. This changed with the new scheduler.



### Create Recurring Schedules That Run Every X Minutes

A recurring schedule can be created with a CRON syntax or a human-readable rate. The human-readable syntax simply defines a rate like: **Run every 5 minutes**.

The CRON syntax looks like that: `0/10 * ? * MON-FRI *`

CRON has six different groups:

Number	Group
1	Minutes
2	Hours
3	Day of the month

Number	Group
4	Month
5	Day of the week
6	Year

You can construct powerful expressions. Let's take this expression as an example:

```
0/10 * ? * MON-FRI *
```

This describes:

1. Every ten minutes
2. Each hour
3. Any day of the month
4. Any month
5. Monday to Friday
6. Any Year

You can imagine that you can create powerful but also complex rules. It is often much easier to use the human-readable schedule function of EventBridge. With that you can create rules like:

- Run every 10 minutes
- Run every 2 days
- Run every 6 hours

For most of the use cases, this is sufficient. Think of your fellow developers when creating complicated CRON jobs.

#### **Create a One-Time Schedule for Ad-Hoc Tasks**

The latest innovation in the EventBridge Scheduler was the creation of one-time tasks. Until 2022 you needed to build this yourself with services like step functions, cron jobs, or Lambda functions.

Now, this is a built-in functionality that makes life much easier.

A common use case can be to schedule emails for a new user sign-up after a given timeframe like 7 days.

### Schedule pattern

Occurrence | [Info](#)  
You can define an one-time or recurrent schedule.

One-time schedule       Recurring schedule

**Date and time**  
The date and time to invoke the target.

2022/12/08  12:24  (UTC +01:00) Europe/Berlin

YYYY/MM/DD      Use 24-hour format timestamp (hh:mm)      Timezone

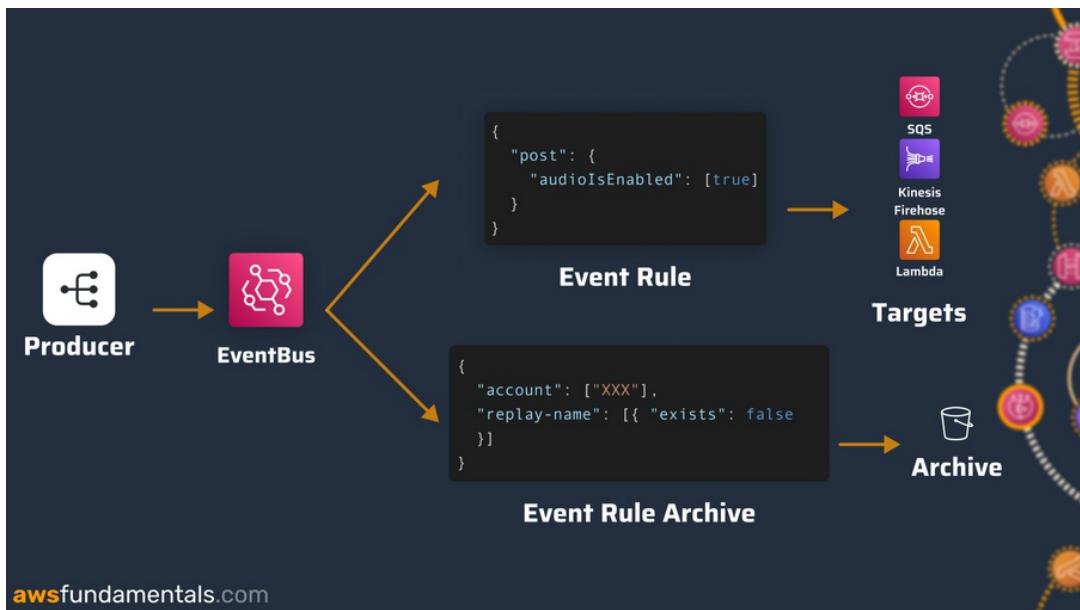
**Flexible time window**  
If you choose a flexible time window, Scheduler invokes your schedule within the time window you specify. For example, if you choose 15 minutes, your schedule runs within 15 minutes after the schedule start time.

Off

Define a date, time, and flexible time window. This task will be executed **at least once** in this time period. EventBridge has a one-minute buffer to execute these tasks. Keep that in mind in case you need a more exact execution.

### **Archive & Replay Lets You save All Events and Replay Them Back to Your Event Bus When Needed**

EventBridge allows you to create an archive. The archive saves all your incoming events. This can be helpful if you introduced a bug and need to replay failed events. It is also a good opportunity to get production data for development.



### The Archive Stores All Events You Sent to Your Bus

The archive stores all your events for a defined time. You can also store all events forever.

In the background, EventBridge uses S3 to save your data. You don't have access to this bucket. But you can replay the events on the event bus. That means you can't look at the events in their JSON format. The archive is only there for its replay capability.

When creating the archive you select the timeframe of how long you want to save the events. You can also save the events forever. The archive itself doesn't cost anything. You only pay for the storage costs on S3. This is about \$0.023 / GB in Ohio.

### Replay Functionality Sends All Events Back to the Event Bus for a Given Timeframe

The replay functionality uses all events saved in the archive and sends them back to the event bus. You can define a start and an end time for the replay. It is also possible to define the rule the archive should replay the events to.

This is pretty helpful if you've introduced a bug in a certain rule. You can go ahead and fix the bug and replay only for this rule.

Imagine an engineer introduced a bug at 10:43 and all consumer Lambdas are now failing. Even if you have DLQs enabled it would be still a hassle. Sending all events from the DLQ back to the bus requires manual work.

If you have Archive & Replay enabled you can fix the bug, and replay all events **without losing one event**. This functionality is really amazing and helpful.

The main benefit here is that this function is completely managed by EventBridge. You don't have to implement anything. You tick a box EventBridge saves all your events in the archive. You can achieve a similar behavior on SNS. But in SNS you need to build it yourself with Kinesis, S3, and Lambda.

### **Schema Bindings Define the Objects & Attributes of Your Events**

When working with events we recommend sharing a schema of your events. A schema is a description of your event properties. It describes the structure of your event.

Your customers who use EventBridge also benefit from these schemas a lot. They can see which attributes they need to send to the event bus. Or if they want to create a consumer they know which attributes are available.

Schemas are available in both OpenAPI 3 and JSONSchema Draft 4 format. Schemas are versioned. Each time the schema changes you can create a new version.

### **Internal AWS Schemas Are Already Available**

For internal AWS events, there are many schema bindings available. This makes it really easy to get started working on AWS events.

You can see all schemas if you go to the EventBridge console → Schema Registry → AWS event schema registry.

This shows you all different schemas. If you're interested in events about ECS search for **aws.ecs**.

The screenshot shows the AWS event schema registry interface. At the top, there are tabs: 'All schemas', 'AWS event schema registry' (which is highlighted in orange), 'Discovered schema registry', and 'Custom schema registry'. Below the tabs is a search bar with the placeholder 'Search AWS event schemas' and a query field containing 'aws.ecs'. To the right of the search bar are navigation icons: a magnifying glass, a '1' indicating the current page, and arrows for previous and next pages. The search results are displayed in a grid format. There are four items listed:

- aws.ecs@ECSContainerInstanceStateChange**: AWS event schema registry, Found in version 1.
- aws.ecs@ECSServiceAction**: AWS event schema registry, Found in version 1.
- aws.ecs@ECSTaskStateChange**: AWS event schema registry, Found in version 1.
- aws.ecs@AWSAPICallViaCloudTrail**: AWS event schema registry, Found in version 1.

Let's take the event **aws.ecs@ECSContainerInstanceStateChange** as an example.

It shows you the detail consisting of an **ECSContainerInstanceStateChange** event. This event occurs if a container in ECS switched its state, for example from stopped to start.

Let's look only at the `detail` object since this is the most interesting one. `detail` refers to another object called `ECSContainerInstanceStateChange`

```
"detail": {"$ref": "#/components/schemas/ECSContainerInstanceStateChange"}
```

If we dig down in the schema and find this object we see that it requires the following attributes

```
{
  "required": [
    "registeredResources", "remainingResources",
    "agentConnected", "versionInfo", "version", "clusterArn",
    "containerInstanceArn", "status", "updatedAt"
  ],
  ...
}
```

If you want to send an event of that type you need to pass all these items. If you want to build a consumer for this event you now know which attributes you will receive.

## **Custom Schema Registry Allows You to Build Your Own Schemas**

The custom schema registry allows you to create schemas by yourself. This is very helpful if you are using a custom event bus and want to share the schema.

This schema can then be shared inside your organization.

## **SaaS Integration Schemas Show the Event Pattern of Partners**

Once you integrate SaaS partners you will have their schemas available as well. This is useful to get started quickly. For example, if you integrate MongoDB you will see a schema of your MongoDB event. It shows you which attributes the event will have and of what type they are.

### **Schema Discovery Detects Your Schema and Creates It**

Building schemas requires manual work. That is why EventBridge allows you to activate **Schema Discovery** for one event bus.

Schema discovery creates your schema automatically. EventBridge discovers all fields of the incoming event and adds them to a schema. The name of the created schema will be `source@detail-type`.

If you start adding and removing attributes your Schema Discovery will be aware of that. It then creates a new schema. It also emits its own event that you can get notified once the schema changes.

```
{  
  "version": "0",  
  "id": "d06350c3-e39d-de87-56a1-87b454207202",  
  "detail-type": "postPublish",  
  "source": "organization.api",  
  "account": "157088858309",  
  "time": "2022-10-10T05:26:20Z",  
  "region": "us-east-1",  
  "resources": [],  
  "detail": {  
    "post": {  
      "content": "Hello World",  
      "audioEnabled": false  
    }  
  }  
}
```

```
    }  
}  
}
```

If we send the above event to a newly created event bus with an activated schema registry. The Schema Discovery service will take about 5 minutes and after that time I'll see the created schema.

The screenshot shows the AWS Lambda Schema Registry interface. At the top, there are tabs: 'All schemas', 'AWS event schema registry', 'Discovered schema registry' (which is highlighted in orange), and 'Custom schema registry'. Below the tabs is a search bar with the placeholder 'Search schema titles and contents.' To the right of the search bar are navigation icons for back, forward, and refresh. The main content area displays a single schema entry: 'organization.api@PostPublish'. It includes the title, 'Discovered schema registry 1 version', and the last update timestamp 'Last updated Oct 10, 2022, 07:28 AM GMT+2'.

This schema shows us that there is a `post` object in our details with the attributes:

- `content: string`
- `audio.IsEnabled: boolean`

#### **Automatically Creating Code Bindings for Your Discovered Schema**

We can even take this one step further and create code bindings. Code bindings make this schema available in our source code so that we can work with it.

EventBridge allows the creation of code bindings for the following languages:

1. TypeScript
2. Java
3. Python
4. Go

This makes it easy to import these bindings into your code. These types can be used to ensure type safety on your consumer or producer side.

### **Schema Discovery Cons - Long Time to Create a Schema, and No Optional Fields**

The schema discovery is an amazing functionality of EventBridge. It makes many things easier. But it also has some drawbacks we have encountered in the past. The main ones are:

- **Time to a schema change** - changing or creating a schema takes about 5 minutes.  
While this is okay in a production scenario, for the actual development process this is a long time.
- **Optional fields** - unfortunately, the schema discovery doesn't work with optional fields. Let's say our `audio.IsEnabled` flag is sometimes true and sometimes **not available** at all. The schema discovery will always create a new version depending on what the last event was. It won't merge the schemas.
- **One event can destroy the schema version** - if you send one event with a wrong event to your bus you can destroy your whole schema version. The discovery uses each new incoming event as the future truth for new events. This can result in many different versions.

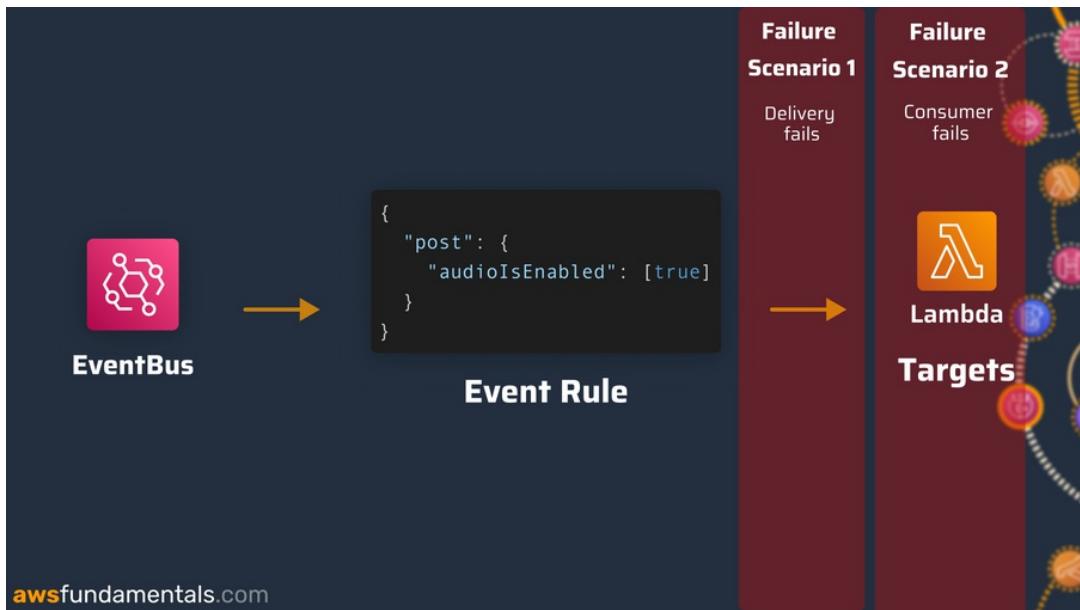
These are some difficulties we saw when working with the discovery service. It still helps a lot with starting to work with EventBridge. But it also forces you to use EventBridge in a certain way that is not always optimal for every use case.

### **Handle Failures by Using DLQs for Target Delivery**

Citing Werner Vogels, CTO of Amazon: "Everything fails, all the time".

Your service will experience some kind of failure. It is important that you know how to handle these failures.

In EventBridge there are **two ways** how errors can happen:



### 1. EventBridge cannot deliver the event to the target

The first scenario is that EventBridge can't deliver the event. For example, your target service is Lambda. The Lambda API has some kind of downtime at the moment. When you try to forward your event it will fail.

Another (more common) example is missing IAM permissions. If your event bus is not allowed to invoke a Lambda function it will fail.

We learned the concept of Dead-Letter-Queues (DLQs) already. You can also attach DLQs to your **event rule**.

You can define retries in the **retry attempts**. You also need to define how old the event can be before you move it to a DLQ.

#### ▼ Additional settings

Configure target input | [Info](#)

You can customize the text from an event before EventBridge passes the event to the target of a rule.

Matched events

Retry policy | [Info](#)

A retry policy determines the maximum number of hours and number of times to retry sending an event to a target after an error occurs.

Maximum age of event - *optional*

The maximum number of hours to keep unprocessed events for. The default value is 24 hours.

24 hour(s) 00 minute(s)

Retry attempts - *optional*

The maximum number of times to retry sending an event to a target after an error occurs. The default value is 185 times.

185 time(s)

Dead-letter queue | [Info](#)

Unprocessed events can be sent to standard SQS queue

- None
- Select an Amazon SQS queue in the current AWS account to use as the dead-letter queue
- Specify an Amazon SQS queue in other AWS account as a dead-letter queue

Select an SQS queue



For example, you can retry your event 5 times, and if the event is 1-hour old move it to the DLQ.

**Important to note here:** This is only about the **delivery** to the Lambda service. **Not** about a failure **within** the Lambda function.

## 2. The consumer fails

The second scenario is the more common one. **Your target service fails.** If we go back to our example, our Lambda function would fail.

If the Lambda function fails, **EventBridge won't get any feedback.** Once the event is delivered EventBridge is done.

Handling failures and retries for the second scenario depends on the actual service. With Lambda, you could either use Lambda Destinations or put an SQS queue in front of the Lambda. I recommend keeping it simple and using Lambda Destinations with an **onError** DLQ. You can read more about that in the Lambda chapter.

**Design your services for failures.** Mock these failures during development. You can throw an error in your consumer function to understand how errors will behave. Always try to understand and to see where errors can happen and what then happens with the events.

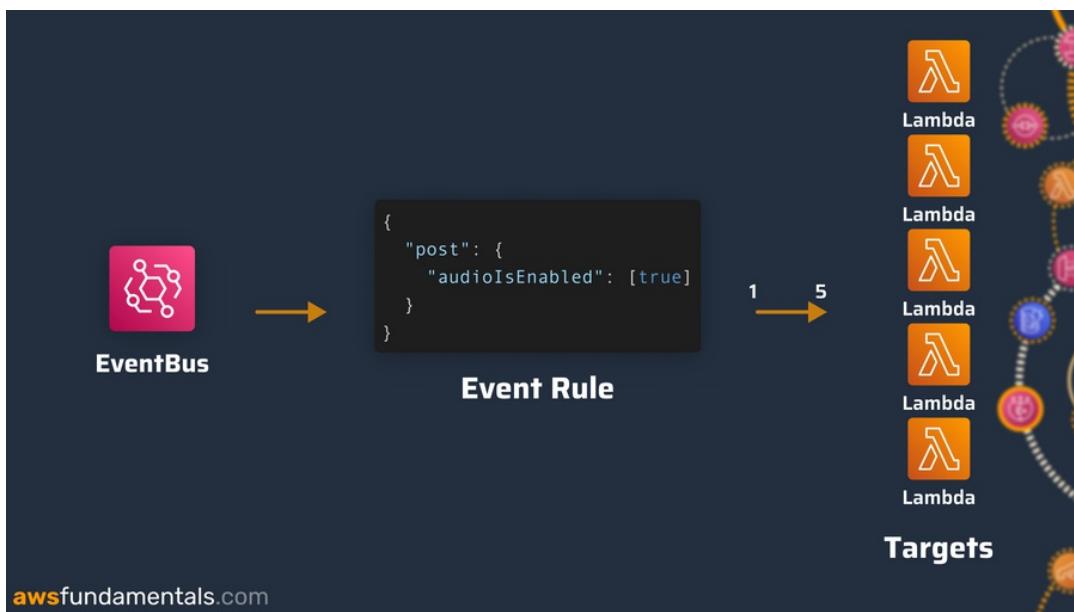
## The Subscription Pattern Defines That Rules Should Belong to One Consumer

We mentioned earlier that EventBridge has a fixed quota of **5 targets per rule**. While this sounds a bit too little we need to look into that and what it actually means.

If you create an event rule it belongs to a target. Let's take our example event rule to see if the flag `audio.IsEnabled` matched.

```
{  
  "post": {  
    "audio.IsEnabled": [true]  
  }  
}
```

The rule lives on the event bus. Now the question is how targets and rules interact. Often developers start that the target **belongs to the event rule**. And the rule decides who should get the event and who shouldn't. Like in the following picture:



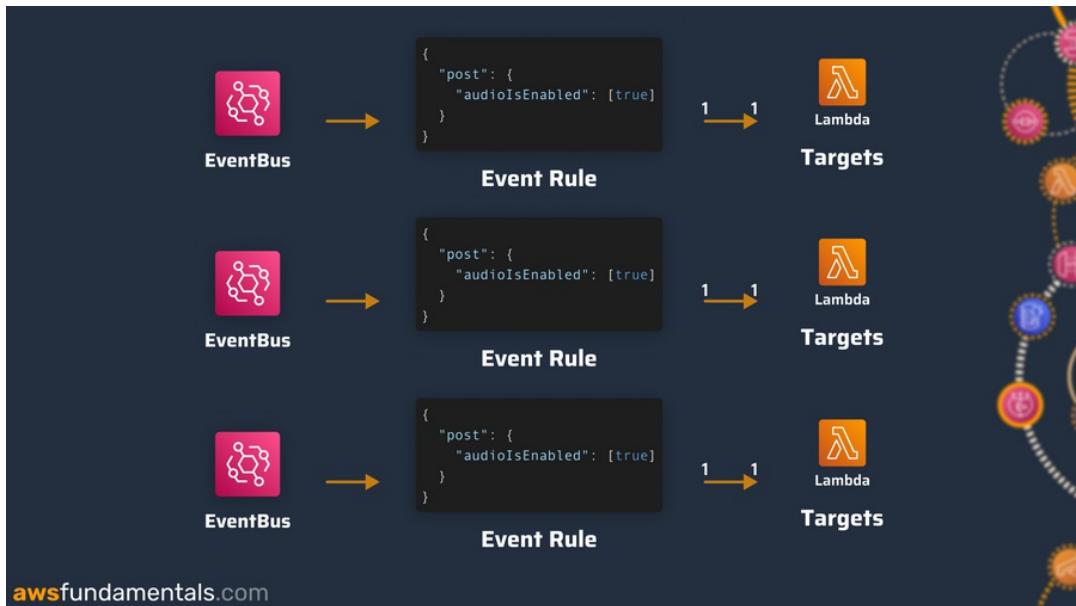
This has some drawbacks in the long run. First of all, here the limitation of 5 targets would indeed be an issue. But second, and more importantly, the rule should **belong to the target** not the target of the rule.

We use EventBridge to **decouple** our architecture. We don't want to increase coupling by coupling targets to the rules. The **consumers** should be able to flexible subscribe and unsubscribe. Without the need of changing a component that interacts with other consumers.

**The event rule should belong to the consumer.** The consumers themselves decide if they want to change the pattern or remove it. **Not the rule.** There shouldn't be any side effects when a consumer wants to change the rule.

**This is the Subscription Pattern.** Targets subscribe to events by creating their event rule. If the event pattern inside of the event rule is the same, that is okay. But the event rule **belongs to the consumer, not the other way around.**

Let's see an example with multiple rules:



Now we have a 1-to-1 mapping between the event rule and the target.

We have three event rules with the exact same pattern but different targets. Even if the pattern is the same we will create separate rules for these. Once one target wants to change this pattern it won't change all three of them. For example, if target 3 wants to change the pattern to `audioEnabled: [false]`. It doesn't need to change for all other targets as well. It only needs to change it for the target rule itself.

You can abstract this very nicely in frameworks like CDK or Terraform.

This is not the only solution here! There are several more patterns for cross-account events or multi-bus events. But this is the most important one to mention if you get started with EventBridge.

The Subscription pattern is already a bit too advanced for an introductory book. But we still wanted to cover it to give you an idea of how to design your rules.

## **EventBridge Encrypts Your Data at Rest, IAM Secures All Services**

EventBridge encrypts your data at rest by default. HTTPS secures encryption in transit. And IAM secures access control between services.

EventBridge uses encryption at rest. That means the service encrypts all your data on the actual server. It uses a default 256-bit AES Encryption. This is the go-to standard and is also used in other services such as DynamoDB.

## **EventBridge Encrypts Your Data on the Server and in Transit**

Your data is also encrypted in transit by using the SSL protocol and HTTPS.

## **IAM Tag Policies Safeguards Access to Resources**

To safeguard your data within one or more AWS accounts you will use IAM Tag Policies. The policies follow the standard pattern of a policy. Tag policies allow you to create conditions based on tags. For example, you can tag all of your production resources with the tag `prod`. In the policy you forbid sending any rules to a production resource:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Action": [
                "events:PutRule",
                "events:DescribeRule",
                "events:DeleteRule",
                "events>CreateEventBus",
                "events:DescribeEventBus",
                "events:DeleteEventBus"
            ],
            "Resource": "*",
            "Condition": {
                "StringEquals": {"aws:ResourceTag/environment": "production"}
            }
        }
    ]
}
```

```
    ]  
}
```

### Event Rules Need Appropriate Resource-Based Policies for the Services They Invoke

An event rule invokes some downstream services such as Lambda, SQS, or SNS. Your rule needs the appropriate IAM policy for that.

If you use CDK Level 2 Constructs the IAM policies are automatically granted. You can connect an event rule with a Lambda Function with an L2 construct. CDK creates the policy for exactly this resource and action automatically.

```
rule.addTarget(new aws_events_targets.LambdaFunction(lambdaTarget));
```

This call will automatically create a resource-based policy for the Lambda function that looks like that:

```
{  
  "Version": "2012-10-17",  
  "Id": "default",  
  "Statement": [  
    {  
      "Sid": "Rule",  
      "Effect": "Allow",  
      "Principal": {  
        "Service": "events.amazonaws.com"  
      },  
      "Action": "lambda:InvokeFunction",  
      "Resource": "LAMBDA_ARN",  
      "Condition": {  
        "ArnLike": {  
          "AWS:SourceArn": "RULE_ARN"  
        }  
      }  
    }  
  ]  
}
```

If you are doing it manually you need to attach the policy to Lambda.

## **There Are Quotas around Event Buses, Publishing Events per Second, and Many More**

Let's have a look at some of the most common quotas in EventBridge. Quotas can be different per region. There are many quotas around API Destinations that we didn't cover in much detail. We lay out the most important quotas here:

Description	Quota
<b>Event Buses</b>	100 per account
<b>Publishing Events - Depending on the region</b>	Between 10,000 (us-east-1) and 400 (eu-south-1) requests per second
<b>API Destinations</b>	3,000
<b>Rules</b>	300 per event bus
<b>Targets</b>	5 per rule
<b>Event Pattern</b>	2048 characters max
<b>Invocations - Depending on the region</b>	Between 18,750 (US) and 750 (Asia, Africa) requests per second
<b>Schema Discovery</b>	Nested up to 255 levels

Many of these limits can be increased.

## **Pricing Is Usage-Based. Internal AWS Events Are Free, You Only Pay for Custom Events.**

EventBridge is a serverless service. The default service events are free. You only pay for custom and partner events. Keep in mind that you also need to pay for the services you invoke.

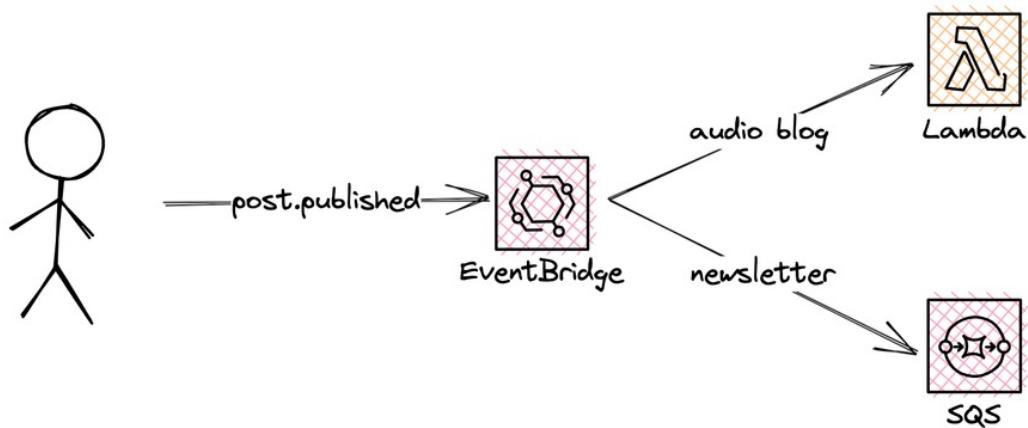
Description	Price
<b>AWS Service Events</b>	Free
<b>SaaS Partner Events</b>	\$1.00 / million events
<b>Cross Account Events</b>	\$1.00 / million events

### Three Example Use Cases for EventBridge

EventBridge is an amazing tool for building event-driven architectures. Use cases include every task where you try to decouple your architecture. Let's see three example use cases.

#### Use Case 1: Event-Driven Architecture with Blogging Platform

We saw the example of a blogging platform throughout this chapter. This is one of the standard use cases to use EventBridge for. EventBridge helps you decouple your architecture by letting you work on tasks asynchronously.

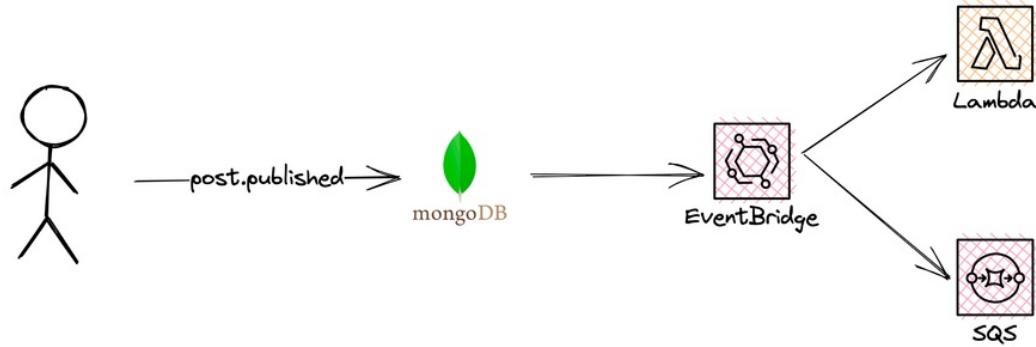


An example of architecture could look like that. A user publishes a post and posts the event `post.published` to EventBridge. EventBridge has rules to match this event. The targets for this rule is one Lambda function to process audio blogs and an SQS queue to handle newsletter.

#### Use Case 2: Integrating Third-Party Vendors - Integrate MongoDB

A second use case is to integrate third-party vendors into AWS. Having all workloads in AWS makes the observability and development of applications often much easier.

But there are often cases where third-party vendors like MongoDB, Autho, or Zendesk are used. EventBridge allows you to integrate them easily.



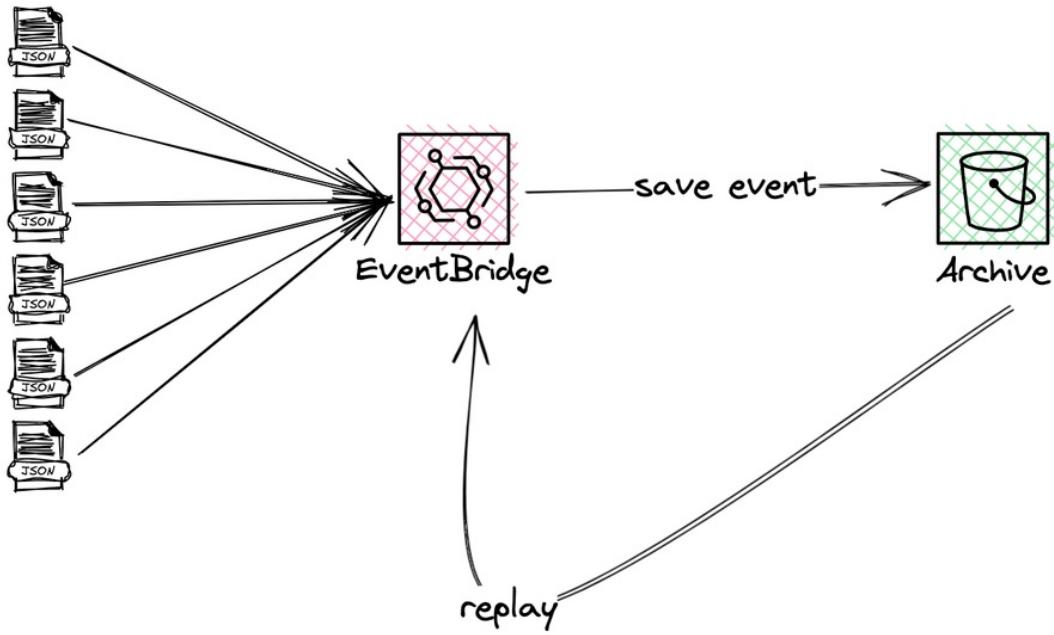
MongoDB for example has native integration with EventBridge. EventBridge can listen to triggers like data inserted, updated, or removed. This event will be present in EventBridge and can be handled like every other event.

In this example, we can listen to published posts on MongoDB. Once a post was published we can do certain actions like creating an audio version or sending out newsletters.

#### **Use Case 3: Recovery after Incidents with Archive & Replay**

Use Case 3 is about the recovery after incidents happen. EventBridge has the functionality of creating an archive and replaying all events from that archive. The archive saves all incoming events.

When you introduce an incident it is possible that all incoming events failed. With Archive & Replay, you have the ability to send all events of a given timeframe back to the event bus.



## Tips for the Real World

Here are a few tips for using EventBridge in the real world

- Understand **error handling**. Make sure to test error handling and understand how to debug issues
- Handle **idempotency**. Be aware of error cases and replay functionalities.
- EventBridge has a high latency. Make sure to understand that for time-critical events
- Create a **validation** for event schemas and incoming events. Using ZOD and middy in Lambdas and in TypeScript can help a lot
- Share **schemas** of your events. It makes the development of subscribers much easier.
- Use the **subscription pattern** for creating event rules.

## Final Words

EventBridge is one of the services we are most bullish on. Companies can build an EDA so easily by connecting several AWS services with each other.

The integration with SaaS partners makes it easy to have a homogenous architecture. Even if you use services outside of the AWS ecosystem.

Managed functions like archive & replay allow you to build reliability into your architecture. Without the need of building and maintaining it yourself. With that, you can ship fast, break stuff, and don't have too many issues attached.

AWS is releasing many new features for EventBridge every year. The community and developer advocates are amazing.

This was the final service of the messaging chapter. The three services give you an overview of how you can build an EDA. Hopefully, we also gave you ideas on when to use which service.

There are no right or wrong paths. You need to decide what suits your application best. Cloud engineering and software engineering in general are iterative approaches. Start simple and iterate about your architecture and code. Your software is never finished and this is a beautiful thing.



# Networking

It's important to understand the fundamental principles of modern networking infrastructure. They provide the foundation for communication and data transfer between different components and services of our applications. A well-designed network infrastructure is essential for ensuring that an application is highly available, secure, and scalable.

The services that play a major role in this area are AWS API Gateway, Amazon Route 53, Amazon Virtual Private Cloud, and CloudFront.

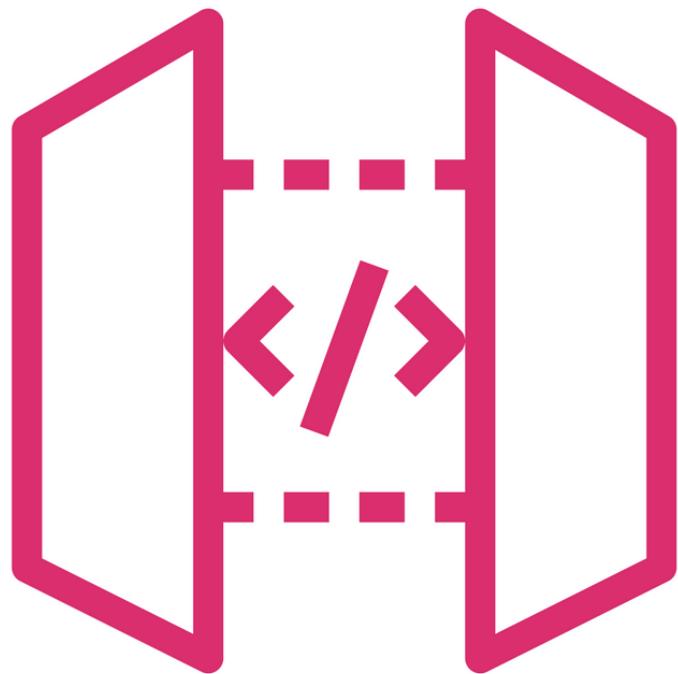
**API Gateway** is a fully managed service for creating, deploying, and managing APIs. It allows you to create RESTful and WebSocket APIs. It also comes with advanced features such as authentication, caching, and monitoring which don't require you to write much or any code.

**Route 53** is a highly available and scalable Domain Name System web service. It allows you to register domain names and route internet traffic to your application. Its advanced features enable you to build applications that are available around the globe with low latency and can automatically deal with outages via failovers.

**Amazon VPC** provides a logically isolated section of the AWS Cloud where you can launch resources in a virtual network. It provides you control over your virtual networking environment, including the selection of your own IP address range, the creation of subnets, and the configuration of route tables and network gateways. You're able to create distinct network subnets for private components to isolate them from the internet or other resources and strictly define how they can communicate with each other.

**CloudFront** is a content delivery network (CDN) that securely delivers data, videos, applications, and APIs to customers globally with low latency, and high transfer speeds, all within a developer-friendly environment.

Together, these services provide you with the necessary toolbox for designing, building, and managing the network infrastructure of web applications.



## AWS API Gateway

# Exposing Your Application's Endpoints to the Internet via API Gateway

## Introduction

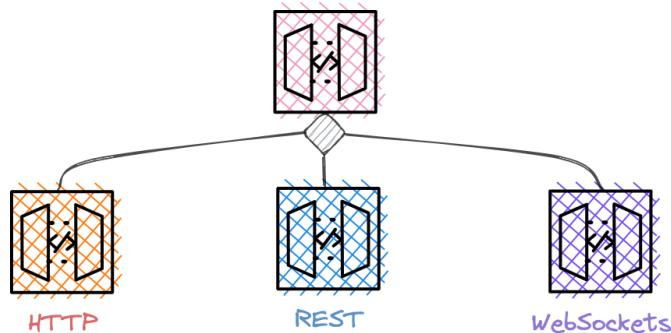
AWS API Gateway is another crucial part of Serverless applications on AWS. It's a fully-managed service that acts as a front door for the internet to your application's ecosystem.

Contrary to other methods that allow exposure to the internet, like Lambda function URLs, it's not only a simple way to trigger invocations via external sources but also does access control, data transformations, rate-limiting, and much more. It's also not limited to Lambda functions but many other AWS services or any backend that speaks HTTP.

API Gateway is beginner-friendly as it's easy to deploy your first API and there are no upfront or idling costs you need to fear.

## Rest, HTTP and WebSockets Are the Three Different Gateway Types

AWS API Gateway comes in three different flavors, with very different feature sets and different pricing. A small wrap-up before going into details:



- **REST** - a collection of HTTP resources and methods that map down to other AWS services like Lambda. It's the most common and flexible gateway type and does help you to do many otherwise tedious tasks like request validation or data transformations without writing any code.
- **HTTP** - comparable to REST, but comes with a reduced set of features. It's drastically cheaper though and easier to set up in the first place — a preferable choice for most Serverless applications.
- **WebSockets** - for building real-time applications with the publish/subscribe pattern. It

allows you to push messages via steadily open communication channels in both directions. AWS API Gateway does the heavy lifting by making connection management simple.

The right choice of gateway depends on your requirements.

#### **Rest Gateway for Full-Fledge APIs That Reduce Boilerplate Code in Your Backend Applications**

The high-value benefits of managed services are not only about not having to operate underlying infrastructure and therefore reducing your costs for operations. It's also about generally reducing the need for a lot of boilerplate implementation code that can be error-prone and conflicting in many ways.

And that's exactly what all AWS API Gateway types do very well as it comes with many no or low-code but high-value features that neither require deep knowledge nor long experiences.

#### **HTTP Gateway for Simple, Cheap, and Secure Applications Expose**

HTTP APIs are designed for a minimal feature set, increasing the speed to deploy your first API while not having to fear high costs per API call. As with REST Gateways, HTTP Gateways are also RESTful API products allowing you to map HTTP resources and methods to your AWS services.

#### **WebSockets for Real-Time Communication between Clients and Servers**

In web applications, we mostly think about clients like browsers or mobile apps that are making HTTP calls to exposed services and afterward waiting for the responses.

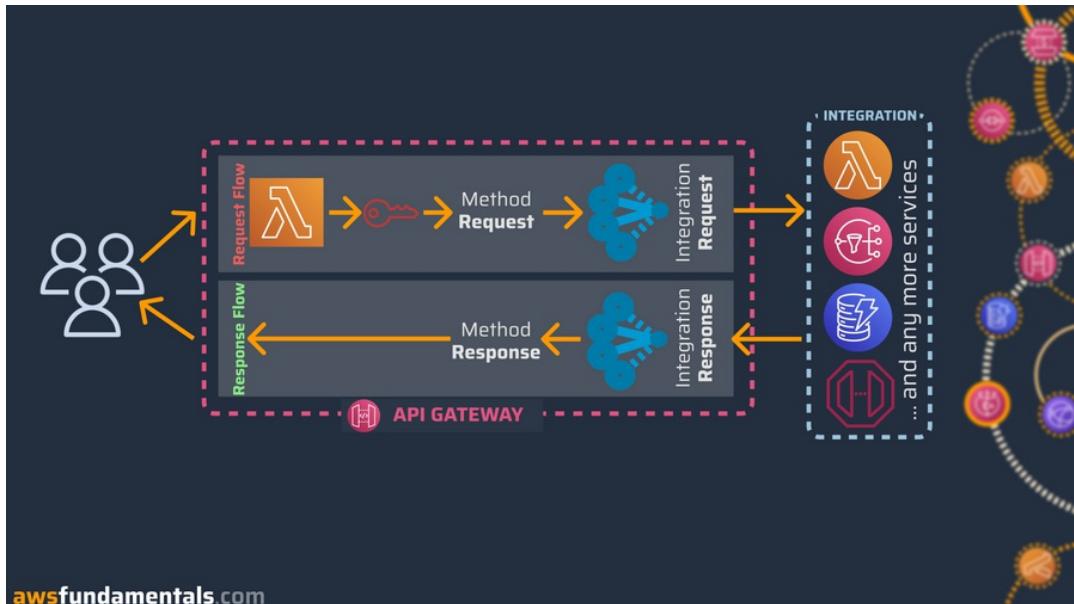
Depending on the connections between the client and the server, this includes low or high latencies as each request involves a complete roundtrip between the client and server. Also, connections are not steadily kept open, so initial requests can take longer due to TLS handshakes that are rather compute-intense. If clients are actively waiting for data to update, they need to regularly poll and cause load on your servers even if the data didn't update yet.

But that's not the only client-server communication option.

WebSockets enable you to keep steady connection channels between your application and your client that allow sending messages in both directions. This allows you to actively push messages to clients. With the publish/subscribe pattern, you can easily cluster clients to manage which data clients should receive.

## A Request Passes through Different Pre and Post-Integration Steps

As said before, API Gateway is not just an HTTP mediator - it's a feature-rich, high-value, and little error-prone front door to your applications ecosystem.



If looking at REST or HTTP gateways, an API is a collection of programmable **resources**. Those resources can map to one or several HTTP methods and to a path that is either very specific or does contain a proxy wildcard so it covers a set of paths.

Each of those resources needs to be connected to an **integration** endpoint to which a request will be forwarded. This can be for example a Lambda function or any HTTP-speaking endpoint. You can choose to either forward the request as-is or apply transformations that will modify the request in a way that the corresponding backend expects it to be. This can also include validations. After the response is received from the integration, you can apply response transformations before it's returned to the client.

A wrap-up about all those core capabilities which will be explored in detail in the next paragraphs:

- **Endpoint creation** - setting up an API and configuring HTTP routes and methods
- **Access Control** - authenticating and authorizing requests to protect your APIs and allow a secure multi-tenant usage of your application.
- **Integrations** - integrate your API method with a backend.

- **Request Validation** - validate your request before sending it to your destination and save the boilerplate code at your backends.
- **Data Transformations** - apply data mapping templates to automatically convert request data to expected inputs.
- **Gateway Responses** - apply mappings to convert outputs to expected responses.
- **CORS** - set up cross-domain rules to allow integration of your API endpoints in different web touchpoints.
- **Deploying APIs** - deploying your APIs to the internet.
- **Caching** - cache requests to lower latencies.
- **Monitoring** - track the requests to your API endpoints.

### **API Endpoints Contain Routes and Methods and Lead to Integrations**

After selecting the desired API Gateway type, the creation of a new endpoint is quick and easy. For example, looking at the HTTP gateway, we need to go through three configuration steps:

1. choosing integration - select how and with which backends or services we want to integrate, e.g. Lambda functions
2. configuring our routes - mapping HTTP methods and paths to our integrations
3. defining stages - we can run multiple stages in parallel and choose when to deploy which stages. HTTP gateway also offers auto-deploy - meaning that saved changes will automatically deploy to one or multiple stages

**Create an API**

**Create and configure integrations**

Specify the backend services that your API will communicate with. These are called integrations. For a Lambda integration, API Gateway invokes the Lambda function and responds with the response from the function. For HTTP integration, API Gateway sends the request to the URL that you specify and returns the response from the URL.

**Integrations** [Info](#)

Lambda	<a href="#">Remove</a>
AWS Region	Lambda function
eu-west-1	arn:aws:lambda:eu-west-1:157088858309:function: <input type="text" value="arn:aws:lambda:eu-west-1:157088858309:function:"> X
Version <a href="#">Learn more.</a>	
2.0	
<a href="#">Add integration</a>	

**API name**

An HTTP API must have a name. This name is cosmetic and does not have to be unique; you will use the API's ID (generated later) to programmatically refer to this API.

[Cancel](#) [Review and Create](#) [Next](#)

In the example, we choose a single Lambda function. The version indicator defines which gateway event type will be forwarded to the function. Version 1 and 2 have different JSON structures and version 2 is the default for the HTTP gateway.

You can find examples for both event types if you go to your Lambda function and choose to test the function in the console interface.

**Configure routes**

**Configure routes** [Info](#)

API Gateway uses routes to expose integrations to consumers of your API. Routes for HTTP APIs consist of two parts: an HTTP method and a resource path (e.g., GET /pets). You can define specific HTTP methods for your integration (GET, POST, PUT, PATCH, HEAD, OPTIONS, and DELETE) or use the ANY method to match all methods that you haven't defined on a given resource.

Method	Resource path	Integration target
ANY	/{{proxy+}}	prod-revue-api
<a href="#">Add route</a>		

[Cancel](#) [Previous](#) [Next](#)

After defining our integrations, we'll map routes to them. In our example, we'll map all HTTP

methods and all paths (using the `{proxy+}` selector) to a single function.

Step 1  
Create an API

Step 2  
Configure routes

Step 3  
**Define stages**

Step 4  
Review and create

**Configure stages** [Info](#)

Stages are independently configurable environments that your API can be deployed to. You must deploy to a stage for API configuration changes to take effect, unless that stage is configured to autodeploy. By default, all HTTP APIs created through the console have a default stage named \$default. All changes that you make to your API are autodeployed to that stage. You can add stages that represent environments such as development or production.

Stage name	Auto-deploy
\$default	<input checked="" type="checkbox"/>

Add stage

Cancel Previous Next

Finally, we define the stage. In our case, a single one with auto-deploy flipped on. By that, we don't need to worry about deployments as changes will be propagated automatically.

After going through **Review and create** your new API endpoint will be created and is available almost immediately.

## IAM Policies, Authorizers, Certificates, and Usage Plans to Protect or Restrict Access to Your Endpoints

API Gateway offers a bunch of different methods to control access to your endpoints.

### API Gateway Resource Policies

Resource policies - defined as JSON - enable you to allow or deny traffic to your gateway. You can use the policy to define who can access API gateway resources, such as methods and stages, and what action they can perform. You could for example grant read-only permission to all resources within the IP range of `192.0.2.0/24`. Additionally, it's only possible to use the `HTTP GET` or `OPTIONS` operation.

```
{  
  "Version": "2012-10-17",  
  "Statement": [  
    {  
      "Effect": "Allow",  
      "Action": "execute-api:Invoke",  
      "Resource": "arn:aws:execute-api:  
        <region>:  
        <account>:  
        <restapi_id>/<http_method>/<resource_path>"  
    }  
  ]  
}
```

```

    "Principal": "*",
    "Action": "execute-api:Invoke",
    "Resource": "arn:aws:execute-api:<regionId>:<accountId>:<apiId>/*/*/*",
    "Condition": {
        "IpAddress": {
            "aws:SourceIp": ["192.0.2.0/24"]
        },
        "StringEquals": {
            "aws:HttpMethod": ["GET", "OPTIONS"]
        }
    }
}
]
}

```

This means that any request that is not a GET request will be denied, regardless of the IP address of the client.

#### **IAM Policies to Grant Permissions to Invoke Other AWS Services**

API Gateways are often a central point that is used to connect and protect different parts of larger application ecosystems. Managing or creating API gateways requires IAM permissions, equal to all other actions you can take in your AWS account.

Additionally, it's possible to authenticate and authorize users for invocation of APIs via IAM policies. This requires the caller to submit their user's access keys. If the user has the proper permissions assigned - or is part of a group that does have the required permissions - the invocation is granted by the API gateway. For this, the authorization type of the gateway needs to be **AWS\_IAM**.

If you're using an integration that directly accesses another AWS service - like a proxy integration to a Lambda function - your API gateway also needs the required permission to invoke your function.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",

```

```
        "Action": "lambda:InvokeFunction",
        "Resource": "*"
    }
]
}
```

### Using the JWT Default Authorizer for an Easy OAuth2 Integration

OAuth2 and OpenID Connect are likely the most used authentication and authorization methods on the web. When using an identity provider like Autho, your client application will receive a token that represents the authentication as a JSON Web Token (JWT). This token is built-up of three different parts - the header, the payload, and the signature - separated via dots.

- **Header** - containing information that is needed to validate the signature of the token - generally speaking the algorithm (e.g. HS256) and the identifier of the key that was used to generate the signature.
- **Payload** - information about the token itself (e.g. **sub** for subject - the identity for which the token was issued - or **iat** for “issued at” - the Unix timestamp when the token was generated) and any custom information you want to transfer with the token.
- **Signature** - the signature that can be verified via the information from the header to assure that the token is valid and was generated by an authorized authority.

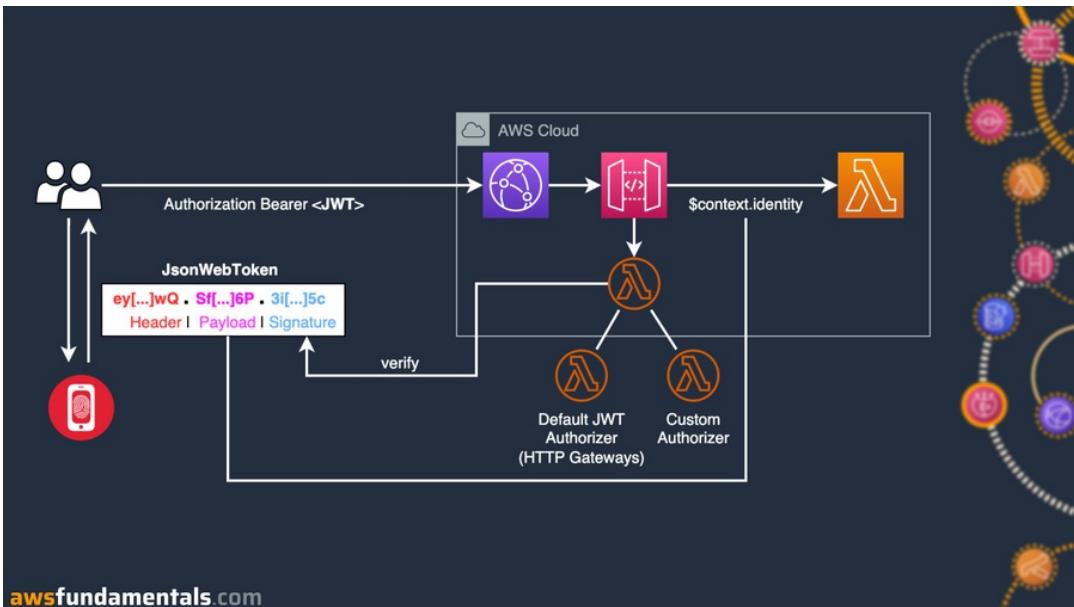
Your OpenID supporting identity provider exposes you to the so-called OpenID configuration endpoint that contains all the necessary information to validate incoming JWTs.

And that’s where the JWT Default Authorizer comes into play. It’s a functionality that is currently only supported by AWS HTTP Gateway and allows you to easily protect methods and routes without writing any code. The only configuration that is needed is the setup of the OpenID configuration URL and the routes and methods you want to protect.

API Gateway will check the Authorization header for a valid JWT.

1. The token needs to pass the signature validation. Each token contains the signature algorithm and the identifier of the key that was used to create the signature. This key will be retrieved by API Gateway from the OpenID configuration endpoint.
2. The token can’t be expired (the current date has to be before the timestamp set by `exp` in the payload of the token).

It will only forward requests that do pass both checks. Every other request will be declined with HTTP 401 *and is not billed*. And it doesn't end here: OAuth2 and OpenID Connect allow you to grant scopes to clients that are also included in your tokens and can be assigned to your routes and methods, meaning API Gateway will also check for the required scopes.



If a request is authenticated and authorized, the request will be enhanced with an authorization context and forwarded to your integration. For Lambda, this means that you can access your context and will find all the contents that were transferred via the payload of the token. In a multi-tenant application, you can now work with a dedicated user context even though haven't written any line of custom code yet.

#### Lambda Authorizers for Limitless Custom Authentication and Authorization

If you're using the REST gateway type, you can't make use of the default JWT authorizer. In this case, you have to rely on the Lambda Authorizer, which is a custom Lambda function you have to create yourself.

```
exports.handler = async function (event, context, callback) {
  const { authorizationToken } = event;
  // request authentication
  // [...]
  // Case 1 - authentication failed: return an error
  callback('Unauthorized', null);
  // Case 2 - authentication succeeded: return a policy that
```

```
// allows the invocation  
callback(undefined, <POLICY>);  
}
```

Your Lambda function doesn't undergo any restrictions. As far as you've assigned additional permissions you can invoke other AWS services or API calls as an example. If you're using OAuth2 you can rely on standard OpenSource libraries like `jsonwebtoken` to validate the signature of incoming tokens and forward any authentication context you wish.

### Mutual Authentication via Certificates and mTLS

API Gateway also comes with support for mutual TLS, which enforces authentication and authorization via the ownership of certificates. It requires you to add a custom domain name to your API gateway stage and disable the default `execute-api` endpoint that's generated by AWS. After uploading a `.pem` truststore file that contains one or multiple trusted certificates from certificate authorities it will fully support mTLS.

### Usage Plans for Rate-Limiting Access and Controlling Costs

AWS REST API Gateway comes with a great additional feature that is free of any charge: Usage Plans. Usage plans allow you to restrict the number of possible API calls to your gateways, not on a global level, but per client. Each client will get his own API key which he has to send in each request and the usage plan will take care of counting invocations. If invocations are exceeded for a specific client, further requests will be denied.

If usage plans are enabled for an API gateway, each request that doesn't send an API key or an unknown API key will be declined with HTTP 401 and is **not charged**. Please keep in mind that you can't create more than 10,000 API keys for a single usage plan.

### Integrations - Defining the Backends You Want to Invoke

Integrations are the target you want to actually reach via requests through your API gateway. You can select between different **integration types** which determine how method request data is passed to your desired backend. AWS differentiates between different integration types:

- **AWS\_PROXY**: For Lambda proxy integrations that connect to a single Lambda function. The request is forwarded as is via the default mapping. It's simple to set up and doesn't strictly couple your API gateway with the backend behind it.

- **AWS**: For custom Lambda integrations and all other AWS service integrations. This setup requires the setup of mapping templates and allows the integration of multiple endpoints. It also requires the definition of input and output data formats.
- **HTTP & HTTP\_PROXY**: Similar to the AWS\_PROXY & AWS setup, those two integrations work with HTTP endpoints and either forward the request as is to the backend or require mapping templates as well as input and output data formats.
- **MOCK**: This special integration type allows you to send responses without having the API gateway to forward the request to any backend. This is helpful for either testing or simulating or configuring proper cross-origin rules for OPTIONS requests.

## Validating Your Requests before They Reach the Invocation Target

API Gateway supports basic request validation which will result in immediate HTTP 400 responses if conditions are not met - without forwarding the call to the integration. Your gateway supports different types of validations:

- **Body** - applying rules to the payload of the request, based on the supplied content type. This includes required fields and their expected type (e.g. string or number).
- **Query parameters & headers** - validating the sent query string and headers.
- **Body, query parameters & headers** - both body & query string, and header combined.

### Method Execution / - ANY - Method Request

Provide information about this method's authorization settings and the parameters it can receive.

#### Settings

Authorization NONE  

Request Validator

NONE

Validate body

Validate body, query string parameters, and headers

Validate query string parameters and headers

API Key Required

#### URL Query String Parameters

For applying payload validation, you need to create a model with a corresponding content type, e.g. `application/json`. In our example, we're enforcing requests to send the `email` field as a string.

Models Create

- </> Empty Edit
- </> Error Edit
- </> requiredemail Edit

## Update Model

Make changes to your model in the form below. Models are declared using [JSON schema](#).

**Model name** requiredemail

**Content type** application/json

**Model description** Edit

**Model schema\*** Edit

```

1  {
2    "$schema": "http://json-schema.org/draft-04/schema#",
3    "type" : "object",
4    "required": [
5      "email"
6    ],
7    "properties": {
8      "username": {
9        "type": "string"
10     }
11   }
12 }
```

By assigning the model to our API gateway's method request, we're activating the validation as soon as the stage is deployed.

← **Method Execution** / - ANY - Method Request Edit

Provide information about this method's authorization settings and the parameters it can receive.

**Settings** ●

**Authorization** NONE Edit i

**Request Validator** Validate body, query string parameters, and headers Edit i

**API Key Required** false Edit

▶ URL Query String Parameters ●

▶ HTTP Request Headers

▼ Request Body Edit ●

Content type	Model name	
application/json	requiredemail <span style="border: 1px solid #ccc; border-radius: 5px; padding: 2px 5px;">Edit</span>	<span style="color: green;">✓</span> <span style="color: red;">✗</span>

The beauty of the validation is, that you don't have to write additional code, make use of libraries, or actually consume compute resources of your actual backend services. AWS API Gateway will take care of the validation without additional costs and automatically reject invalid requests which then won't trigger an integration request.

## Transforming Your Data to Meet the Integrations Expectations

Besides validations, API Gateway is also able to apply transformations. This helps to convert requests to an expected input format without changing anything on your actual integration. This helps to decouple the API Gateway from your actual backend, as you can for example integrate changes in your backend without affecting the integrated clients coming through your gateway.

It's one of the key features of AWS API Gateway.

### The Velocity Template Language - Programmatically Define How Data Has to Be Transformed

The incoming request payload and/or query parameters can be converted into a structure that's defined via the **Velocity Template Language (VTL)**.

As an imaginary example, we'll transform the incoming body of the JSON payload, include path details from the route, and also add authorization context details that are set by the previously executed Authorizer Lambda function.

What we're receiving at our route `/actions/buy/{productId}`:

```
POST /actions/buy/2138
{
  "items": 2
}
```

What we want to do:

1. Change the names of the incoming fields in the payload.
2. Add the identifier of the customer to the payload.
3. Add the customer identifier which is in the authorization context as `subject` to the payload.

What the transformation could look like:

```
#set($inputRoot = $input.path('$'))  
{  
    "customerId": "$context.authorizer.claims.sub",  
    "productId": "$input.params().path.get('productId')",  
    "numberOfProducts": "$inputRoot.items"  
}
```

Via `$input` you're able to access the request information, containing the payload, path, and query parameters. With `$context` you can access the authorization context provided by the Authorization Lambda.

Transformation templates can be hard to build, read, and even maintain. But they do not require writing any code. They do not result in additional billing. They don't rely on your computing resources.

Data transformations can be also applied to responses, meaning you're able to transform responses from your integrations to the expected format of your clients. It's also possible to assign different transformations based on the HTTP status code.

## Handling Invocation and Validation Errors

With gateway responses, you're able to define what happens if the gateway fails to process an incoming request. Instead of returning the default API gateway error if a Lambda execution fails or throws an unexpected error, you can return a response in the same format as your success invocations.

Let's try this by focusing on exactly those Lambda execution errors.

At first, we need to create a method response with an HTTP status - we're choosing HTTP 500 as we want to map all the invocation errors for Lambda. At this point, it's also possible to define a model that explicitly defines how the response has to look. In comparison to the model for the request, it's not used for validation but only for telling clients about the contents of the response.

## [Method Execution](#) / - ANY - Method Response [Edit](#)

Provide information about this method's response types, their headers and content types.

HTTP Status	
▶ 200	<a href="#">Edit</a> <a href="#">Delete</a>
▼ 500	<a href="#">Edit</a> <a href="#">Delete</a>

### Response Headers for 500

Name	
No headers	

[+ Add Header](#)

### Response Body for 500 [Edit](#)

Content type	Models	
application/json	Empty	<a href="#">Edit</a> <a href="#">Delete</a>

[+ Add Response Model](#)

Next, we want to map all **HTTP 5xx** responses from our Lambda invocation to the HTTP 500 method response we created. For this, we can use a regular expression that will look for all status codes starting with 5 and therefore include for example HTTP 500 (invocation error) or HTTP 503 (timeout).

## [Method Execution](#) / - ANY - Integration Response [Edit](#)

First, declare response types using [Method Response](#). Then, map the possible responses from the backend to this method's response types.

	Lambda Error Regex	Method response status	Output model	Default mapping	
-					

Map the output from your Lambda function to the headers and output model of the method response.

**Lambda Error Regex**  [?](#)

**Method response status**  [?](#)

**Content handling**  [?](#)

[Cancel](#) [Save](#)

This integration response can then also make use of a mapping template to extract the

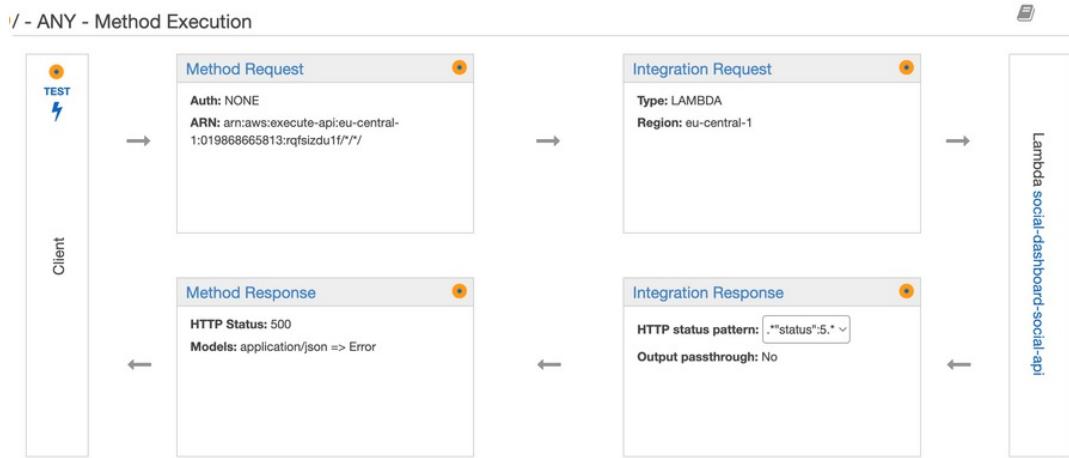
information we want to pass in the final gateway response.

For this, we need to know what invocation errors and timeouts look like at Lambda:

```
{  
  "errorMessage": "<replaceable>string</replaceable>",  
  "errorType": "<replaceable>string</replaceable>",  
  "stackTrace": [  
    "<replaceable>string</replaceable>,  
    ...  
  ]  
}
```

We're mainly interested in the `errorMessage` so we can create a simple template mapping in our integration response with `$input.path('$.errorMessage')`.

This will now only extract the error message into a simple string that is later returned via the gateway response in the response payload.



That's it. Lambda execution errors like timeout will now be converted to a response that only contains the error message.

### Dealing with Cross-Origin Requests or “Not Fearing CORS”

If you're already experienced in the web developer area, you've encountered cross-origin issues with AJAX requests. Maybe you've copied a StackOverflow response to some part of your code to make the errors go away.

Let's briefly talk about what CORS is and how to properly configure it at your API gateways.

## Cross-Origin Resource Sharing 1x1

CORS isn't a bad thing, it's a good thing. It circumvents the policy of only allowing requests to the same domain you're currently browsing a website at. The server can't enforce rules but only supplies information to clients. If the client is a browser, it will take care of applying those rules.

As a simplification, you can imagine this as a dialog between two parties:

- **Browser** - "Hi backend offering services on domain **B**, I'm executing a script on domain **A** that wants to execute a **POST** to your domain. Is this allowed?"
- **Backend** - "Yes, Domain **A** is allowed to make GET & POST requests to my domain."

It will send the HTTP request with two additional headers to the target domain:

- **origin** - the domain the script is executed on; in our case: domain **A**.
- **Access-Control-Request-Method** - the HTTP method of the target request; in our case: **POST**.

If the server allows this domain & HTTP method combination, it will respond with an HTTP 200. The browser will then continue to send the actual request.

## Properly Configuring Your Gateway for CORS

Configuring cross-origin rules for your HTTP or REST gateway is equally simple. Its configuration resides in a designated tab.

## Applying CORS Rules to HTTP Gateway

After opening your gateway, you'll find a **CORS** section on the left. There you'll find the configuration that will be automatically applied to all of your gateway routes and methods. You can configure multiple origins and headers or just use the wildcard \* to allow integration at any domain.

## Cross-Origin Resource Sharing

**Configure CORS** [Info](#)

CORS allows resources from different domains to be loaded by browsers. If you configure CORS for an API, API Gateway ignores CORS headers returned from your backend integration. See our [CORS documentation](#) for more details.

**Access-Control-Allow-Origin**

**Access-Control-Allow-Headers**

**Access-Control-Allow-Methods**

**Access-Control-Expose-Headers**

**Access-Control-Max-Age**

**Access-Control-Allow-Credentials**  NO

**Configure** **Clear**

### Setting up CORS for Your Rest Gateway

It's equally easy at a REST Gateway. After clicking on the corresponding field in the navigation, you can configure your rules. The interface is not as shiny as at HTTP Gateway, so you need to split multiple headers and origins in the same input field via commas.

Enable CORS

Gateway Responses for **test API**  DEFAULT 4XX  DEFAULT 5XX [?](#)

Methods  OPTIONS [?](#)

Access-Control-Allow-Methods  [?](#)

Access-Control-Allow-Headers  [?](#)

Access-Control-Allow-Origin\*  [?](#) [!](#)

Advanced

**Enable CORS and replace existing CORS headers**

After clicking on the enable button, AWS API Gateway will apply your rules by creating the corresponding proxy route with the OPTIONS method and assigning them mock integrations if the requests match the expectation for your rules.

## Enable CORS

```
✓ Create OPTIONS method
✓ Add 200 Method Response with Empty Response Model to OPTIONS method
✓ Add Mock Integration to OPTIONS method
✓ Add 204 Integration Response to OPTIONS method
✓ Add Access-Control-Allow-Headers, Access-Control-Allow-Methods, Access-Control-Allow-Origin Method Response Headers to OPTIONS method
✓ Add Access-Control-Allow-Headers, Access-Control-Allow-Methods, Access-Control-Allow-Origin Integration Response Header Mappings to OPTIONS method
Your resource has been configured for CORS. If you see any errors in the resulting output above please check the error message and if necessary attempt to execute the failed step manually via the Method Editor.
```

And that's already all you need to do.

## Deploying Your Endpoints to the Internet

With the HTTP gateway, you'll receive a default stage that can be deployed automatically after each change you took in the configuration. With REST gateway, each of your stages needs to be deployed manually.

Think about a stage like a configuration of your gateway. You can have several in parallel and you can map a stage to the default execution API domain or your custom domain. If you have several stages - like development and production - you can assign them to different path prefixes.

## Configuring Custom Domain Names to Increase Trust in Your APIs

You're not forced to use the `execute-api` endpoint URLs provided by AWS but use your own custom domain names. This requires you to have the ability to create DNS entries for your domain.

It's not a requirement to use Route 53 as your domain name service here. The only thing that's required is to create your records pointing to the API gateway endpoint URL from AWS and to create a (free) certificate for your domain in AWS Certificate Manager (ACM). If your domain is managed by another provider, e.g. Cloudflare, this is perfectly fine.

An **important notice beforehand**: if you're using a regional endpoint for your API gateway, you need to create your certificate in your target region - if using edge-endpoint type you always have to create your certificate in **us-east-1**. ACM for public certificates is free of charge, so you don't have to worry and you can create the certificate for the same domain name in **any region** at the **same time**. They can co-exist.

You can quickly create a certificate in ACM via **Request** and choosing **public certificate** with DNS validation. Enter the domain you want to cover - this can include alternative domain names, e.g. by adding `*.awsfundamentals.com` the certificate can be used for any subdomain - and submit the request.

Certificates (1)						
	Certificate ID	Domain name	Type	Status	In use	Renewal eligibility
<input type="checkbox"/>	6a94316c-648c-4a5b-a652-9a73f45a518c	awsfundamentals.com	Amazon Issued	<span>Issued</span>	Yes	Eligible

AWS will prompt you to either create the necessary DNS validation records in Route53 automatically (if you want to use Route53) or to create the entries manually, e.g. if you're using another provider.

Shortly after creating the records, ACM will notice their existence and set the status to **Issued** which immediately allows you to use your new certificate all across AWS, including our AWS API Gateway.

When switching back to your gateway, you can now create the custom domain name and map an API deployment to it. After this has finished, the last step is to create an A record at your DNS provider pointing to the AWS-provided API gateway URL.

### Caching Your Requests to Decrease Latencies

API Gateway comes with a caching feature out of the box. This is not free of charge and the price depends on the cache size. Caching is also **charged by the hour** and **is not eligible for the AWS Free Tier**.

When caching is enabled, API Gateway will store responses for a defined period of time. Consecutive requests that map down to the same key, e.g. the path of your request combined with the HTTP method, will return the results from the cache instead of invoking the integration.

For REST gateways, caching has to be enabled per stage. Select your API gateway, then go to `Stages` and select your desired stage. In the settings tab, you'll find `Enable API cache`.

If you enable caching, the API gateway will create its own distributed cache (this will take several minutes) that will be used to store responses for GET requests. Subsequent requests will then result in retrieving information from the cache instead of calling the integration.

Using caches properly is not an easy task. Improper cache keys or cache settings can lead to delivering stale (outdated) data which can cause additional side effects. Therefore it's important to think about your cache key and which parts of the request are taken into building

the cache key.

Let's look at an example query, that is retrieving orders of an authorized user:

```
GET /orders?status=...
Authorization: Bearer eyXf...
```

The results of the request will always depend on the user that took the request and the status of the order. Two requests from the same user that are requesting different order statuses should not return the same results. Furthermore, different users should never receive the same result due to a cache hit.

We need to at least include the query parameter `status` and the `Authorization` header as a cache key.

### **Monitoring APIs to Immediately Get Aware of Issues**

By default, CloudWatch collects different metrics for your API gateway, including the number of requests, the latency of your integration, and the number of HTTP 4xx and 5xx responses.

Additionally, you can set up CloudWatch logging for your gateways to debug issues related to request execution. Execution logging will result in the gateway managing CloudWatch logs and streams and reporting information about requests and responses. This includes errors, execution traces, data used by authorizers, and more.

For REST gateways, the log group name will have the following pattern: `API-Gateway-Execution-Logs_{rest-api-id}/{stage_name}`. Depending on the number of requests to your API endpoint, enabled logging can result in costs for the storage of logs in CloudWatch. It's recommendable to immediately set a retention period to your log to expire automatically after a given period of time.

### **Building Real-Time Communication Applications with WebSockets**

The WebSockets API Gateway is managed service that enables you to build applications that support real-time, bi-directional communication via the WebSockets protocol. Both clients and servers can send messages in both directions in real-time. Instead of sending individual requests that often require opening a new HTTP connection, a socket connection can be kept open to send messages immediately.

AWS WebSockets Gateways take over the critical part of maintaining the connections. You don't need to know much or anything about how the connections work, as the API Gateway will

solely manage the connections to the clients. On each connection event, e.g. if a connection was closed due to a network issue, API Gateway will fire an event to a dedicated Lambda function so we can take action.

WebSockets API Gateway also provides features such as authentication, authorization, monitoring, and logging to help you build and manage reliable and secure WebSocket APIs.

Let's jump into the fun and create our first WebSockets gateway.

Compared to other examples in this book, this part is maybe not trivial and does involve a lot of steps. Don't feel bad if something doesn't work out in the beginning!

### Creating A WebSocket API Gateway

Go to the API Gateway Management Console and click on `Create API`. If you don't have any gateway in your current region, you'll see an overview with a `build` button for each of the different types.

We need to specify the name of our gateway and the route selection expression. This path will be later used by the API gateway to forward the type of action to our function. We can stick to the default of `request.body.action`.

The screenshot shows the 'Specify API details' step of the AWS API Gateway creation wizard. It has two main sections: 'API name' and 'Route selection expression'.

**API name:**  
The 'API name' field contains the value 'awsfundamentals-websockets'. A note below says: 'The name is cosmetic and does not have to be unique.'

**Route selection expression:**  
The 'Route selection expression' field contains the value '\$ request.body.action'. A note below says: 'A route selection expression tells API Gateway which route to call when a client sends a message.'

At the bottom, there are three buttons: 'Cancel', 'Review and create', and 'Next' (highlighted in orange).

Next, we have to define which keys should be used for the different event types. There are

always three predefined routes we have to support:

- `$connect` - triggered when a new client connects to our gateway
- `$disconnect` - trigger when a client disconnects, either on purpose or by an interrupted network connection
- `$default` - if no matching route is found

Additionally, we can define our own routes. For now, let's stick to the predefined routes only and continue with the setup.

## Add routes

API Gateway uses routes to expose integrations to clients. API Gateway evaluates the route selection expression of your API at runtime to determine which route to invoke.

**Predefined routes** [Info](#)

The `$connect` route is triggered when a client connects to your API.

**Route key**

`$connect` [Remove](#)

The `$disconnect` route is triggered when either the server or the client closes the connection.

**Route key**

`$disconnect` [Remove](#)

The `$default` route is triggered if the route selection expression can't be evaluated against the message or if no matching route is found.

**Route key**

`$default` [Remove](#)

**Custom routes** [Info](#)

Add custom routes to invoke integrations based on message content.

[Add custom route](#)

[Cancel](#) [Previous](#) [Next](#)

Each of our routes needs to be attached to either an HTTP integration, a mock, or a Lambda function. We want to handle all of our connection events with a single Lambda function. So let's open a new tab for the Lambda console and create a simple Node.js Lambda function first.

The function's code can be very simple for now. Let's only log the incoming event so we can see what the API gateway will forward.

```
export const handler = async(event) => {
    console.log(JSON.stringify(event));
};
```

Afterward, jump back to our API Gateway wizard and chose our new function for each of the connection events.

## Attach integrations

To deploy this API, you must set up at least one route. All routes that you set up must have an integration attached. To set up integrations later, select Mock as the integration type for your routes. [Info](#)

The screenshot shows the 'Attach integrations' step of the API Gateway wizard. It displays three separate configuration sections:

- Integration for \$connect:** Set to Lambda integration type, AWS Region eu-west-1, Lambda function arn:aws:lambda:eu-west-1:157088858309:function:we1
- Integration for \$disconnect:** Set to Lambda integration type, AWS Region eu-west-1, Lambda function arn:aws:lambda:eu-west-1:157088858309:function:we1
- Integration for \$default:** Set to Lambda integration type, AWS Region eu-west-1, Lambda function arn:aws:lambda:eu-west-1:157088858309:function:we1

At the bottom right, there are 'Cancel', 'Previous', and 'Next' buttons.

When clicking **Next** you'll be taken to a final overview of the settings that will be applied to our new gateway. Confirm it to create our first WebSockets gateway.

Now we already have a working WebSockets gateway that will accept new connections, keep

them alive, and will forward every connection even to our new Lambda function.

Let's click on the `Stages` tab and our one and only stage `production` to get the URL to connect to our API gateway. It's prefixed with `wss`.



The screenshot shows the AWS API Gateway interface. In the top navigation bar, it says "Amazon API Gateway APIs > awsfundamentals-websockets (qu91138z2g) > Stages > production". Below this, there are tabs for "APIs", "Stages", and "Create". The "Stages" tab is selected, showing a list with "production" highlighted. On the right, the "production Stage Editor" is open. It displays two URLs: "WebSocket URL: wss://qu91138z2g.execute-api.eu-west-1.amazonaws.com/production" and "Connection URL: https://qu91138z2g.execute-api.eu-west-1.amazonaws.com/production/@connections". There are also buttons for "Delete Stage" and "Configure Tags".

Let's directly test the workings by creating a local Node.js script. Let's jump into a new empty directory:

```
# init a new project
npm init

# install the web sockets package
npm i ws

# create a new script file that will be our client
touch client.js
```

Open the `client.js` in your favorite editor and put in a simple script that will only open a new connection to our gateway.

```
const WebSocket = require("ws");

const run = async () => {
  const webSocket = new WebSocket(
    "wss://qu91138z2g.execute-api.eu-west-1.amazonaws.com/production"
  );
  webSocket.onopen = () => {
    console.log(`Connected to ${webSocket.url}`);
  };
  await new Promise((resolve) => setTimeout(resolve, 120000));
};

run();
```

Start the script via `node client.js` and you should see our success message as it should be able to connect to the gateway. We've added a blocking promise to not finish the script execution immediately.

Let's check the outputs of our Lambda function by going to its corresponding CloudWatch log group and the latest log stream. You'll find the event we received from the gateway.

```
{  
  "headers": {...},  
  "multiValueHeaders": {...},  
  "requestContext": {  
    "routeKey": "$connect",  
    "eventType": "CONNECT",  
    "connectionId": "dmwL5duMDoECJPQ=",  
    ...  
  }  
}
```

I've cut out everything that's not important to us right now. What we're interested in is the `routeKey` and the `connectionId`. The connection identifier is a unique identifier for this connection and is needed to send messages to this exact client.

Let's stop the script and check which event we receive if our self-built client has disconnected.

```
{  
  "headers": {...},  
  "multiValueHeaders": {...},  
  "requestContext": {  
    "routeKey": "$disconnect",  
    "disconnectStatusCode": 1001,  
    "eventType": "DISCONNECT",  
    "disconnectReason": "Going away",  
    "connectionId": "dmwL5duMDoECJPQ=",  
  }  
}
```

We see that our expected connection identifier and the route key `$disconnect`. We also see the status code and reason for the disconnect.

### Using DynamoDB to Store Connection Information

We're now able to receive events about opening and closing connections. But how do we work with open connections? We need to store the connection identifiers first.

DynamoDB, as a fully-managed key-value store, is a perfect choice for this task. Let's create a new simple table that only stores our connection information.

## Create table

**Table details** [Info](#)

DynamoDB is a schemaless database that requires only a table name and a primary key when you create the table.

**Table name**  
This will be used to identify your table.  
  
Between 3 and 255 characters, containing only letters, numbers, underscores (\_), hyphens (-), and periods (.)

**Partition key**  
The partition key is part of the table's primary key. It is a hash value that is used to retrieve items from your table and allocate data across hosts for scalability and availability.  
 ▼  
1 to 255 characters and case sensitive.

**Sort key - optional**  
You can use a sort key as the second part of a table's primary key. The sort key allows you to sort or search among all items sharing the same partition key.  
 ▼  
1 to 255 characters and case sensitive.

We'll pick `connectionId` as the partition key and we don't need a sort key.

Next, we jump back to our connection management Lambda function and go to its execution role policy ([Configuration > Permissions > Clicking on the role name link](#)). Let's add the needed statement to the JSON policy to insert and delete items in our table.

```
{  
    "Sid": "DynamoDBAccess",  
    "Effect": "Allow",  
    "Action": [  
        "dynamodb:PutItem",  
        "dynamodb>DeleteItem",  
        "dynamodb:Scan"  
    ],  
    "Resource": [  
        "arn:aws:dynamodb:eu-west-1:157088858309:table/ws-connections",  
    ]  
}
```

We'll also add the scan option, so we have access to all of the table's currently saved connections so we can later broadcast messages to all connected clients.

Let's adapt the `index.js` file of your Lambda function that handles our connections. Let's do this locally by creating a new `index.js` file with the following contents:

```
const AWS = require("aws-sdk");

const dynamoDb = new AWS.DynamoDB.DocumentClient({
    region: process.env.AWS_REGION,
});

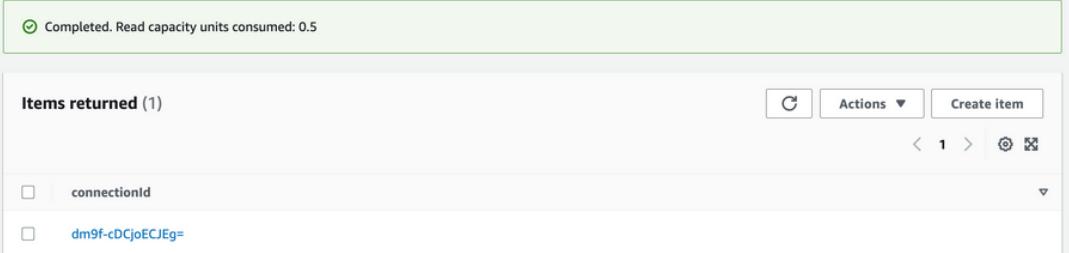
const TableName = "ws-connections";

exports.handler = async (event) => {
    if (!event.requestContext || !event.requestContext.routeKey) return;
    const { routeKey, connectionId } = event.requestContext;
    console.log(`Received ${routeKey} event for ${connectionId}`);
    switch (routeKey) {
        case "$connect":
            await dynamoDb.put({ TableName, Item: { connectionId } }).promise();
            break;
        case "$disconnect":
            await dynamoDb.delete({ TableName, Key: { connectionId } })
        ).promise();
            break;
        default:
            console.log(`Unknown routeKey: ${routeKey}`);
    }
    console.log(`Finished ${routeKey} event for ${connectionId}`);
    return {
        statusCode: 200,
    };
};
```

We can now upload our function by copying the code into the Lambda console.

Open a new browser tab afterward and jump to our new DynamoDB table. By clicking on `Explore table items` on the right we can go into the interactive search console to scan for our items. We should see an empty table for now.

Let's start our client again via `node client.js` and have a look at our table again. We'll see a new item!



The screenshot shows the AWS Lambda function table interface. At the top, there is a green status bar with the message "Completed. Read capacity units consumed: 0.5". Below it, the table header says "Items returned (1)". There is one item listed with the key "connectionId" and the value "dm9f-cDCjoECJEg=". On the right side of the table, there are buttons for "Actions" and "Create item", and navigation controls for the table.

If we close our client by interrupting our script (e.g. by pressing `cmd+c` on macOS), we'll immediately see that the item is gone again after running another scan.

That's exactly what we wanted. Our table now always contains all connections that are currently active.

### Sending And Receiving Messages Between Clients

The fundamental operation part is done now. Let's jump to the interesting: actually sending messages between connected clients.

What do we need for this?

1. we need to adapt our minimal client script to be able to send messages and listen to received messages.
2. we need to extend our Lambda function attached to our function to handle incoming messages from connected clients.

For the first part, let's send a message every 5 seconds that contains the name of our client. This could look like the following:

```
const WebSocket = require("ws");

const name = process.argv[2];

const run = async () => {
  const webSocket = new WebSocket(
    "wss://qu91138z2g.execute-api.eu-west-1.amazonaws.com/production"
  );
  webSocket.onopen = () => {
```

```

        console.log(`Connected to ${webSocket.url}`);
        setInterval(() => webSocket.send(`Hello from ${name}!`)), 5000);
    };
    webSocket.onmessage = (event) => {
        console.log(`Received: ${JSON.parse(event.data).message}`);
    };
    await new Promise((resolve) => setTimeout(resolve, 120000));
};

run();

```

We can now start our client with an argument that indicates the name of our client by typing `node client.js first-client`. For each new terminal tab, we can start another client with a new name!

If we'd log out of the received event in our Lambda function, it would look like the following:

```
{
  "requestContext": {
    "routeKey": "$default",
    "messageId": "dnA-ietxDoECE2g=",
    "eventType": "MESSAGE",
    "connectionId": "dnA-gettDoECE2g=",
    ...
  },
  "body": "Hello from first-client!",
  "isBase64Encoded": false
}
```

We see that the event type is now `MESSAGE` and it's handled by the default route key of `$default` as we didn't specify a custom route here. We also see the message we submitted.

Now let's do the backend part by adapting our Lambda function and actually sending the message back to all connected clients.

```

const AWS = require("aws-sdk");
const dynamoDb = new AWS.DynamoDB.DocumentClient({
  region: process.env.AWS_REGION,
});
const apiGatewayClient = new AWS.ApiGatewayManagementApi({

```

```

    endpoint: "https://qu91138z2g.execute-api.eu-west-
1.amazonaws.com/production",
});

const TableName = "ws-connections";

exports.handler = async (event) => {
  if (!event.requestContext || !event.requestContext.routeKey) return;
  const { routeKey, connectionId } = event.requestContext;
  console.log(`Received ${routeKey} event for ${connectionId}`);
  switch (routeKey) {
    case "$connect":
      await dynamoDb.put({ TableName, Item: { connectionId } }).promise();
      break;
    case "$disconnect":
      await dynamoDb.delete({ TableName, Key: { connectionId } })
    ).promise();
      break;
    default:
      console.log(`Default routeKey ${routeKey} received.
Broadcasting...`);

      const { body: message } = event;
      const connectionIds = await getConnectionIds();
      for await (const id of connectionIds) {
        if (id === connectionId) continue;
        await sendMessage(id, message)
      };
    }
    console.log(`Finished ${routeKey} event for ${connectionId}`);
    return {
      statusCode: 200,
    };
};

async function sendMessage(ConnectionId, message) {
  console.log(`Sending message to ${ConnectionId}: ${message}`)
  await apiGatewayClient
    .postToConnection({

```

```

        ConnectionId,
        Data: message,
    })
    .promise();
}

async function getConnectionIds() {
    return dynamoDb
        .scan({ TableName })
        .promise()
        .then((result) => result.Items.map((item) => item.connectionId));
}

```

We're now broadcasting a received message to all clients, except for the client that actually sent the message.

For posting messages to the gateway, we need another permission for our Lambda function. It's the action `execute-api:ManageConnections`. Jump to your functions execution role and adapt the policy to include the required permission. For simplicity reasons, you can stick to allowing access to all resources instead of adding our gateway's ARN.



Remember that you can use `postToConnection` not only from our connection management Lambda but from every resource that has the necessary `ManageConnections` permission for your WebSockets gateway. This means you can actively send messages to connected clients from any backend.

Let's try it out by running to clients in multiple terminal tabs via `node client.js first-client`, `node client.js second-client`, and `node client.js third-client`

We'll see the following outputs from `first-client`:

```

Received: Hello from third-client!
Received: Hello from second-client!
Received: Hello from third-client!

```

```
Received: Hello from second-client!
```

The other clients should also receive messages from others, but not from themself.

If you made it here, this is awesome. You've put together a lot of small parts that now work together and can be used to build great applications!

### **What Costs to Expect with the Different API Gateway Types**

AWS API Gateway runs on a pay-per-use model, making it very beginner friendly as you don't need to worry about hourly charges for unused gateways.

If we exclude caching, the pricing is solely per request. Looking at current rates, in 2023 the following rates are important to remember:

- **REST:** \$3.5 per 1m requests.
- **HTTP Gateway:** \$1 per 1m requests.
- **WebSockets:** \$1 per 1m messages & \$0.25 per 1m connection minutes.

Examples are for `us-east-1` and prices can vary slightly per region.

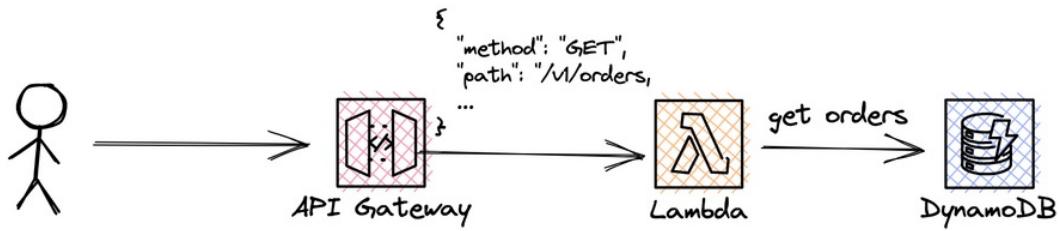
What really stands out here: if the features of HTTP gateway cover your needs, always stick to HTTP gateway as it's more than 71% cheaper than REST.

### **Use Cases for AWS API Gateway**

We've learned that AWS API Gateway is not a simple front door for applications. It's an enabler for other great features and plays a key role in a lot of applications and ecosystems. Let's explore some of its real-world use cases.

#### **Use Case 1: Exposing Your Serverless Applications to the Internet**

The typical web application is a client-server combination: a frontend - regardless if it's a web page or mobile app - that communicates via a backend via HTTP. On the server side, your backend is a construct of one or multiple Lambda functions that include your business logic.



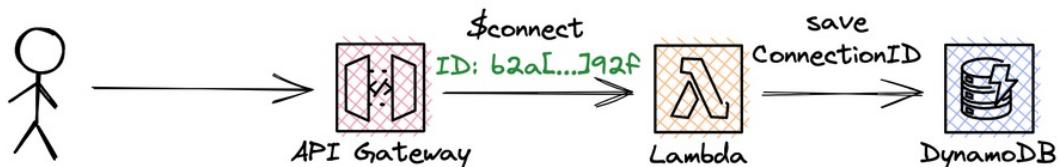
Exposing those functions to the internet is done via a REST or HTTP API Gateway. This allows you to actively enforce rate-limiting, request validation and transformation, routing, authentication, and authorization without writing much or any code at all.

#### Use Case 2: Real-Time Communication Between Your Clients and Your Backends

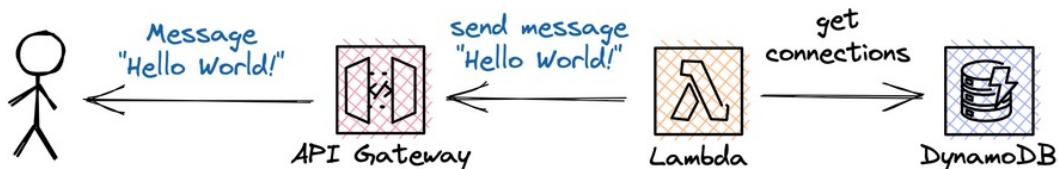
This was our hands-on example, but let's revisit it quickly.

Many applications require real-time communication between the clients and the backend. For messaging apps, you don't want all your clients to actively poll for new messages every few seconds. It causes unnecessary server load and also causes a high latency between a message that has been sent and the actual delivery at the receiver.

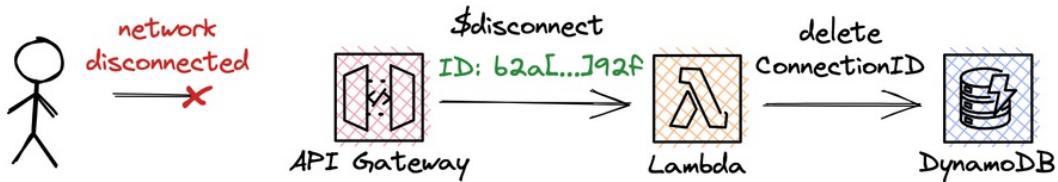
With WebSockets API Gateway a client can open a connection to the gateway that is fully managed by the gateway itself. The only thing that needs to be actively implemented is how connection events are handled and what you want to do with those connections. A common pattern is to save the connection identifiers that are provided by the gateway to DynamoDB.



Via a connection identifier, you can actively push messages to the corresponding client.



When the client disconnects, either on purpose or via a network timeout, API Gateway will forward a disconnect event to your function so you can take care of deleting the connection identifier from DynamoDB.



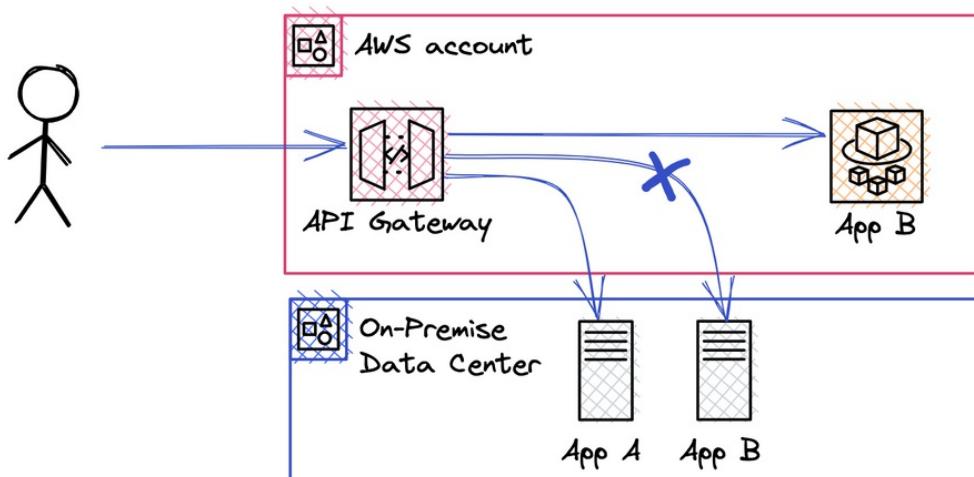
This pattern can be used for a lot of different applications and doesn't need any infrastructure management on our side.

### Use Case 3: Using the Lift and Shift Approach to Migrate Applications to the Cloud

Many open employee positions for developers are created for projects that want to be migrated to the cloud. This is not a trivial task for applications that are already in use by customers and can't be confronted with much or any downtime.

With the lift-and-shift approach and the help of AWS API Gateway, you can slowly move existing applications to the cloud without making any major changes to the application itself.

At first, you can create your API Gateway with a proxy integration that only forwards requests to your existing on-premise application. You can either use your existing domain name by shifting the DNS entry to your new API gateway or migrate all your clients to the new domain. After that, you can start to build your application infrastructure in the cloud, e.g. by creating a cluster in ECS that will run your application in containers. As you're now in full control of how requests are routed to all applications, you don't have to shift everything at once but step by step.



If you've finished migrating a part of your application, you can start routing requests (e.g. based on the prefix of the path) to the new destination.

#### **Use Case 4: Monetizing APIs by Setting up a Billing Plan**

In the later stages of a successful product, it's common to allow for third-party integrations by other developers. They can then start to use your APIs to build new features or even whole applications on top of your existing infrastructure and data. This can help to drive your product further and increase its range via new features you didn't even think of.

It's also common to charge developers for access to APIs, either on a pay-per-use basis or using a subscription model. Billing plans are part of the Usage Plan feature from API Gateway and can be individually configured for each client.

#### **Tips and Tricks for the Real World**

AWS API Gateway is a powerful service with a lot of features and it's not easy to remember all of them. Let's quickly jump through some small, big, and rather unknown key features.

- You can use the `Import from Swagger` feature to quickly create an API Gateway API from a Swagger definition file.
- The `Mock` integration types allow you to quickly create a gateway without specifying a backend integration.
- Integrating with an identity provider that supports OAuth2 is very easy when using HTTP Gateway and selecting the JWT Default Authorizer. But API Gateway is not limited to that: a custom Lambda Authorizer can be created to enable any authentication and authorization mechanism. Also: API Gateway supports mutual TLS.
- It's easy to attach a custom domain name to your gateway if you have a certificate that is managed by AWS Certificate Manager. For regional gateway endpoints, the certificate has to reside in the same region as your endpoint; for edge endpoints, it has to be in `us-east-1`.
- You can use the "API Gateway stage variables" feature to store and retrieve variables at runtime, such as environment-specific configurations or feature flags.
- API Gateway helps you to reduce boilerplate code with different features like request validation and transformation. This not only lowers costs by reducing invocations to your backend APIs but also certainly avoids a lot of bugs as it reduces the lines of codes you need to write and maintain.
- Usage Plans come for free. If you want to provide your API to different consumers, you

can easily monitor and limit the usage of your API via quotas.

## Final Words

API Gateway is a powerful and flexible service that can help you build and manage APIs for a wide range of use cases. Whether you're building a public API for your business or creating an internal API for your team, API Gateway can help you get started quickly and scale your APIs as needed.



Amazon **Route53**

# Making Your Applications Highly Available with Route 53

## Introduction

Introduced by Amazon in 2010, Route 53 is a managed service to reliably redirect traffic via domain names to your applications.

It doesn't matter if it's an S3 bucket, a Load Balancer, an API Gateway with Lambda functions, or even an on-premise server behind it.

## Understanding the Fundamentals of the Domain Name Service

Naming is fundamental in any distributed ecosystem as it helps to identify entities. As the internet is by far the largest distributed system, this concept is crucial.

In a nutshell: with the internet's DNS, we get human-friendly hostnames (e.g. `aws.amazon.com`) for computer-friendly IP addresses (e.g. `13.35.252.74`).



A common example to visualize this is to think of it as a phone book for the internet. If you want to call somebody, you need some kind of directory that helps you find their number.

## The Domain Name System Is a Decentralized System

The special thing about DNS management is its decentralized control. This enables its high scalability and avoidance of a single point of failure. Furthermore, it has a hierarchical structure that is maintained by registries in different countries.

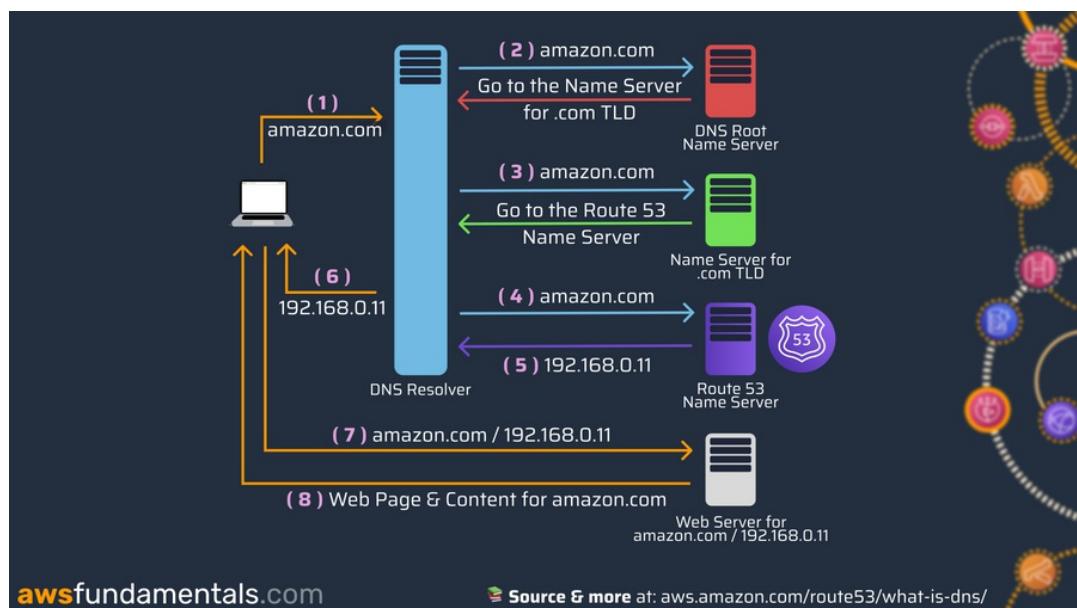
On top of the hierarchy are the authoritative name servers which do hold the actual addresses for the DNS records.

When we go down we also find top-level domain servers, root name servers & recursors.

## Multiple Services on Different Levels Take Part in Resolving Domain Names

When resolving domain names, we need to distinguish between two types of services:

- **Authoritative DNS Service** - actually responds to DNS queries and does have the final authority over a domain. Those services actually provide IP address information to clients & recursive services
- **Resolver or Recursive DNS Service** - does not own DNS records, but connects to other DNS services as an intermediary to resolve any given record. A record can be immediately provided to a client if it is already cached.



## Caching to Speed up Requests and Reduce the Load on Resolvers

Caching is essential to keep the internet's DNS resolvers healthy. Caching is the process of saving information for a short period to reduce the load on up or downstream systems that are part of a request.

There are different locations where caching takes place. The two most obvious of those caching locations are **your browser & your operating system**. The closer the caching is to your browser, the better, as fewer processing steps are necessary. How long a DNS record is stored is specified by its Time-To-Live (TTL) attribute.

## Setting up Route 53 as Your DNS

You can use Route 53 for your existing domain, regardless of where you purchased it. You can also use Route 53 as the DNS service for a new domain.

### Route 53 as the DNS Service for Your Existing Domain

When transferring a domain to Route 53, you'll need to distinguish between two cases:

- your domain is currently in use and is receiving traffic
- your domain is not receiving any traffic

With both migration cases, your domain should stay available all the time. Nevertheless, option one lets you easily roll back in case of errors while the second one could leave it unavailable for up to several days.

### Migrating to Route 53 for a Domain That's in Use

When migrating a domain that's currently in use, meaning it gets traffic from users, we want to make sure that there are no service interruptions.

Let's distinguish between different scenarios:

- **Your DNS configuration is simple with only a few records** - you can manually reproduce it by creating the necessary entries in Route 53
- **Your DNS configuration is more complex, but you only want to reproduce the configuration without exploring more in-depth features of Route 53** - for most DNS providers you can get a zone file (which is always in an RFC-compliant format) that contains all the necessary information about your DNS setup. Simply import the zone file

in Route 53. The import can process up to a maximum of 1000 records.

- **Your DNS configuration is more complex and you want to use Route 53's advanced routing features** - have a look at the advanced features of Route 53 (which we'll explore in later paragraphs) and compare which features were covered by your previous DNS provider. You can then still export and import the zone file and then create or update records later.

Let's dive into the steps you have to go through to ensure a smooth transition.

1. **Get your current DNS configuration** - as described above, either manually or via the zone file.
2. **Create a hosted zone at Route 53** - The hosted zone needs to have the name of your domain and will later contain all your records. AWS will automatically create a name server record (NS) and a start of authority (SOA). You'll later update the registration in your previous DNS provider with the four values from the NS record to enable Route 53 to manage your domain.

## Create hosted zone Info

### Hosted zone configuration

A hosted zone is a container that holds information about how you want to route traffic for a domain, such as example.com, and its subdomains.

#### Domain name Info

This is the name of the domain that you want to route traffic for.

awsfundamentals.com

Valid characters: a-z, 0-9, ! " # \$ % & ' ( ) \* + , - / ; < = > ? @ [ \ ] ^ \_ ` { | } . ~

#### Description - optional Info

This value lets you distinguish hosted zones that have the same name.

The hosted zone is used for...

The description can have up to 256 characters. 0/256

#### Type Info

The type indicates whether you want to route traffic on the internet or in an Amazon VPC.

Public hosted zone

A public hosted zone determines how traffic is routed on the internet.

Private hosted zone

A private hosted zone determines how traffic is routed within an Amazon VPC.

### Tags Info

Apply tags to hosted zones to help organize and identify them.

No tags associated with the resource.

Add tag

You can add up to 50 more tags.

Cancel

Create hosted zone

3. **Creating your records** - if a zone file is available, you can import it directly. Else you can create records manually via the console interface or programmatically via the AWS SDK or Infrastructure as Code tools (see later chapter).
4. **Lower the TTL settings** - the Time-To-Live (TTL) of a record defines how long an entry can be cached at any point of the DNS resolving chain. As we want to transition between different DNS providers which could lead to issues, we want to have a very low TTL for our NS record so entries are not cached for a longer period. This should be done at your previous service provider **and** at Route 53.

5. **Waiting for the TTL to expire** - DNS resolvers will have your records cached if they were in use. We need to wait until all TTLs are expired so your previous name servers are not stored in any cache. This can take **up to two days**.
6. **Update the NS records to point to the Route 53 name servers** - go to the NS record in your hosted zone and copy the four name servers. Jump back to your previous DNS service provider and find the settings for domain management. Mostly, there will be some radio buttons to select between the domain management of the service provider and using your own custom name servers. Select the custom setting and enter the name servers you've copied from your Route 53 NS record.
7. **Monitor traffic for your application or website** - if the traffic stays consistent and as expected, all the steps worked out. If traffic slows or stops, switch back to your previous name servers and have a look back at the steps you've taken. Determine what went wrong.
8. **Change the TTL back to a higher value** - the transitioning phase is completed and you can allow resolvers to cache your NS records for a longer period.
9. (Optional) **Transfer domain registration to Amazon** - your domain is now managed by Route 53, but not registered at Amazon. This step completes the have all your domain management in a single place, but it is not necessary.

#### **Migrating to Route 53 for an Inactive Domain**

For migrating a domain that is not receiving any live traffic, migration does require fewer steps. The only steps you need to take are steps 1, 2, 3, and 6. You don't need to adapt TTL settings or wait for any cache expirations.

#### **Using Route 53 with a New Domain**

If you want to purchase a new domain and use Route 53, you can also buy a domain directly via Amazon and Route 53's console interface.

The screenshot shows the AWS Route 53 Dashboard. On the left, there's a sidebar with a navigation menu:

- Route 53**
- Dashboard** (selected)
- Hosted zones
- Health checks
- IP-based routing**
- CIDR collections
- Traffic flow**
- Traffic policies
- Policy records
- Domains**
- Registered domains
- Pending requests
- Resolver**
- VPCs
- Inbound endpoints
- Outbound endpoints
- Rules
- Query logging
- DNS Firewall**
- Rule groups
- Domain lists
- Application Recovery Controller**
- Getting started
- Readiness check
- Resource sets

The main content area is titled "Route 53 Dashboard" and includes the following sections:

- DNS management**: A hosted zone tells Route 53 how to respond to DNS queries for a domain such as example.com. Includes a "Create hosted zone" button.
- Traffic management**: A visual tool that lets you easily create policies for multiple endpoints in complex configurations. Includes a "Create policy" button.
- Availability monitoring**: Health checks monitor your applications and web resources, and direct DNS queries to healthy resources. Includes a "Create health check" button.
- Domain registration**: A domain is the name, such as example.com, that your users use to access your application. Includes a "Register domain" button.
- Readiness check**: Shows 0 readiness checks.
- Routing control**: Shows 0 control panels.

At the bottom, there's a "Register domain" section where you can enter a domain name (e.g., awsfundamentals.com) and check its availability.

When you register a domain, Amazon will automatically make Route 53 the DNS service for the domain. Route 53 will also create a hosted zone with the name of your domain, assign the name servers, and update the domain to use those name servers without you having to take any actions.

1: Domain Search

2: Contact Details

3: Verify & Purchase

### Choose a domain name

awsfundamentals	.com - \$12.00	<b>Check</b>
-----------------	----------------	--------------

Availability for 'awsfundamentals.com'

Domain Name	Status	Price / 1 Year	Action
awsfundamentals.com	<span style="color:red;">✗</span> Unavailable		

Related domain suggestions

Domain Name	Status	Price / 1 Year	Action
awsfundamentals.link	<span style="color:green;">✓</span> Available	\$5.00	<b>Add to cart</b>
awsfundamentals.net	<span style="color:green;">✓</span> Available	\$11.00	<b>Add to cart</b>
awsfundamentals.ninja	<span style="color:green;">✓</span> Available	\$18.00	<b>Add to cart</b>
awsfundamentals.org	<span style="color:green;">✓</span> Available	\$12.00	<b>Add to cart</b>
awsfundamentals.tv	<span style="color:green;">✓</span> Available	\$32.00	<b>Add to cart</b>
awsfundamentalsacademy.com	<span style="color:green;">✓</span> Available	\$12.00	<b>Add to cart</b>
awsfundamentalsllc.com	<span style="color:green;">✓</span> Available	\$12.00	<b>Add to cart</b>
awslearningfundamentals.com	<span style="color:green;">✓</span> Available	\$12.00	<b>Add to cart</b>
awstrainingfundamentals.com	<span style="color:green;">✓</span> Available	\$12.00	<b>Add to cart</b>
digitalawsfundamentals.com	<span style="color:green;">✓</span> Available	\$12.00	<b>Add to cart</b>
freeawsfundamentals.com	<span style="color:green;">✓</span> Available	\$12.00	<b>Add to cart</b>

**Cancel** **Continue**

## Understanding Hosted Zones

A hosted zone acts as a container for a number of records that specify how traffic should be routed for the root domain and its subdomains. Hosted zones come in two different flavors: **public** and **private**.

- Public-hosted zones do route traffic on the internet and can be resolved from anywhere in the world.
- Private-hosted zones are for DNS records within your private VPC and are not exposed to the internet. The domains can only be resolved from inside your VPC.

## Routing Traffic to Your Resources

You can use Route 53 to route traffic to various AWS services, including:

- API Gateway APIs

- CloudFront distributions
- EC2 instances
- Elastic Load Balancers
- App Runner Services
- A statically hosted website at S3

There are different types of routing that you can use for your domain names. Let's dive into those policies in the paragraph.

### **Choosing the Right Routing Policy**

The type determines how Amazon Route 53 responds to queries for those domain names. The selection of the best-fitting record types highly depends on your requirements.

#### **Simple Routing to Forward Requests to One or Multiple Resources**

The standard DNS record without any strings attached. Typically used for single resources that are performing given functions for your domain.

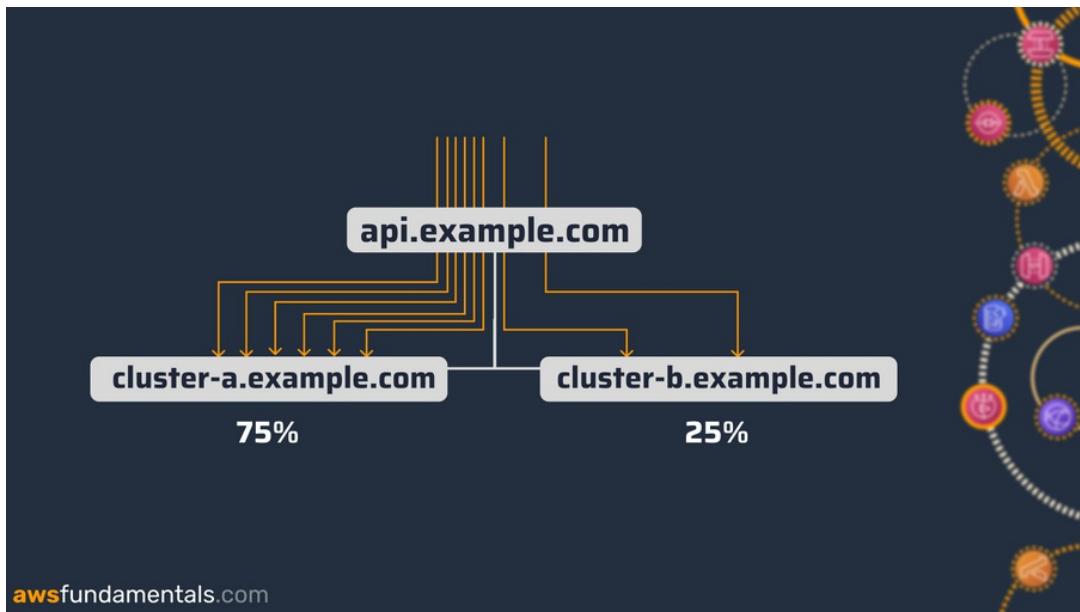
You can't create multiple records with the same name for this record type. What you can do: specify multiple values for your single record.

Route53 will return all values for an incoming query to the client. The client can then choose one of those by himself.

#### **Weighted Routing to Allocate Traffic Proportionally between Multiple Targets**

As the name already suggests, it allows you to define multiple records for the same domain name. You choose how much traffic is routed to each record by giving it a percentage.

Prominent use cases are load balancing and testing new features or releases.



Weighted routing not only enables you to quickly scale your application but also build blue/green deployments or do traffic shifting that is fully in your control.

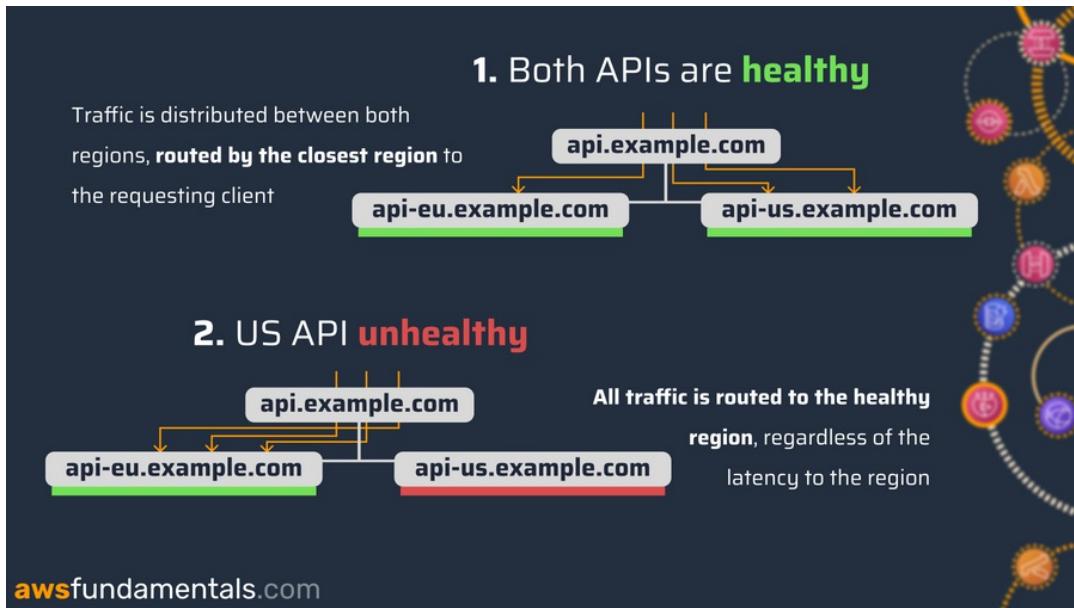
#### **Failover Routing to Send Traffic to a Fallback Resource If the Main Target Is Unhealthy**

What if you have a multi-region setup with latency-based routing, but the closest region for a customer is not available for any reason? Surely, we don't want to route requests to this region.

That's where health checks come in. You can define health checks in Route 53 that monitor AWS-native or even external endpoints for their availability.

Those checks are configurable:

- the time between two independent checks
- from which locations they should origin



**The exciting part:** you can attach those health checks for example to your Latency-based records. If a health check for one of your locations becomes unhealthy, the corresponding target won't be propagated anymore for this DNS record.

There you have it: a quite easy active-active multi-region failover setup.

#### **Latency-Based Routing to Forwarding Requests to a Resource That Is the Closest to the Customer**

In a multi-region setup, in most cases, you want to route requests to the closest regions, as it will on average serve the fastest responses.

Via Latency-based routing, you create multiple records for a given domain name. Each record is created for a specific region and if your DNS record is queried, Route 53 will resolve it by choosing the one with the lowest latency.

Latencies between hosts can change. If a client is close to several regions, routing results can vary over time.

#### **Geolocation Routing for Redirecting Traffic Based on the Location of the Customer**

Geo-Location records allow routing based on the origin of your clients. This enables you to easily localize content or implement geo-restrictions to comply with regulations.

The granularity of locations is either by continent, country, or even US state.

## **Building Health Checks and Setting up Failovers to Avoid Downtimes on Region Issues**

As seen, a significant feature of Route 53 is its capability of providing health checks and routing traffic based on these conditions. This should be considered for any application architecture to provide resilient services that are able to operate even if drastic incidents in one or multiple AWS availability zones or regions occur.

### **What Costs to Expect with Route 53**

Route 53 has very transparent pricing.

- **On-demand pricing** - you only pay for what you use, as with most other fully-managed services on AWS.
- no minimum usage commitments or upfront costs.

You'll pay for:

- **Each managed zone** - currently, this is \$0.5 per month.
- **Per DNS query** - currently \$0.4 per million, with slightly higher costs for queries of Geolocation, Latency, and Geoproximity routing records.

Additionally, you'll pay for health checks, starting from \$0.50 for a default HTTP health check for AWS-hosted endpoints.

### **The Use Cases For Route 53**

The use cases of Route 53 are manifold due to its large feature set. As it's not a simple domain name system, but an advanced tool that enables you to build resilient and highly available architectures.

#### **Use Case 1: A Multi-Region Setup for Serverless Architectures**

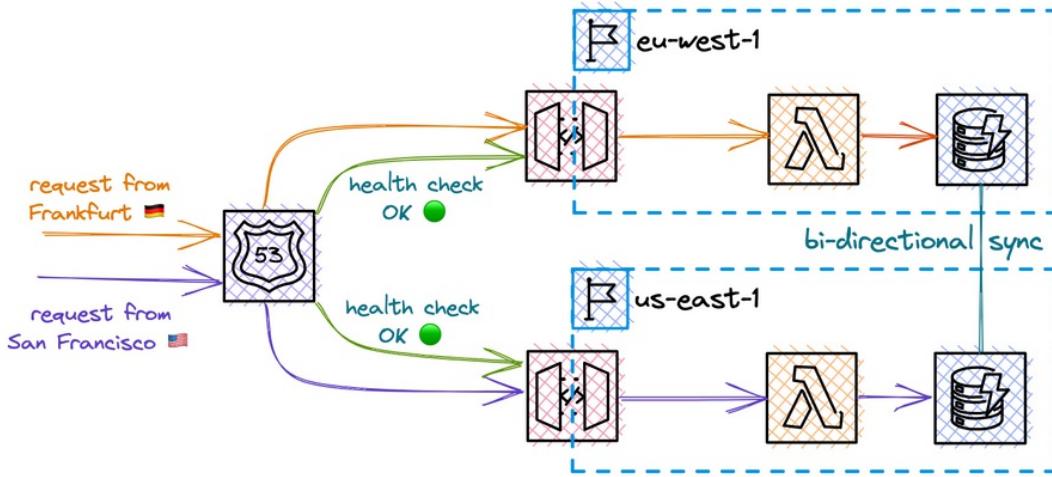
If you're focusing on serverless architectures, you can deploy your eco-system around the world without drastically increasing your costs. This is due to the fact that unused resource capacities are not contributing to your bill. So deploying your Lambda-powered application in more regions won't have much or any effect on your charges at the end of the month.

Building such a multi-region setup with Route 53 is straightforward and only involves a few

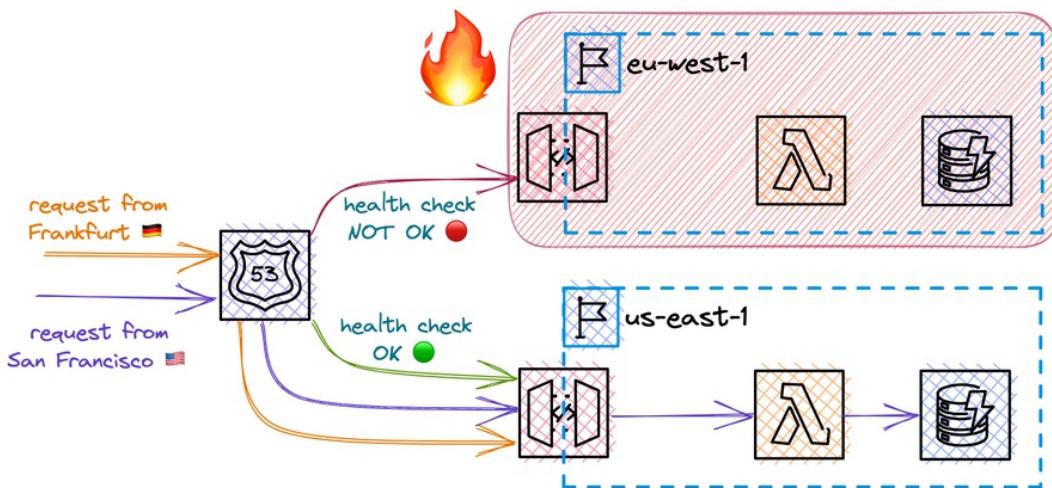
steps:

1. Configuring health checks for every region.
2. Setting up latency-based DNS records to each of your region's regional endpoints.
3. Attaching the health checks to your DNS records.

If all regions are healthy, requests will be routed to the region with the lowest latency to the origin. Even if there's little or no traffic at all in a single region, it won't hurt in any way.



If a region has an outage, or you break the application by accident via a faulty deployment, there won't be any significant outage as Route 53 will quickly failover and won't return the unresponsive region in any DNS query.



By that, requests for origins with long distances may become slower, but at least the availability

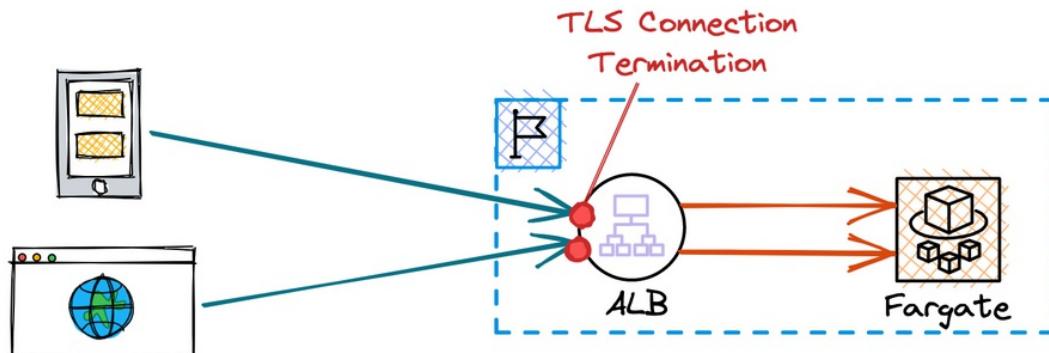
of the application is only affected for a very short time period.

### Use Case 2: Building your own Canary Deployment System

Canary deployments roll out a new version of an application to a small, sometimes dedicated part of its user first to mitigate the risks of deploying an erroneous version to a huge fraction of users. In the first stage of this rollout, it can be checked how users interact with the features and if everything goes according to plan. This means that there are no spikes in error logs or other unexpected behavior that may cause a rollback to the initial version.

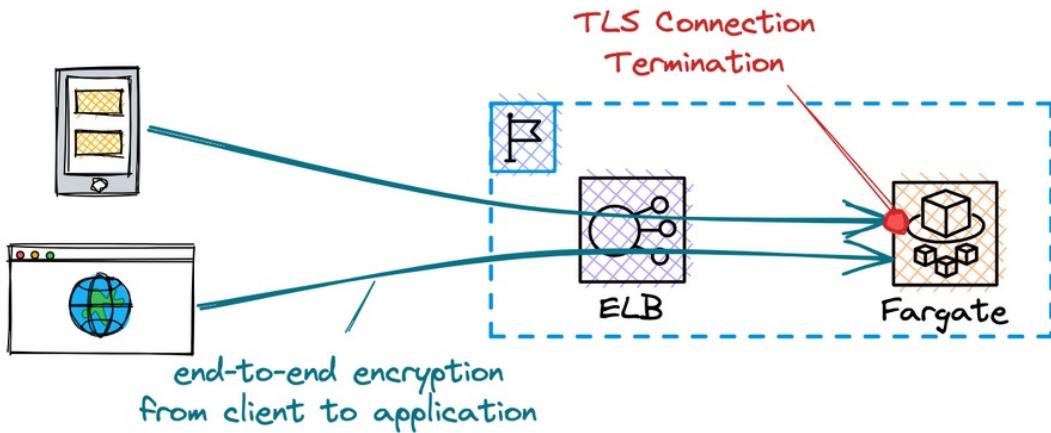
Let's do a small recap of some networking fundamentals before we continue with our scenario.

Services like the Application Load Balancer (ALB) offer request routing based on request details out of the box. It can be used to route traffic based on headers or cookies, so you can explicitly send specific users to a dedicated deployment of your application.



The ALB will terminate the TLS connection. The downstream service, such as an ECS task, communicates with the ALB over a separate HTTP or HTTPS connection, depending on the configuration of the ALB.

But some architectures do not support such simple traffic routing. This includes applications that do require secure connections to the application container. For example, if you're running an ECS cluster and the TLS connection will be terminated within the container itself (e.g. with an NGINX container), not by the load balancer. This requires the usage of an Elastic Load Balancer (ELB), instead of the ALB. The ELB works on the transport layer and does not terminate the TLS connection, but forwards the request as is. This increases the security even further, as the connection has real end-to-end encryption from the client to the application itself.

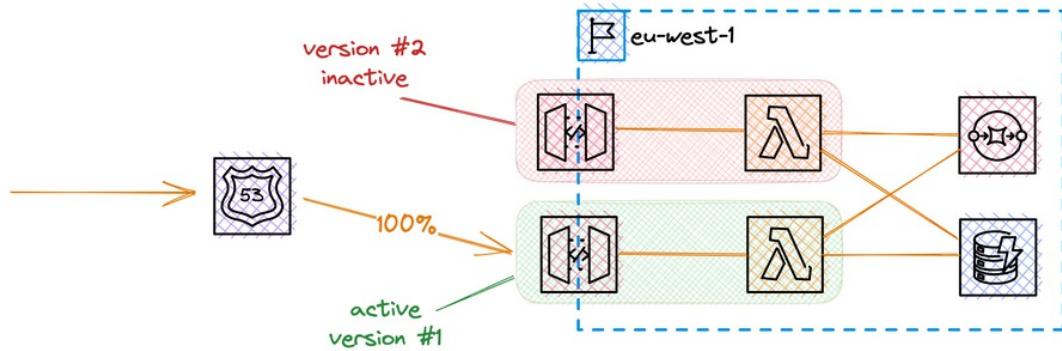


The obvious downside: all services in between, including the ELB, can't look in the request details for HTTPS requests, so it's not possible to take any decisions based on the request for the routing. What we can do is set weights on target groups to route the percentage of traffic to every group.

Why does this matter? If we're looking at Serverless architectures, we're not working with load balancers as there are no long-running containers. We're working with AWS API Gateway. The API Gateway also offers a feature to route requests to different Lambda function versions. When a client makes a request to the API, we can specify the version in the request, and API Gateway will route the request to the appropriate Lambda function version.

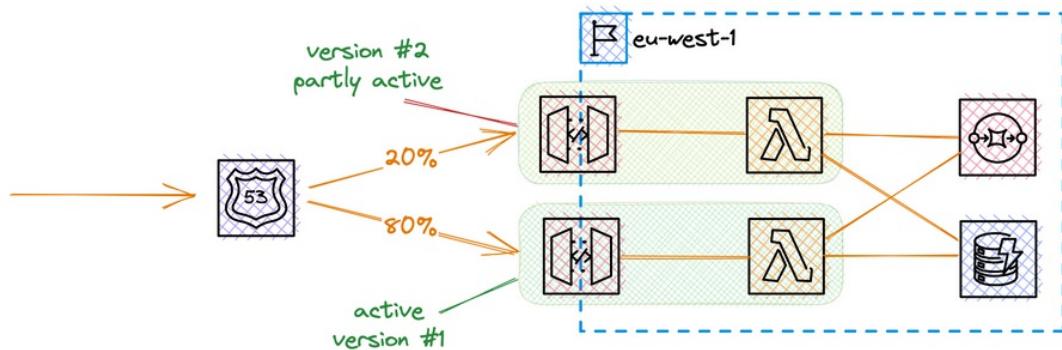
Often times our complete infrastructure deployed is coupled with our Lambda function's code, so setting up routing to different versions of functions is not trivial. It's also not easily possible to roll back in cases of errors if the infrastructure is not strictly separated from our function's code.

But think back to our main advantage of Serverless architectures: we can easily replicate our whole infrastructure without having to fear costs. That's why we can run multiple stacks (so multiple replications of our application, including most or all of its infrastructure) and delegate the routing decisions to Route 53 by using records for weighted routing.

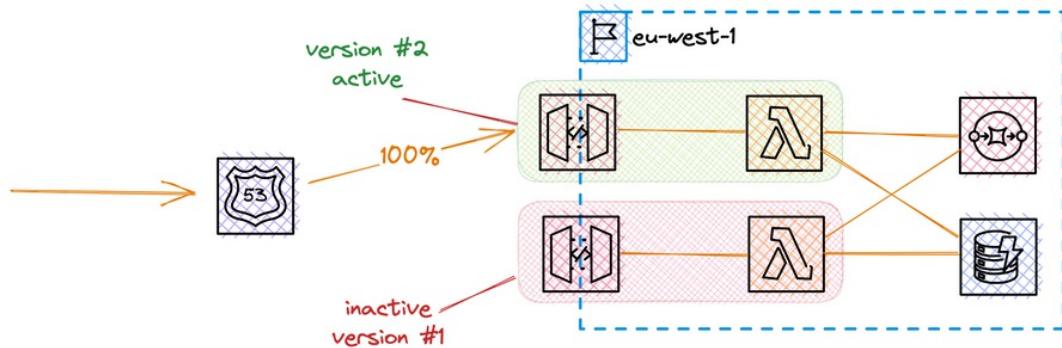


We'll deploy multiple stacks of our application infrastructure. In the example, this contains an AWS API Gateway deployment and an AWS Lambda function. We have other global infrastructure that is independent of our application stack: SQS and DynamoDB.

Each of our regional API gateways will receive its weighted DNS record. The inactive stack with weight set to zero and the active one set to 100.



When we want to roll out a new code deployment, we'll deploy the inactive stack first. After it successfully completed, we can adapt the weights in our DNS record. For example by setting the stack of the new version to receive 20% of the traffic while the old one keeps 80%.



After we've run this configuration for a while and made sure there are no error spikes or other

suspicious behavior in our application, we can switch the rest of the traffic to the new version.

## Tips & Tricks for the Real World

Route 53 is great and simple to use. Keep a few things in mind:

- **Directly registering domains** - Route 53 allows you to register domains directly through AWS, making it easy to manage your domain name and DNS settings in one place.
- **Delegating any existing domain to Route 53** - you delegate the permissions to administrate your domain to Route 53 from any DNS provider. This includes the famous platforms GoDaddy and Namecheap.
- **Health checks are powerful to ensure high availability** - By attaching health checks to your records you can automatically failover when a resource becomes unavailable. This ensures that users can still access your service.
- **Use Alias records to attach a domain to AWS resources** - with Alias records, you can map domain names to load balancers, CloudFront distributions, API Gateway endpoints, or even another Route 53 record in the same hosted zone. When a query reaches Route 53, it will resolve the address for your resource, so it will always receive the latest IP even if it has changed beforehand.
- **Use multi-answer routing** - With Route 53, you can return multiple results for the same domain name. The client can then choose which IP address it wants to use.

## Final Words

Route 53 is a DNS service with a huge feature set without any strings attached. As with other managed services, you don't end up with additional operational burdens.

Its pricing is fair and transparent and the free tier will go a long way in the first place.



Amazon **VPC**

# Isolating and Securing Your Instances and Resources with VPC

## Introduction

Amazon Virtual Private Cloud (VPC) enables you to define and launch AWS resources in a logically isolated virtual network. It can imitate your local data center, but with all the benefits of the cloud's scalable infrastructure.

### Why Would or Do You Need to Use a VPC?

Running resources in the cloud means that they can be open to the world. That's not a downside, but mostly what you actually want. You want customers to be able to access what you've built.

But this comes with the risk of attacks from the internet.

VPC helps you to **lock down your resources and protect them** in different ways from the outside via multiple methods. Additionally, not all of your resources have to or even should be publicly available or have access to the internet at all. This for example includes databases or caches. VPC also enables you to protect those resources by restricting their network access and availability.

### Getting into the Networking Fundamentals to Understand How VPCs Work

Before we jump into VPC's capabilities, it is good to have a quick overview of networking fundamentals that will be used in the following chapters.

#### Subnets to Slice Your Network into Distinct, Isolated Parts

Within Amazon VPC, subnets allow you to segment a VPC even further. This allows you to split resources into different segments. For example grouping into resources that need or don't need internet access.

#### Access Control Lists and Security Groups to Secure Your Network and Its Resources

VPCs use security measures such as security groups and network access control lists (ACLs) to control inbound and outbound traffic and protect resources from unauthorized access.

## Routing Tables to Control the Flow of Traffic between Subnets

VPCs use routing tables which contain a set of rules, called routes, that determine where network traffic is directed. Each route has a destination IP range and a target, which can be a subnet, an Internet Gateway, or a Virtual Private Gateway.

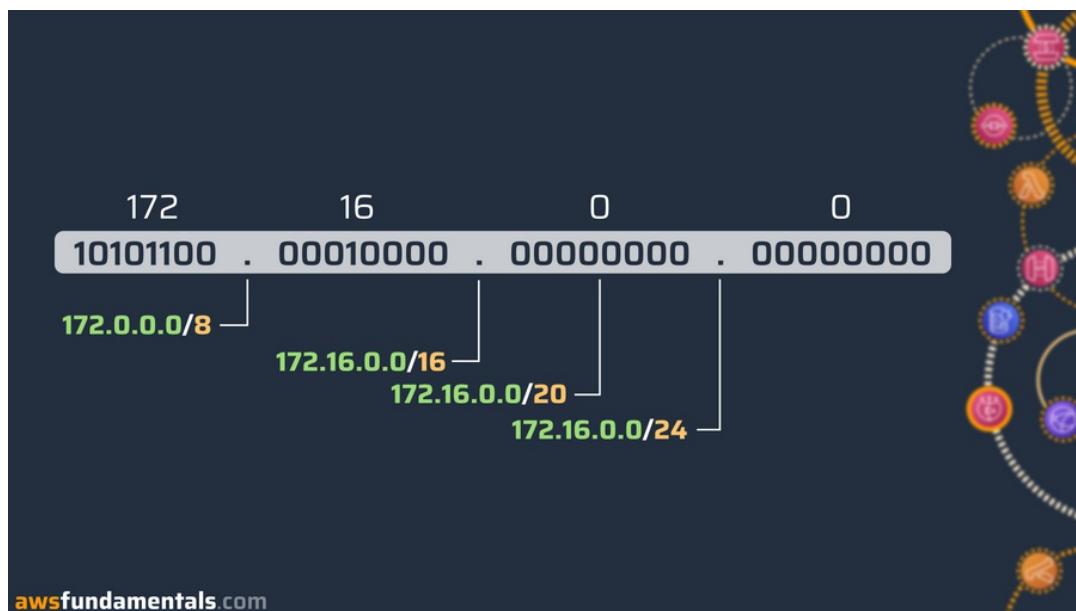
### Classless Inter-Domain Routing Blocks (CIDR)

Your VPC needs a range of IPv4 addresses that attached network interfaces can use. They have been defined as Classless Inter-Domain Routing (CIDR) blocks.

They are made up of two number sets:

- **The prefix** - the binary representation of the address.
- **The suffix** - the total number of bits in the entire address.

The allowed block size for a VPC is between 65,536 (netmask /16) and 16 IP (netmask /24) addresses.

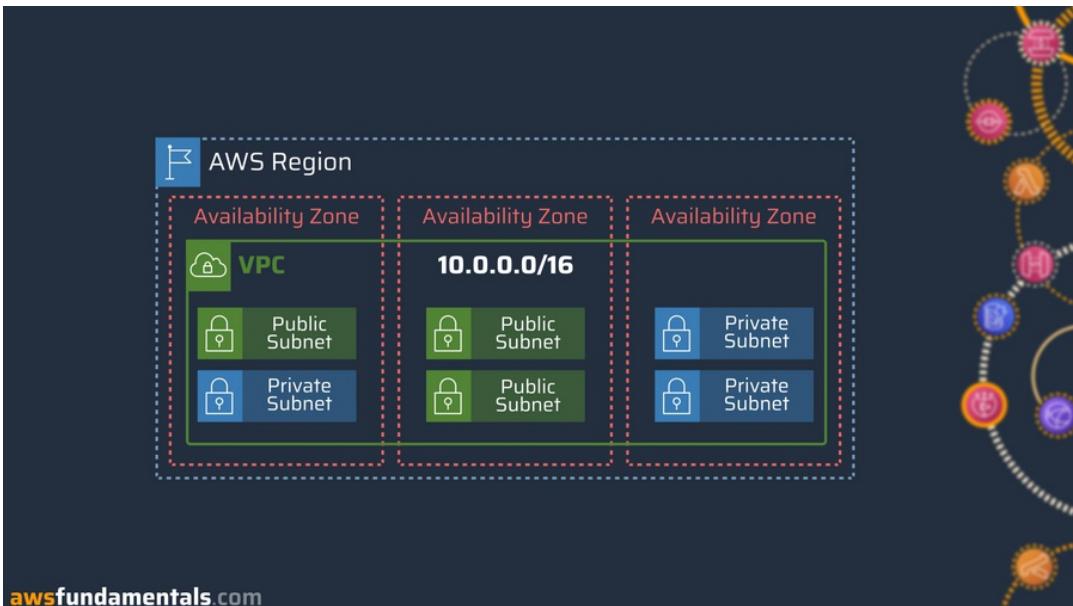


## Virtual Private Clouds

Each VPC is created for a region and always spans across all availability zones. Each availability zone can contain subnets that are another breakdown of your VPC.

Subnets can't span multiple availability zones but only a single one. For redundancy and

availability reasons it's therefore recommended to have at least two subnets for a single region so you can have resources in at least two availability zones.



#### Each Region Comes with a Default VPC and a Default Set of Subnets

Each AWS account created after the end of 2013 comes with a default VPC per region. Each of those default VPCs also has a public subnet in each availability zone, an internet gateway, and settings to enable DNS resolution.

#### Slicing Your Network into Isolated Parts via Subnets

Subnets allow you to further separate your network into smaller parts. The most important segregation is between public and private subnets:

- **Public** - for resources that need to be accessed from the internet.
- **Private** - for resources that only need to be accessed internally and therefore do not need or get a public IP address.

Resources in each subnet can be protected with multiple layers of security, e.g. with their own Security Groups or Network Access Control Lists.

Some AWS services require you to launch instances into a private subnet, e.g. ElastiCache.

## Adding Layers of Security with Network Access Control Lists and Security Groups

Depending on the type of resource and your requirements, it's necessary to further drill down access to and between your resources.

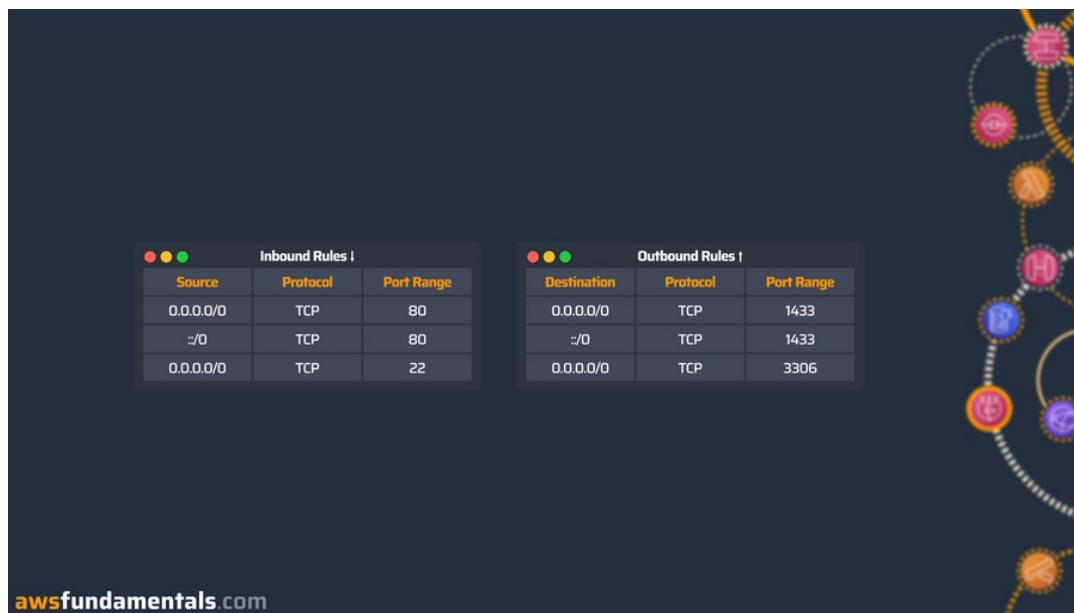
Common requirements are:

- a resource should be or should not be able to access the internet
- a resource should be able to be accessed by other resources in another subnet
- a resource should only be able to access a dedicated IP range

With Security Groups (**SGs**) and Network Access Control Lists (**NACLs**)

### Security Groups to Restrict Access to Individual Resources

Security Groups define and allow rules for your traffic - inbound or outbound. They enable traffic filtering based on protocols and port numbers.



Security groups operate on the instance level and are stateful. Stateful means that return traffic doesn't need to be allowed explicitly.

## Network Access Control Lists to Restrict Access for All Resources within a Subnet

Network Access Control Lists act as a firewall on the network level. They can hold one or multiple allow and/or deny rules which are ordered via priority numbers.



Evaluation starts with the lowest rule number and the first match will be executed.

Each subnet must be assigned to a network ACL and return traffic must be explicitly allowed as NACLs are stateless.

## Using Gateways for Outbound Internet Access in Your Network

We've talked about how to protect resources and how to enable or restrict traffic on different network levels. But how do we provide internet access for resources in general? How are public subnets connected to the internet and how do we allow resources in private subnets to connect to the internet?

### Internet Gateways to Enable Internet Access for Resources in Public Subnets

An Internet Gateway (IGW) is an AWS-managed highly-available VPC component that allows resources that reside in public subnets to communicate with the internet.

Private subnets do not have a routing connection to the IGW, therefore they do not have any connection to the internet by default.

## NAT Instances for Allowing Private Resources to Communicate with the Internet

How do you get access to the internet for resources in private networks? That's where either NAT Instances or NAT Gateways come into play.

NAT Instances can be created in a public subnet that has internet access. Afterward, you have to allow access from your private instances to the NAT Instance to grant them internet access. This is done by adding a route in the routing table of your private subnet to the NAT instance.

The granted access is only available in one direction: from your instance to the internet. This means your resources will not become accessible through the web.

Besides residing in a public subnet, NAT instances must have an Elastic IP address. As the NAT gateway is an instance, it's also a good practice to run them in at least two availability zones to protect against outages.

## NAT Gateways - The Managed NAT Instance

NAT Gateways are praised as the successor to the NAT instances as they are easier to set up. Also, they are **managed by AWS and scale by default**, so you don't have to take care of anything. NAT Gateways do not need to be associated with a security group. They also do not require to have an Elastic IP address.

As everything comes with some bitter aftertaste: NAT Gateways can become **very expensive** as the traffic flowing through them is charged at a high rate. Especially large accounts with heavy traffic end up with a high percentage of their bill alone due to NAT Gateway.

## Routing Requests in Your Network via Route Tables

Traffic inside your VPC needs directions. That's why you can create route tables, which are sets of rules that you can associate with a subnet (custom route tables).

Each route table entry needs a destination and target which defines how traffic is routed.

- **Destination** - a range of IP addresses where traffic should go defined as a CIDR block.  
e.g. an external corporate defined as 172.16.0.0/12.
- **Target** - the gateway, network interface, or connection through which to send the destination traffic, e.g. an internet gateway.

## **The Default Route Table**

Each of your VPCs comes with a default route table. It controls the traffic for subnets that do not have a custom route table attached.

## **Monitoring the Traffic in Your Network**

Maybe some requests are not reaching your instance in your VPC. This requires you to get insights into how traffic is flowing within your subnets to resolve the issue.

### **Capturing VPC Network Traffic via Flow Logs**

You can monitor your VPC via Flow Logs. Those logs capture details about how IP traffic is going to and from network interfaces in your VPC. The logs can be shipped to either CloudWatch, S3, or Kinesis Data Firehose.

```
[...] eni-5123b7ac012345678 219.42.22.48 172.16.0.101 [...] ACCEPT OK  
[...] eni-5123b7ac012345678 172.31.16.139 219.42.22.48 [...] REJECT OK
```

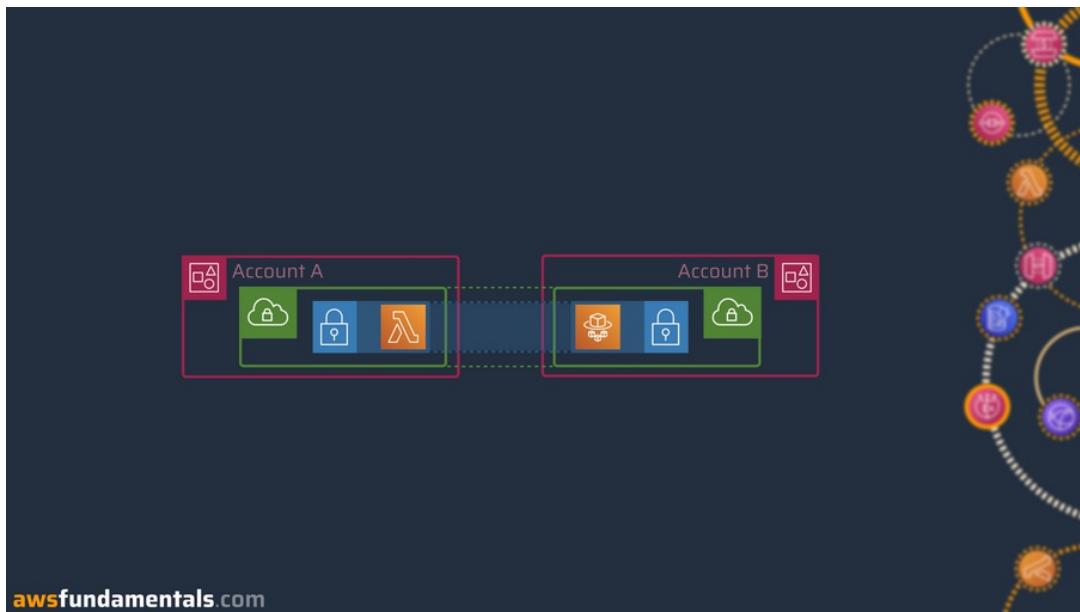
Looking at the example flow logs above, an incoming request was accepted, but the response was rejected. This could happen even if you've defined allow rules for inbound traffic in your security group and network ACLs. As security groups are stateful, responses are allowed. ACLs are not stateful, so a missing outbound allow rule does result in a rejection.

## **Bringing Different VPCs Together with Peering and Sharing**

A well-structured application is often split into separate AWS accounts for security and/or organizational reasons. VPC comes with multiple methods of either connecting VPCs within a single AWS account or sharing a single VPC over multiple AWS accounts to not interfere with the multi-account concept.

### **VPC Peering to Connect with VPC in Other Regions or AWS Accounts**

Peering connections allow you to route traffic between two VPCs as if they were in the same VPC. It also allows you to not only connect to VPCs in other regions but also in other AWS accounts.

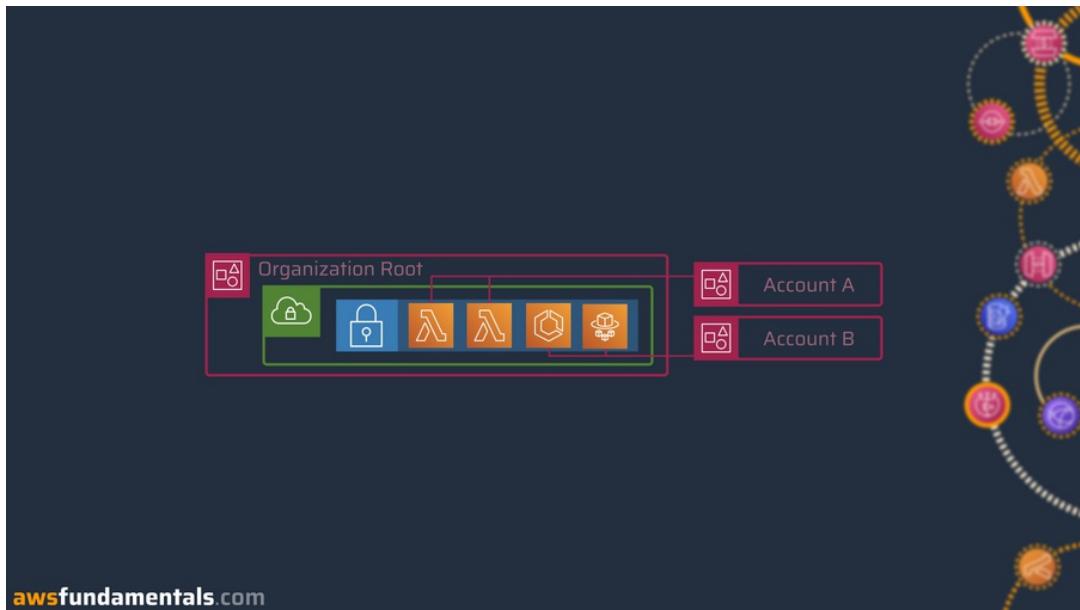


Important to remember: CIDR blocks for your connected VPCs can't overlap.

#### VPC Sharing for Centrally Managed VPCs That Can Be Used by Multiple AWS Accounts

Share a VPC with other accounts that are part of the same AWS Organization, so that multiple accounts can launch resources into the centrally-managed subnets but still be in full control of their resources. Participating accounts can't modify resources in shared subnets that they do not own.

This allows for a fine-grained separation of accounts for billing and access control, but still having components with high interconnectivity.

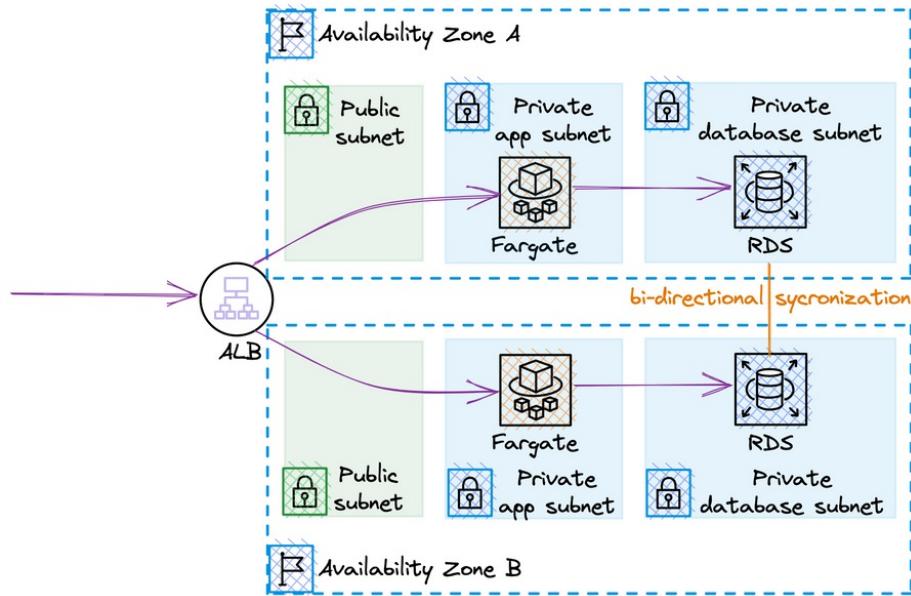


## The Various Use Cases for VPCs

As VPC enables you to fully control your network and resources, the use cases can't be counted.

### Use Case 1: Building and securing Multi-Tier Architectures

With VPC, you can build multi-tier architectures with public-facing resources like load balancers and DNS servers, and private application servers and databases. The application servers will still have a route to the internet, while they are protected from the internet as they are only available via the load balancer of the public subnets.



The databases on the other hand reside in a strictly private subnet with no outgoing internet access. They will be only available via the applications running in the other private subnet.

### Use Case 2: Complying with Regulatory Requirements

VPC enables you to control the physical location of data, as well as access to data and network traffic to it. This is not only necessary from a security point but also often from a compliance perspective. Many countries do have laws that govern the storage, use, and collection of data, for example, the General Data Protection Regulation (GDPR) in the European Union, or the California Consumer Privacy Act (CCPA) in California.

The regulations not only require you to encrypt data at rest (while data is stored) and transit (while data flows between destinations) but also enforce strict access control mechanisms. Those can be fulfilled by conscientiously making use of subnet, security groups, and network access control lists.

### Use Case 3: Running Hybrid Architectures with on-Premise and the Cloud

Many companies host their data and run their applications on-premise in their own data centers. Not only because they didn't finish the migration to the cloud, but because they want to.

For these companies, it's often very useful to run a hybrid architecture between on-premise and cloud ecosystems. And that's perfectly doable with Amazon VPC, as on-premise

infrastructure can be connected with AWS networks.

While sensitive workloads that need data to be stored locally can reside on-premise, the company can still leverage the scalability and flexibility of the AWS cloud for all other workloads.

### Tips and Tricks for the Real World

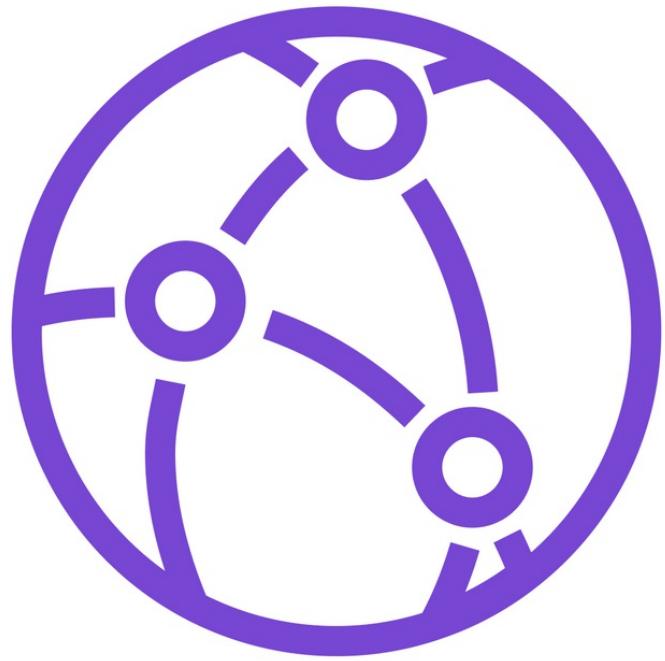
Amazon VPC is maybe one of the least favorite services for beginners, as it requires rather deep knowledge of network fundamentals. Let's go through some important reminders when working with VPC.

- **VPC integration is often optional, but advisable** - services like AWS Lambda do not require you to attach a VPC to your function, as it's secured by AWS IAM. Nevertheless, IAM doesn't offer all necessities to comply with strict compliance issues. For example, it's not possible to restrict outgoing traffic from your functions to the internet. This is only possible when using VPCs.
- **Use tags** - tagging your resources properly, including your subnets, make it easier to organize and manage them. When you shift to using Infrastructure-as-Code tools, this will become even more important.
- **Enable flow logs** - debugging connection issues within your VPC is difficult. Without flow logs, it's mostly a guessing and searching game.
- **Security groups are stateful, network ACLs are stateless** - it's one of the common mistakes that are made that lead to connection issues. Security groups are stateful, which means you don't need to explicitly allow return traffic from your instances. Network access control lists on the other side, are stateless. For letting return traffic flow, there needs to be an explicit rule.
- **It's possible to use static IPs** - public-facing IP addresses for EC2 instances can change when stopping and restarting the instance. With Elastic IP Addresses, you can configure a fixed IP address for your instances which makes it easier to connect to them.
- **Use placement groups for improved network performance** - with placement groups, you put instances close to each other in the same availability zone. This will improve the network performance between those instances.

## **Final Words**

VPC enables you to slice your network based on your security requirements to easily isolate components on different levels. VPCs also support network security features such as security groups and access control lists (ACLs), which allow you to control inbound and outbound traffic to resources in the VPC.

VPCs can be used in conjunction with a variety of AWS services, such as S3, RDS, or DynamoDB, to create a complete and secure infrastructure for hosting applications and data in the cloud that do not miss anything an on-premise data center can offer.

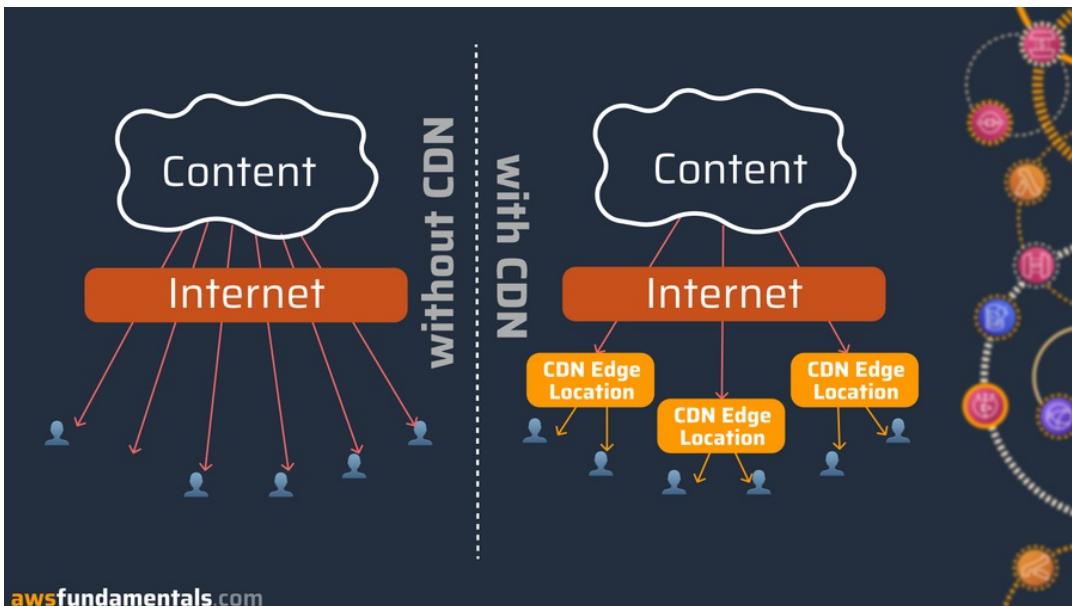


Amazon **CloudFront**

# Using CloudFront to Distribute Your Content around the Globe

## Introduction

CloudFront is a Content Delivery Network (CDN) that consists of a set of globally distributed caching servers that can store content returned by your origin servers, enabling fast, low-latency requests to your content around the globe.



Instead of needing to manage multiple regions and take care of syncing content, CloudFront brings your content close to almost any location in the world. This will drastically speed up your content's delivery.

## Using CloudFront Enables You to Deliver Content in an Efficient, Reliable, and Fast Way

Using CloudFront brings a lot of benefits. The key aspects are:

- **Faster performance & reliability** - CloudFront supports several network-layer optimizations such as TCP fast-open, request collapsing, keep-alive connections & more. It also supports multiple origins, so you can easily increase redundancy for your architecture.
- **Security** - CloudFront supports the latest version of Transport Layer Security (TLSv1.3) to encrypt and secure traffic between clients & CloudFront. Additionally, you rely on geo-restrictions to prevent users from accessing your content from specific locations.

- **Customizable edge behaviors** - you're in full control of how CloudFront caches requests, accesses your origin servers & which metadata is forwarded. With edge functions, you can intercept and adapt requests and further customize behavior.
- **Cost-effective** - CloudFront pricing is pay-per-use without any minimum fee. Traffic between other AWS cloud services and CloudFront is free & AWS offers a generous free tier for outgoing traffic from CloudFront each month

### **A Globally Distributed Network of Edge Locations**

A distribution is an actual instantiation of CloudFront. It contains the configuration of your origin servers - the actual endpoints that contain your content - as well as how CloudFront should cache those contents and based on which request details.

AWS itself will take care of caching the content in the closest edge locations (CloudFront has more than 225 edge locations in 47 countries) to the request's origin.

### **Choosing Where to Retrieve Your Content by Defining Origins**

An origin for your content can be anything that is able to serve content via HTTP. A common choice, which is also very easy to set up, is S3.

In this example, there's no need for public access to your S3 bucket anymore, as your content will be distributed via CloudFront.

CloudFront will retrieve content dynamically from your bucket and store it within the edge locations. Consecutive requests for the same resource will return the cached contents directly via CloudFront, without invoking a request to your origin anymore.

### **Controlling the Cache Behaviors within Your Edge Locations**

Caching will bring a huge performance benefit for web applications, but it can be also a trap when not properly configured as you can easily distribute outdated content.

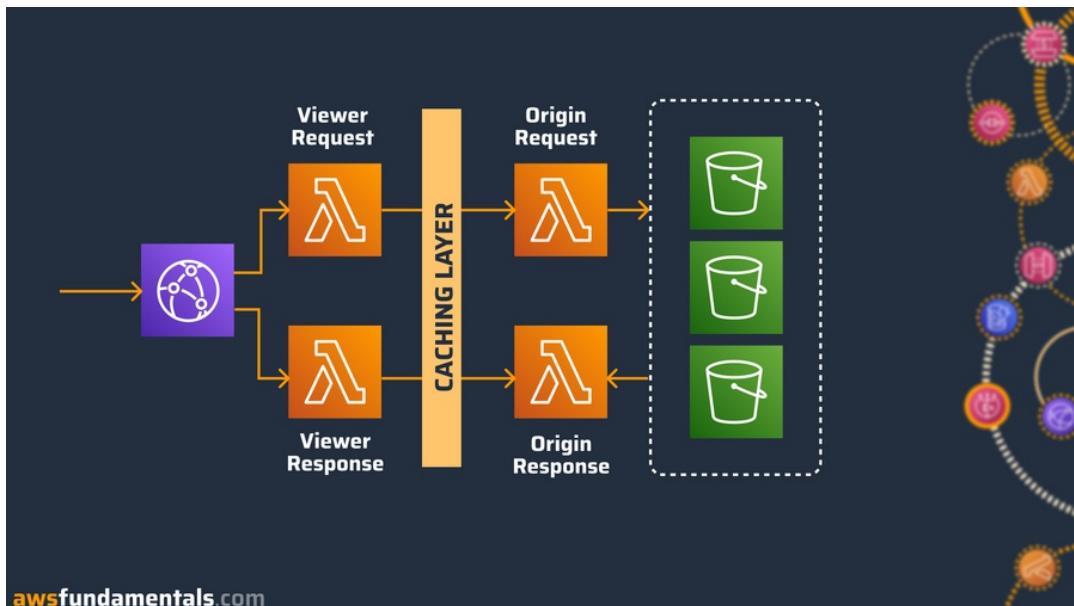
You're able to choose between two caching policy types:

- **AWS-managed** - CloudFront will cache your requests by the domain name of your distribution and the requested path
- **Custom** - define which parts of the request should be used as a cache key, e.g. specific headers or cookies

The custom policy allows for explicit configuration of your caching strategy so you're in full control of which content is cached, for how long, and by which cache key. This also allows you to send requests that explicitly ignore existing cached content.

### Running Code on the Edge to Implement Custom Business Logic

Besides having lower latencies and higher transfer speeds, CloudFront comes with the major benefit of enabling you to run custom code directly on CloudFront's edge nodes.



How does this work and at which step of the request can you run your own code? Looking at the image above, there are four different options, which can be used all in parallel:

- **Viewer Request** - invoked at the start of **every** request, even if the content is already cached for the current cache key.
- **Origin Request** - before CloudFront requests your origin to populate its cache with the target content.
- **Origin Response** - after CloudFront has received a response from your origin server.
- **Viewer Response** - invoked at the end of **every** request, equal to the viewer request.

This allows for almost limitless features as you are in total control of the requests and responses.

## **The Different Types of Functions You Can Run on the Edge**

There are two different types of functions you can run in response to the specific CloudFront events: Lambda@Edge and CloudFront functions. Both differ in their set feature sets and pricing.

### **Lambda@Edge - An Extended Feature Set for Non-Basic Functionality**

Lambda@Edge comes with the full feature set, including network and file system access. As your default Lambda function, each Lambda@Function comes with AWS-SDK so you can easily access other services like DynamoDB, SNS, or even other Lambda functions.

### **CloudFront Functions - For Simple and Cheap Functions**

CloudFront functions are the lightweight alternative to Lambda@Edge with fewer capabilities, but with better latency and cheaper pricing.

The main difference is that CloudFront functions do not have a network, file system, or request body access, so you're limited to only working with local variables and the request context. CloudFront functions are also not allowed to run longer than 1 ms and you can't set up origin request or response functions.

This means: CloudFront functions are great for simple tasks like request modifications or routing between origins. Complex tasks are still a task for Lambda@Edge.

## **Securing and Restricting Access to Your CloudFront Distributions and Origins**

There are several ways to secure and restrict access to your CloudFront distributions and origins.

### **Integration with AWS IAM**

When using native services as origins for CloudFront like S3, protecting your origins is solely possible via AWS IAM.

### **Origin Identities**

For our common example of retrieving content from S3, you're able to create an **Origin Access Identity (OAI)** for CloudFront and grant this identity access to your bucket via a bucket policy.

```
{
  "Id": "BucketPolicy",
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "CloudFrontAccess",
      "Action": [
        "s3:GetObject",
        "s3>List"
      ],
      "Effect": "Allow",
      "Resource": [
        "arn:aws:s3:::<bucket-name>",
        "arn:aws:s3:::<bucket-name>/*"
      ],
      "Principal": {
        "AWS": "<origin-identity-arn>"
      }
    }
  ]
}
```

## Origin Access Controls

In August 2022 AWS introduced **Origin Access Control (OAC)** as a successor to OAIs. OACs work with short terms credentials and resource-based policies which bring more security. It also allows for other features like support for Server-Side-Encryption via keys from KMS.

When creating your CloudFront distribution and you choose OAC instead of OAI, CloudFront will provide you with the resource-based policy for S3.

### **Protect against Common Threads and Implement Custom Rules for Every Security Requirement with AWS Web Application Firewall**

CloudFront supports AWS Web Application Firewall (WAF), which lets you monitor the HTTP/s requests that are forwarded to CloudFront and let you control access to your content. You can attach a single WAF to one or several of your CloudFront distributions.

It also protects your distributions from common web exploits that could affect the availability, performance, or security of your applications like as SQL injection, cross-site scripting (XSS)

attacks, or Denial of Service (DoS) attacks.

You're also able to create custom security rules in AWS WAF to meet your specific security needs.

Lastly, AWS WAF integrates with CloudWatch, which allows you to monitor the effectiveness of your security rules and receive alerts when suspicious activity is detected.

#### **Limiting Access to Your Content by the Requester's Location**

CloudFront automatically detects the origin of the client requests, which you can further use to create approval or blocking lists.

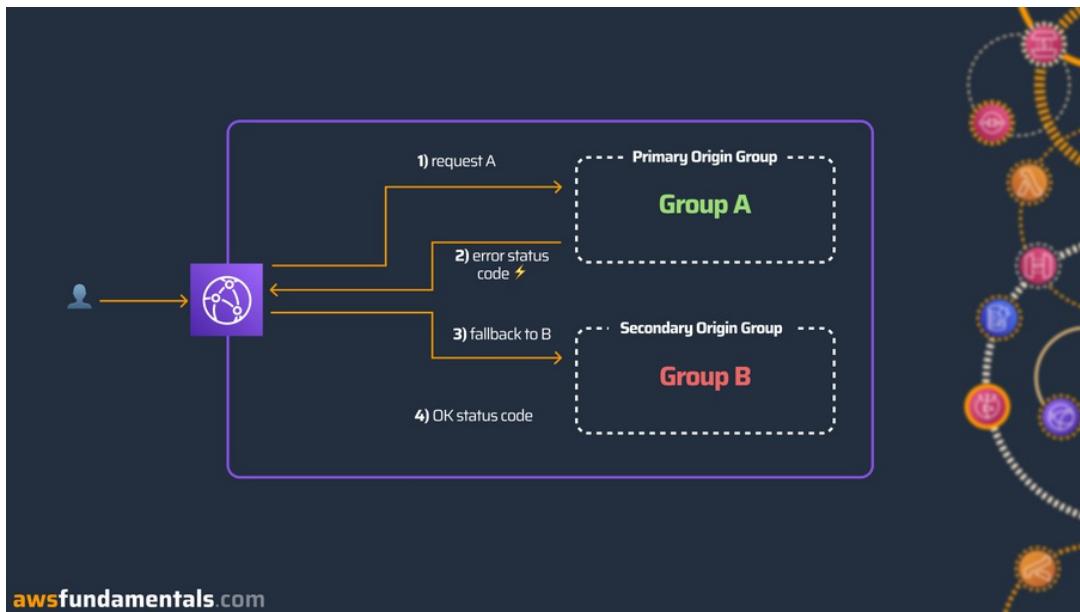
With this you can either:

- allow users to access your content only if they are in one of the approved countries or
- block users from accessing if they're in one of the banned countries in your block list.

The accuracy of the mapping between IP addresses and country is 99.8%. If CloudFront is not able to determine the location, it will always serve the content which was requested.

#### **Redundancy and Failure-Safety via Origin Failovers**

CloudFront also got you covered for high availability by providing Origin Failovers. You can define a primary and a secondary origin group and define which HTTP codes will result in a failover to the secondary origin.



## Creating Our First Distribution: Serving a Static Website from S3 That's Protected via a CloudFront Function

We've learned about all the fundamentals that are important to know about CloudFront. Let's jump into the doing again and create our first small setup that connects multiple services together: **CloudFront, S3, and CloudFront functions**.

What we need to do:

1. **Creating a bucket** that does contain a simple HTML. The bucket should block all public access, as we only want to serve the bucket's content via CloudFront.
2. **Creating a CloudFront distribution** that uses our bucket as an origin.
3. **Attaching a CloudFront function** to our distribution that will prompt the requester for basic auth credentials.

Let's start right away.

### Creating an S3 Bucket with a Simple HTML File

Go to the S3 console and click on `Create Bucket`. Add a unique name for the bucket and afterward directly click on create as, by default, buckets will be blocked from public access.

Now we'll create a simple HTML file that will only greet us with `Hello CloudFront!`.

```

<html>
  <head>
    <style>
      .center {
        display: flex;
        align-items: center;
        justify-content: center;
        height: 100vh;
      }
    </style>
  </head>
  <body>
    <div class="center">
      <h1>Hello CloudFront!</h1>
    </div>
  </body>
</html>

```

Upload the file as `index.html` to our bucket's root directory.

**awsfundamentals-cf** Info

**Objects** Properties Permissions Metrics Management Access Points

**Objects (1)**

Objects are the fundamental entities stored in Amazon S3. You can use [Amazon S3 inventory](#) to get a list of all objects in your bucket. For others to access your objects, you'll need to explicitly grant them permissions. [Learn more](#)

[Actions ▾](#)

[Create folder](#) [Upload](#)

Find objects by prefix

<input type="checkbox"/>	Name	Type	Last modified	Size	Storage class
<input type="checkbox"/>	<a href="#">index.html</a>	html	December 23, 2022, 20:07:07 (UTC+01:00)	279.0 B	Standard

That's already all we need to do in the first step.

## Setting up Our CloudFront Distribution

We can now continue to CloudFront. Go to the CloudFront console and click on `Create distribution`. We'll be prompted with the wizard where we need to choose our origin - select the bucket we created before.

For the origin access, we want to use the new control settings. It will open in a new modal. Stick to the defaults and click `Create`. CloudFront will hand over the generated policy we need to attach to our S3 bucket after we finish the creation of our distribution.

### Create control setting

**Name**  
awsfundamentals-cf.s3.eu-west-1.amazonaws.com  
The name must be unique. Valid characters: letters, numbers and most special characters. Use up to 64 characters.

**Description - optional**  
Enter description  
The description can have up to 256 characters.

**Signing behavior**  
 Do not sign requests  
 Sign requests (recommended)  
 Do not override authorization header  
Do not sign if incoming request has authorization header.

**Cancel** **Create**

Now we already have all we need. We'll stick to the default cache settings and not attach a CloudFront function yet.

## Origin

**Origin domain**  
Choose an AWS origin, or enter your origin's domain name.

**Origin path - optional** [Info](#)  
Enter a URL path to append to the origin domain name for origin requests.

**Name**  
Enter a name for this origin.

**Origin access** [Info](#)

- Public  
Bucket must allow public access.
- Origin access control settings (recommended)  
Bucket can restrict access to only CloudFront.
- Legacy access identities  
Use a CloudFront origin access identity (OAI) to access the S3 bucket.

**Origin access control**  
Select an existing origin access control (recommended) or create a new configuration.

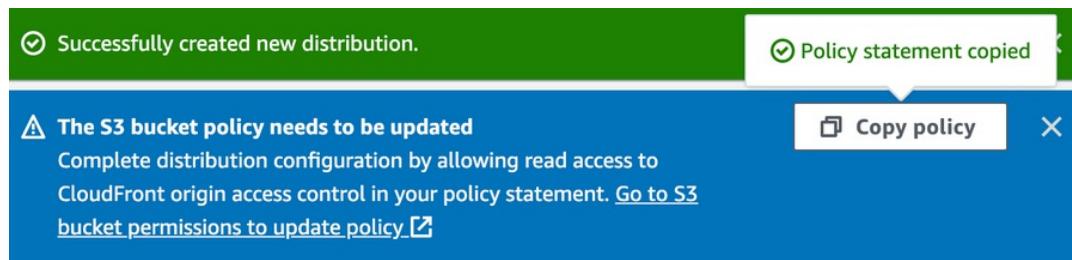
▼
Create control setting

**Bucket policy**  
Policy must allow access to CloudFront IAM service principal role.

- I will manually update the policy

**⚠ You must update the S3 bucket policy**  
CloudFront will provide you with the policy statement after creating the distribution.

As said before, we're prompted that the generated control policy is ready to copy. Click on the [Copy policy](#) button and navigate to our S3 bucket.



Go to your bucket, to the permissions tab, edit our bucket policy and paste our required policy. It should look like this:

```
{
```

```

"Version": "2008-10-17",
"Id": "PolicyForCloudFrontPrivateContent",
"Statement": [
{
    "Sid": "AllowCloudFrontServicePrincipal",
    "Effect": "Allow",
    "Principal": {
        "Service": "cloudfront.amazonaws.com"
    },
    "Action": "s3:GetObject",
    "Resource": "arn:aws:s3:::awsfundamentals-cf/*",
    "Condition": {
        "StringEquals": {
            "AWS:SourceArn": "arn:aws:cloudfront::157088858309:distribution/E1FETG1WT0RMYX"
        }
    }
}
]
}

```

That's it. Our distribution is now already ready to serve our HTML file. Let's open the domain of our CloudFront distribution (you'll find it directly in the distribution overview) and append the `/index.html`.

### **Adding Authentication with a CloudFront Function**

Let's jump to the `Functions` tab in the left-hand navigation menu of CloudFront and create our first function with the name `basic-auth`.

Let's add the following code which will check for basic auth credentials:

```

function handler(event) {
    var authorization = event.request.headers.authorization;
    var expected = "Basic YXdzZnVuZGFTZW50YWxzOlBhc3N3b3JkMSE=";

    if (
        typeof authorization === "undefined" ||
        authorization.value !== expected
    )
}
```

```
) {
    return {
        statusCode: 401,
        statusDescription: "Unauthorized",
        headers: {
            "www-authenticate": {
                value: 'Basic realm="Enter credentials for this super secure
site"',
            },
            },
        };
    }
    return event.request;
}
```

The expected string is a base64 encoded string that is built up via `<username>:<password>`. In our case, it's set to `awsfundamentals` and `Password1!`.

Let's jump to the `Test` tab of our function and check if it does what it should. If we provide the expected header, we'll get an HTTP response.

The screenshot shows the AWS Lambda Test function interface. At the top, there are sections for "Header" (with "authorization" set to "Basic YXdzZnVuZGFtZW50YWxzOlBhc3N3b3JkMSE="), "Request cookies - optional" (with "Add cookie" button), and "Query string - optional" (with "Add query string" button). Below these are "Cancel" and "Test function" buttons. The main area displays the "Execution result" tab, which is selected (indicated by a green checkmark). It shows the following details:

- Status:** Succeeded
- Stage:** DEVELOPMENT
- Compute Utilization Info:** 16
- Output:**

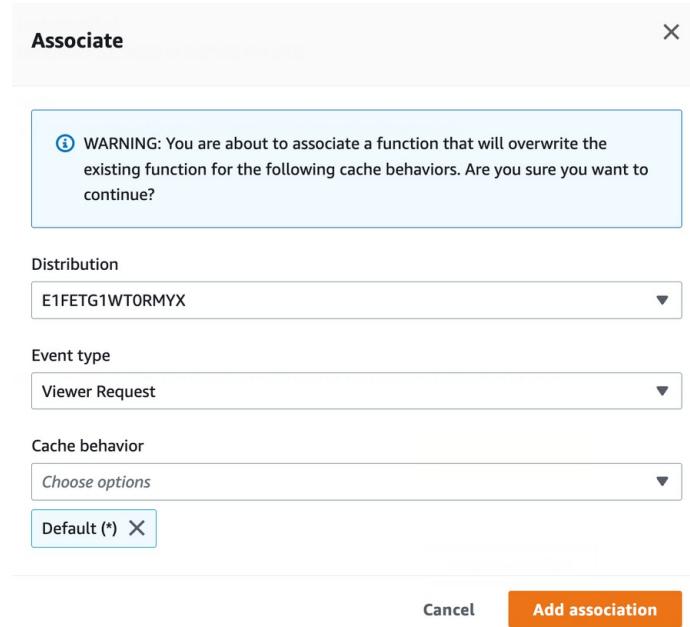
```
{
  "request": {
    "headers": {
      "authorization": {
        "value": "Basic YXdzZnVuZGFtZW50YWxzOlBhc3N3b3JkMSE="
      }
    },
    "method": "GET",
    "querystring": {},
    "uri": "/index.html",
  }
}
```
- Execution logs:** (button)

When not providing an authorization header or an invalid value, we receive HTTP 401. That's what we wanted!

The screenshot shows the AWS Lambda Test function interface. The "Execution result" tab is selected (green checkmark). It displays the following details:

- Status:** 401 Unauthorized
- Stage:** DEVELOPMENT
- Compute Utilization Info:** 19

Let's publish our function via `Publish > Publish function` so we can attach it to our CloudFront distribution. This can be done in the new section that will pop up right below.



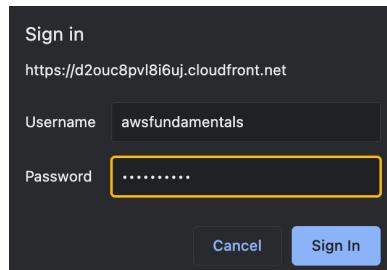
Select our distribution, chose `Viewer Request`, and select the default cache behavior.

Associated distributions		<a href="#">View distribution</a>	<a href="#">Remove association</a>	<a href="#">Add association</a>
		<a href="#">Search distributions and cache behaviors</a>		
Distribution ID	Description	Cache behavior	Event type	
E1FETG1WT0RMYX	-	Default (*)	Viewer Request	

That's it already. If we jump back to the distribution overview, we'll see that CloudFront is deploying our update. This can take up to several minutes, so don't wonder.

Origins	Status	Last modified
awsfundamentals-cf.s3.eu-west-1.amazonaws.com	<span>Enabled</span>	<span>Deploying</span>

After the deployment is finished successfully, we can go to our distribution's URL again and retrieve our HTML file. We'll be prompted for our credentials now!



After you've typed them in, we'll see our welcome message again!

That's it! We've built a small static page delivery project with CloudFront, S3, and some small computation on the edge.

### **Expected Costs with CloudFront**

CloudFront follows a pay-as-you-go model, like other managed services by AWS. If you're using AWS native services for your origins, you're not paying additional fees for the data transfer between those services and CloudFront.

CloudFront has two pricing components: **data transfer charges** and **request charges**.

1. **Data transfer charges:** These charges are based on the amount of data that is transferred to end users through CloudFront. Data transfer charges vary depending on the region where the data is transferred and the type of data transfer (e.g., HTTP or HTTPS).
2. **Request charges:** These charges are based on the number of requests that are made to CloudFront. Request charges vary depending on the type of request (e.g., GET, HEAD, POST) and the region where the request originates.

CloudFront also has fees for specific features, such as SSL/TLS certificates and invalidation requests (removing objects from CloudFront's edge caches).

Looking at the AWS Free tier, you're granted the following free contingents each month:

- 1 TB of outgoing data transfer
- 10,000,000 HTTP and HTTPS Requests
- 2,000,000 CloudFront Function invocations

You can always use the pricing calculator to estimate the cost of using CloudFront if you have a rough guess about the expected data transfer and request volumes, as well as the specific CloudFront features that you plan to use.

### **Monitoring Your Distributions with CloudWatch**

CloudFront generates different types of reports which enable you to analyze how your distribution is used & by which audience.

- **Cache Statistics** - requests overview by status code and method, cache hits, misses, and

errors.

- **Popular Objects** - most requested files, including cache-hit ratio for those files.
- **Top Referrers** - the 25 top sources for requests.
- **Usage** - number of requests, transferred data by protocol and destination.
- **Viewers** - including devices, browsers, operating systems, and locations.

## The Manifold Use Cases of CloudFront

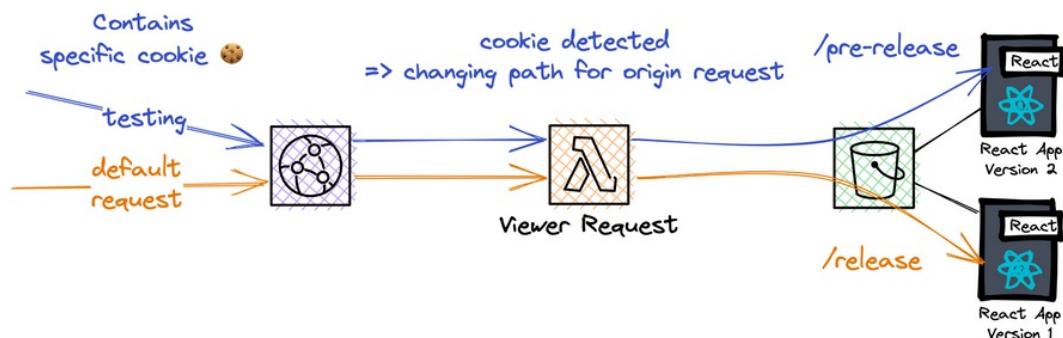
We've already gone through the benefits of using a CDN for the distribution of your content. This is true for any web application that is delivering static content in any form. But due to CloudFront's capability of running custom business logic at its edge locations, there's a lot more you can do with just a few lines of code.

Let's have a look at two of them.

### Use Case 1: Routing Traffic between Different Origins to Access Different Versions of Your Application

When building web applications, it's a common practice to have a preview stage for testing explicitly before bringing the new version to production.

CloudFront makes this very simple for frontends that can be exported to static HTML, JS, and CSS files. There's only a need for another bucket that will receive the contents of upfront versions of your releases and a Lambda@Edge or CloudFront function that will take care of dynamically routing to the right origin.



This can be based on a custom header or cookie of your choice. With browser plugins like Requestly, you can easily switch between configurations.

And it doesn't end here: you're not only able to route between origins but also rewrite paths. There's nothing stopping you from putting every build of your frontend into a separate folder and defining routing rules for each of those versions.

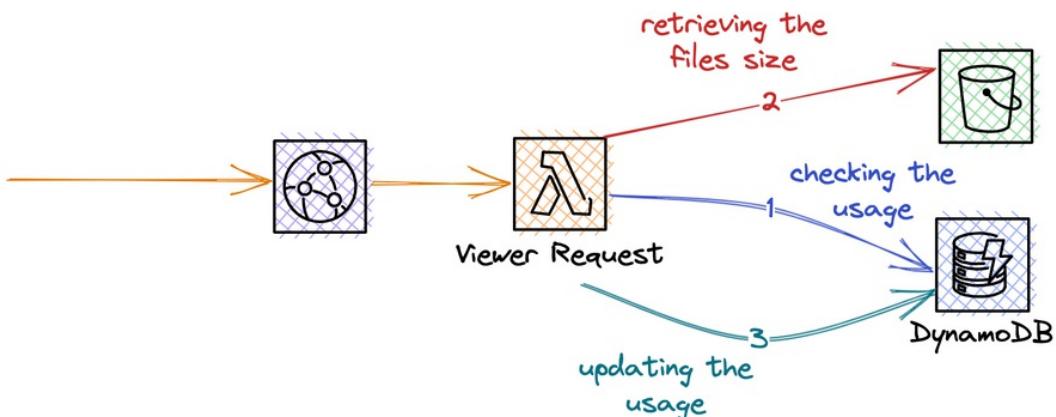
#### Use Case 2: Counting Transferred Bytes and Limiting Downloads per Customer

When using Lambda@Edge, you are not just limited to the variables in your function and the request context. Your edge functions do have network access and which allows them to access other AWS services if the required permissions are granted via IAM, which ends up with almost limitless possibilities.

Let's have a look at a simple example: your customers are allowed to upload media files that can later be distributed to their friends via download links. As data transfer with CloudFront is not free and sizes of photos and videos are generally very large today due to great smartphone cameras, this could quickly become a cost trap if you just collected too many power users.

A simple Lambda@Edge function is able to help. For each request to your CloudFront distribution that targets an archive, we need to run through a set of steps.

1. check if the current limits are already breached - in the example we're using DynamoDB to count the downloaded bytes per customer. If it exceeds a certain threshold, we'll directly return an error response, e.g. HTTP 429 indicating that limits are reached.
2. get the size of the target object via a HEAD request.
3. collect the downloaded bytes and add them to the user's state.



As Lambda@Edge functions are billed in 1ms periods, DynamoDB is accessed within single-digit milliseconds and everything is billed on-demand, this feature doesn't come with much or

any additional costs if your application doesn't reach a very large scale.

## Tips and Tricks for the Real World

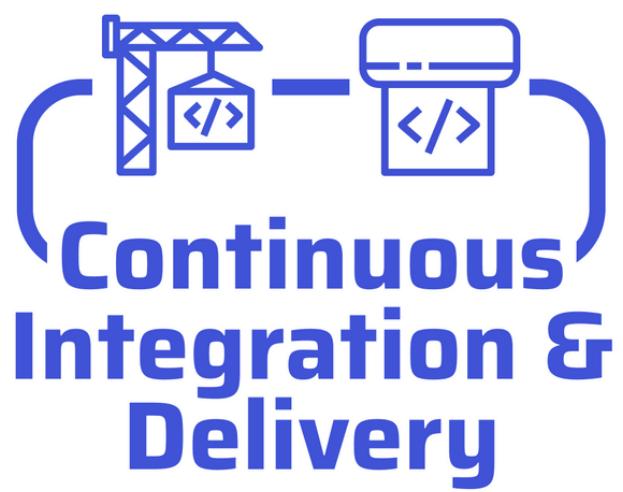
CloudFront is powerful and enables you to not only host simple websites close to anyone in the world but much more. Let's go through some important reminders which help you to get the best out of CloudFront.

- **CloudWatch automatically integrates with CloudWatch** - this lets you monitor CloudFront's performance for each of your distributions. It includes data transfer rates, request rates, and error rates.
- **CloudFront functions can be tested in CloudFront's console interface** - deploying updates to CloudFront functions can take a long time, as they need to get published and then distributed around the globe. Luckily, there's a feature in CloudFront's console interface that lets you test functions upfront. You can provide custom request paths, query parameters, and headers to see the results of the functions before actually deploying them. This saves a lot of time and headaches.
- **Edge functions will write logs to the region where they were invoked** - even though Lambda@Edge functions need to be published in `us-east-1`, the logs will be written in the regions where the function was invoked. If you're located in Germany, most likely an edge-functions container in `eu-central-1` will be started and invoked. The corresponding logs will also end up there. The function's log group name will nevertheless contain the function's origin `us-east-1`.
- **Providing a custom domain name to your function** - if you're having a custom certificate for your domain in AWS Certificate Manager (which is free), you can easily attach a custom domain name to your distribution. After creating a corresponding DNS entry, either in Route 53 or in your favorite domain name service, it will be available under this domain. The default CloudFront URL will be still available in parallel.
- **You can integrate with multiple origins at the same distribution** - this allows you to distribute content from multiple sources, such as S3 and an on-premises server. With CloudFront functions or Lambda@Edge, you can easily route between those destinations based on every possible rule.
- **You can run multiple caching rules in parallel** - maybe you don't want to cache content in a specific path as those files always need to be returned in their latest version. This is perfectly possible with CloudFront, as it's up to you how CloudFront should serve

content and how to evaluate and evict the cache.

## Final Words

CloudFront is a globally distributed network of edge locations that speeds up the delivery of static and dynamic web content, such as HTML, CSS, JavaScript, and images. It integrates with other Amazon Web Services services to give an easy way to distribute content to end users with low latency and high transfer speeds.



# Continuous Integration & Delivery

Nowadays, building and delivering software is an iterative approach. We've moved away from the waterfall model of completely designing an application from A to Z, building all features in one rush, and shipping everything to production.

One of the key principles of agile development is to break the software development process into small, manageable chunks. The cross-functional team of developers, designers, and other stakeholders work together to deliver a usable piece of software in small cycles.

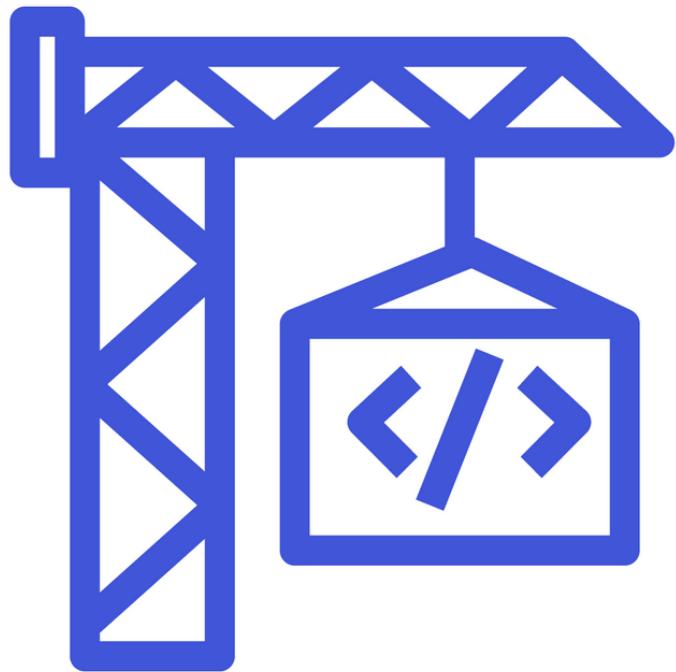
The process of releasing software has to be an automated process. That also ensures that quality standards are met and existing features are not broken with updates. And that's exactly where **continuous integration & delivery (CI/CD)** tools are stepping in.

**Continuous integration** is the practice of continuously integrating code changes into a shared code repository. As developers write code, they frequently push it to the code repository, where it is automatically built and tested. This helps to catch errors early in the development process and reduce the risk of integration problems later on.

**Continuous delivery** takes it a step further by automating the deployment of code changes to different environments such as development, QA, and production. This allows development teams to quickly and safely roll out new features and updates.

**AWS CodeBuild and CodePipeline** are two AWS services that play a key role in supporting CI/CD for web applications. **CodeBuild** is a fully managed build service that compiles source code, runs tests, produces software packages, and deploys them into the environment.

**CodePipeline** is a fully managed service on top of CodeBuild. It allows you to model and visualize the entire release process by building delivery pipelines that are built up via different CodeBuild jobs and go from checking out source code to deploying infrastructure and the final distribution to production.

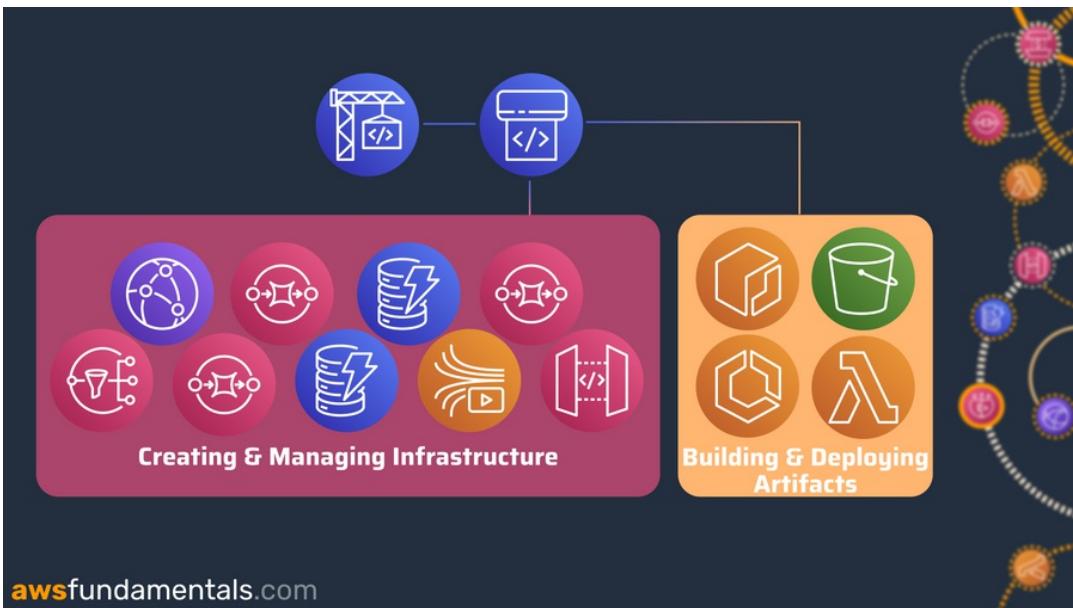


## AWS **CodeBuild** & **CodePipeline**

# Creating a Reliable Continuous Delivery Process with CodeBuild & CodePipeline

## Introduction

AWS CodeBuild and AWS CodePipeline are fully-managed continuous integration services that help you to build, package & deploy your application in a safe and reliable manner. It enables your team to focus on building the actual application and worrying less about efforts or operations to ship it to customers.



CodeBuild and CodePipeline help you build your deployment package. It can also create and manage infrastructure via Infrastructure as Code tools like Terraform, CloudFormation, Serverless Framework, CDK, and many others.

## An Automated Continuous Integration & Delivery Process Ensure Reproducible and Reliable Application Releases

Compiling applications mostly requires a specific environment that offers all the needed tools, languages, and frameworks that are part of the build process. Furthermore, you want to deploy changes to your application or product regularly. This ensures that the current development version that is out there for either internal or external testing doesn't differ too much from the latest state of development. Thirdly, even a small-sized team of developers is usually very diverse, not only from a cultural perspective but also from a technical point of view. Everybody got his own preferences for development tools or even operating systems.

We also want to decouple the development process from the release process. Even more, we want to restrict manual production access by developers and team members. This is important to avoid human error. It's often also necessary to fulfill compliance requirements that may apply in a given country.

Lastly, in cases of any issue, we want to be able to easily roll back to a previous version to restore a healthy application state.

Let's rephrase our requirements into a short summary:

- we require a **small or even large toolchain** for our build processes
- we want to have **regular rollouts** of new changes
- **we don't want to rely on our build processes** on developers, as development ecosystems may differ greatly from each other
- we want to **restrict manual access to production systems** to reduce the chances of human error
- we want to have **reproducible builds and deployments** - including rollbacks to previous versions in case of issues

This sums up the following: we're in need of a dedicated build environment with a stable toolchain, that runs without the necessity of human interactions and which is strictly separated from a security point of view.

And that's exactly what continuous integration & delivery services and platforms like AWS CodeBuild & CodePipeline are built for.

#### **AWS CodeBuild for Building Your Projects and Applying Your Applications and Infrastructure**

As the name already implies, CodeBuild is the service that actually **builds**. But don't get too caught up in this phrase, as you can basically execute **anything** at CodeBuild. It doesn't just strictly map down to packaging and building applications.

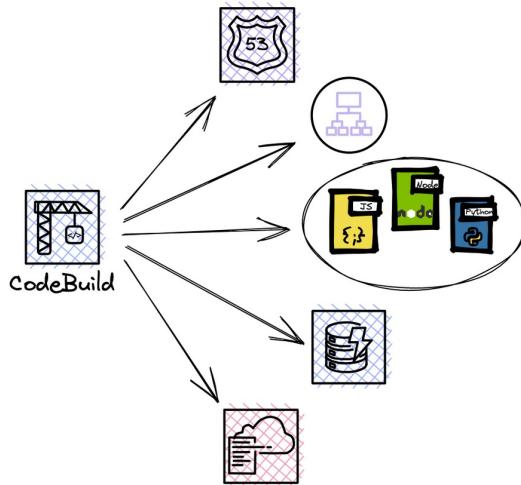
You're also able to:

- execute infrastructure manipulations via infrastructure as code tools like Terraform or CDK.
- change routing destinations, e.g. by changing Route 53 record weights or adapting the

target group weights of your load balancer.

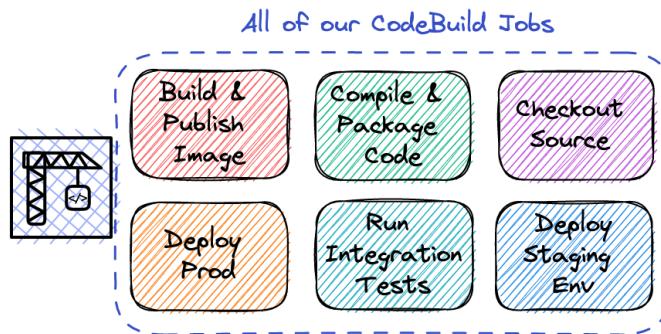
- execute the creation of backups, e.g. for DynamoDB.
- trigger Lambda functions to run batch jobs.
- or any other process that can be poured into a script.

As CodeBuild can run any container image you want to provide, there are no limits on which tasks it can take over.



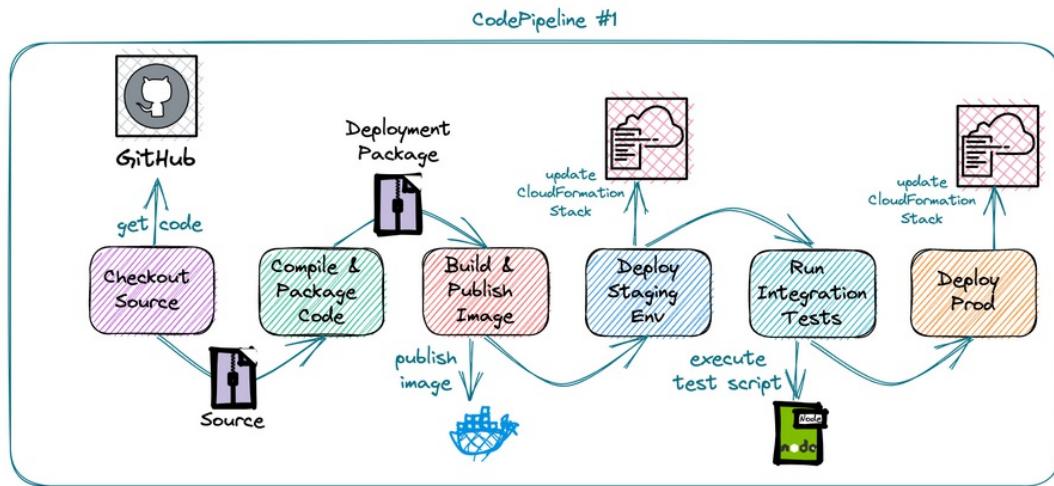
### AWS CodePipeline for Orchestrating All Your Build Jobs and Deployments

A structured and well-thought-out delivery process doesn't just include a single build job. It's an orchestration of many jobs, that enable you to deploy code or infrastructure in a reliable and safe manner.



And that's exactly what AWS CodePipeline does: it's your CodeBuild orchestration tool. With CodePipeline, you can create pipelines that connect jobs into stages to create a multi-step

rollout process that can be easily reproduced and understood.



### The Symbiosis of Both to Create a Resilient Delivery Process

If you want to stay within AWS for your delivery process, CodeBuild is a must. You don't *have to* use CodePipeline but it offers itself as it perfectly extends the features of CodeBuild to build sophisticated continuous integration & delivery processes.

Those services work together very well and are quite easy to configure and set up.

### A Serverless and Managed Service That Lets You Focus on What's Important

One major point that really speaks for AWS CodeBuild and CodePipeline: both are completely managed - there are **no upfront costs nor costs for idling servers**. You only pay per build minute and a very small fee per existing pipeline (currently \$1 per month). You don't pay anything for the first 30 days of each pipeline.

Not having to maintain any build servers is a huge benefit and saves a lot of effort and operational burdens that can be used more sensibly in other places.

### Understanding the Key Terms of CodeBuild and CodePipeline

For understanding how CodeBuild and CodePipeline work, we need to go over their key terms and fundamentals.

## **Build Images & Containers as an Underlying Base for Your Jobs**

Builds are executed within containers. For the images that run within the containers, you'll have two options:

- provide your own image either via ECR or another non-AWS repository
- use one of the managed images by AWS CodeBuild

If you're using the managed images, they already include the runtimes for most programming languages.

## **Build Specs That Are the Blueprint of What a Job Needs to Do**

Build specs define what your build job should actually do. Each spec file is a YAML and contains various build commands and related settings.

You can provide your build spec files either in your source code or provide a spec file when you create your build project. If provided via the source code, CodeBuild will look by default for **buildspec.yml** in the root directory. You can adjust this in your build project.

Your buildspec file can be organized via the different phases which are executed by CodeBuild. Let's have a look at a small example.

```
version: 0.2

env:
  # injects secrets into environment variables
  secrets-manager:
    MY_SECRET: some/secret
    # injects ssm parameters into environment variables
  parameter-store:
    MY_SSM_PARAMETER: /some/ssm-parameter
phases:
  install:
    commands:
      - npm i
  build:
    commands:
      - npx sls package
```

```

artifacts:
  # archive files which can be injected into downstream build projects
  # that follow in your pipeline created via AWS CodePipeline
  files:
    - '.serverless/**'

# cache paths so the next execution will be faster
cache:
  paths:
    - '.node_modules'

```

## Using Phases to Further Structuring Your Jobs

Looking into the phase details of our example, we can see the phases CodeBuild will run through. An example execution looks like this:

- (<1s ms) **Submitted** - your job is submitted to CodeBuild's internal job queue.
- (32s) **Queued** - the job is waiting for unreserved compute resources.
- (53s) **Provisioning** - downloads your image & starts the container.
- (4s) **Download Source** - downloading your source code.
- (160s) **Install** - runs our NPM install.
- (53s) **Build** - runs our Serverless packing command.
- (<1s) **Upload Artifacts** - archives our artifacts and uploads them to S3.
- (2s) **Finalizing** - completing the job & freeing resources.
- **Complete** - build project is completed.

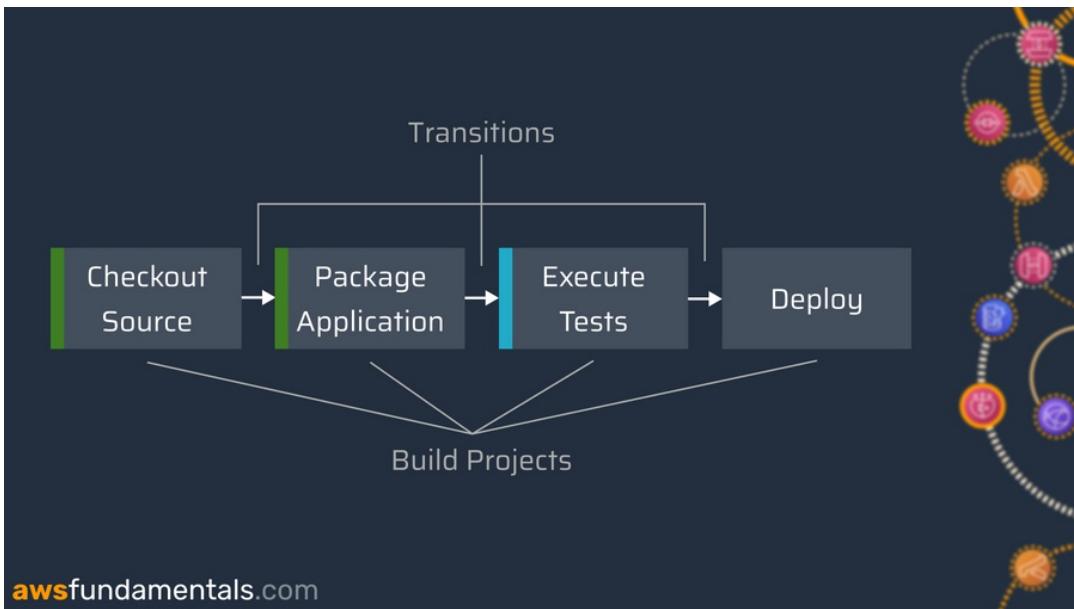
## Sources to Define Where Your Code Comes from and Triggers to Automatically Run Your Jobs Based on Conditions

CodeBuild can check out a repository in the beginning. As a source provider, you can use Amazon S3, AWS CodeCommit, GitHub, or BitBucket.

Additionally, you can define if builds should be triggered automatically on source changes, e.g. a new commit to a specific branch in your repository.

## Pipelines to Align Your Jobs and Orchestrate Them into Steps

Pipelines are an orchestrated collection of build projects. It allows structuring continuous integration and deployment into phases, including builds, quality gates, and actual deployments.



The actual structuring of your pipelines is completely up to you and your requirements. You can orchestrate as many projects as needed and with as many quality gates as required.

## Moving Outputs of Your Jobs through Different Stages of Your Pipelines via Artifacts

At the end of a build project execution, you can choose to archive artifacts (**outputs**) which will then be saved at S3 and can be injected into other build projects (**inputs**) of your pipeline execution. CodePipeline itself will keep track of versioning, so you'll always end up injecting the outputs of your current pipeline execution.

```
artifacts:
  # default artifact 'build_output'
  files:
    - 'dist/**'
secondary-artifacts:
  # additional named artifacts
  terraform_output: # name of the artifact
  files:
```

```
- '**/*'  
base-directory: 'infra'  
name: 'terraform_output'
```

The default output artifact will be named `build_output`. There's the option to create additional output files via `secondary-artifacts` with a unique naming.

### **Monitoring Your Delivery Pipeline to Quickly Get Aware of Issues**

Your CodeBuild and CodePipeline projects execute the most critical actions in your AWS account. That's why monitoring is crucial, as with every other resource.

#### **Logging**

All console output at CodeBuild is by default ingested at CloudWatch if the necessary permissions are assigned to the service roles. This allows for traceability if build projects fail.

#### **Build Notifications**

AWS CodePipeline integrates with AWS Chatbot which allows for simple and readable notifications via your favorite communication tools, e.g. Slack.

You're also able to filter for events, as you may only want to get notified for specific events like failures.

### **Securing Your Pipelines with AWS IAM, VPC Integrations, and Encryption**

Your continuous integration & delivery solution will likely have the broadest permissions as it needs to be able to create, update and destroy your infrastructure and deploy code artifacts and may be the only entity to access the production environment.

This makes it a critical component from a security perspective.

#### **Using IAM Roles to Grant Permissions to Build Projects and Pipelines**

CodeBuild and CodePipeline are fully integrated into AWS IAM. Each of your build projects and pipelines will use IAM service roles to get their permissions.

This means that permissions can be as restrictive as needed and do not need to be shared in the first place.

## **Integrating Your CodeBuild Projects into a VPC to Access Restricted Resources**

By default, CodeBuild projects can't access resources that reside in a VPC. By adding your VPC ID, the VPC subnet IDs, and the VPC security group IDs to your build project configuration, CodeBuild will be able to access private resources in your VPC.

Example use-cases that require VPC integration:

- your relational database resides in a VPC and integration tests do require access
- regression tests require directly verifying cached data in a Redis clusters
- a build step requires access to a web service that only allows requests from allow listed IPs

If there is no dedicated requirement, CodeBuild doesn't need to be integrated with a private VPC as it's fully protected by IAM.

## **Encrypting Your Artifacts to Have Additional Protection for Confidential Outputs of Your Build Jobs**

Your build outputs are sensitive information. As they will be saved at S3 you can enable encryption as with all other objects. You can either choose to use that S3-managed encryption key or go with a dedicated customer-managed key from KMS only for your CodePipeline project.

This also applies to caching.

## **Enforce Manual User Interaction for Critical Stages via Approvals**

Your typical release pipeline does not run through all stages without user interaction. Most teams do want to have a fixed release cycle which requires developers to manually approve the deployment to production.

CodePipeline meets this requirement via approval steps that can be integrated between build projects and do require a response from a user that has the needed `codepipeline:PutApprovalResult` permissions assigned.

Users will also be asked to put a message for either approval or rejection, so decisions can be traced later.

## **Getting an Understanding of How CodeBuild and CodePipeline Is Billed**

As mentioned earlier, CodeBuild and CodePipeline are solely pay-per-use except for a small fee per created CodePipeline project.

The price per build minute depends on the memory & CPU configuration of your build container. AWS specifies this as **small**, **medium**, or **large**.

## **Advantages and Downsides**

After going through all the fundamentals, it's great to recap what AWS CodeBuilds and AWS CodePipelines do very well and what might be improved in the future. In any project, you'll spend a significant amount of time with continuous integration & delivery services so it's important that it does meet your requirements.

### **CodeBuild and CodePipeline Shine through on-Demand Pricing, Little Operations, and High Reliability**

There are a lot of things done right by both services.

- **Pay-as-you-go Pricing** - you'll never pay for idling build agents as it's an on-demand service with linear pricing based on your actual usage. This makes it a perfect fit for side projects or any project that does not require running builds 24 hours a day.
- It's **fully managed** - a managed service is a good service. In the case of AWS CodeBuild and AWS CodePipeline, you don't need to maintain anything besides the configuration of your projects.
- **It is highly available** - both services offer high availability and also grant service credits for certain availability levels that are not met within a month. You'll receive 10% of your charges back for a monthly uptime percentage that is less than 99.9% (less than 1 hour per month), 25% for less than 99%, and full reimbursement of costs for less than 90%.

### **There's Room for Improvement regarding Build Times, Interface Design, and Container Images**

There's no perfect service or solution for anything. This includes AWS CodeBuild and CodePipeline. Let's visit the most prominent critics.

- **A lot of build time is added due to container and infrastructure provisioning** - each CodeBuild job will bootstrap a build container with your desired image and therefore

run to a lot of initial phases. This includes: submitting the job to AWS' CodeBuild queue to wait for computation resources, preparing the needed infrastructure for your build environment, downloading the build image, starting the container itself and maybe executing any pre-build commands, installing scripts, or downloading outputs from previous build jobs. All of this takes time for which you'll be billed. In non-serverless build environments with steady build agents, this happens way faster as the infrastructure stays intact, containers are already running and caches may be already in place at the build agent.

- The **console interfaces are not intuitive or clearly arrange** - the console interface has a lot of nested views and is very heavy on written texts instead of good visualizations. Also, there's no single overview for the build projects of several pipelines which developers are used to if they know build tools like Jenkins.
- **On-demand Pricing can be a trap** - we've listed this on the pro side, but it can also be negative. For projects that require constant execution of builds, CodeBuild can get very expensive. Have a second thoughts and use AWS Pricing Calculator to get an estimate of your bill beforehand if your project meets either one of the following:
  - extensive automated quality gates and testing that does require a lot of time
  - many automated checks on pull requests or branches
  - many build projects that are executed without much idling times
  - very regular release cycles, maybe even multiple times per dayEither one of those criteria may result in a large bill only for CodeBuild. A build tool with steady build agents that are always available (e.g. self-hosted Jenkins on ECS with Fargate) may be a cheaper option.
- **Self-maintained container images increase operational costs** - AWS offers managed images for CodeBuild and you're able to choose to automatically make use of the latest version. It contains a default set of developer tools and languages that are sufficient for a lot of cases. Nevertheless, you may have requirements for a lot more tools and dedicated versions which result in having to use your own container image. Relying on non-managed container images results in operational costs in keeping them up to date by updating them in a regular manner. If your application landscape may be built via a lot of different tools and technologies, this can result in having a single very large build image for all build projects, spiking build times due to longer phases for downloading and bootstrapping containers, or several, dedicated container images that do require more efforts to be kept up to date.

## **Use Cases For AWS CodeBuild and CodePipeline**

The main use case for both services is to automate all the regular processes in all environments of your software development process. Generally speaking, each CodeBuild job is an on-demand container that can be used to execute automations. With CodePipeline, you're able to arrange jobs and put them into a simple or complex orchestration.

### **Use Case 1: Building and Pushing New Docker Images to ECR Which Will Later Run in Your Fargate Tasks**

In the ECS chapter, we've gone through the example of how to run applications, which are wrapped in a Docker image, in ECS. In a real-world example, the whole process of deploying a new version of our application would be automated.

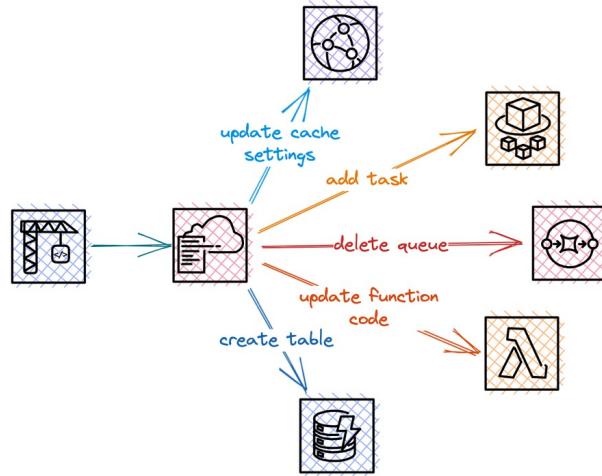
This involves several steps:

- compiling our code.
- building our Docker image.
- pushing the new version of the image to ECR.
- updating our container's reference in the task definition to point to our new image URL.

This can be solely done via scripts (e.g. with Node.js, Python, or Go) and the AWS CLI or by a combination of scripts and Infrastructure-as-Code tools.

### **Use Case 2: Applying and Updating Your Infrastructure**

Today, continuous integration tools like CodeBuild not only compile code and deliver it to a server. It's also used to create and update your whole account's infrastructure via Infrastructure-as-Code tools like for example CloudFormation, AWS Cloud Development Kit (CDK), or Serverless Framework.



We'll see how to work with some of the most prominent Infrastructure-as-Code tools in the previous chapters.

### Use Case 3: Running API Integration And End-to-End Tests

Automating processes will make your life easier. But it can also be dangerous, as those automations are out of your control. Especially when looking at our Infrastructure as Code templates.

An unwanted change could drop a table, remove a running ECS task that is critical for your application or even delete a whole database.

This is why it's important to have multiple environments that are as similar as possible. It's also important to integrate automated quality gates into your release process so that you can ensure (to some grade) that things are working as expected.

### Tips & Tricks for the Real World

It doesn't take much to get started with AWS CodeBuild and CodePipeline, but there are quite a few things to know to get the best out of it. Let's revisit them again and also learn a few more of the options to fine-tune our build processes.

- **Make use of the caching feature** - regularly installing a huge set of dependencies takes a lot of time which you'll get billed for. Setting up proper caching will help to reduce build times, and therefore the cycle time from starting a pipeline to deploying your application, significantly. Also, try to keep very steady dependencies directly in the Docker image.
- **Use environment variables to pass information to your builds** - don't hardcode

secrets anywhere, but pass them from the parameter store (part of AWS Systems Manager) or the AWS Secrets Manager. Environment variables are also perfect for passing specifics like the branch to build or the version number that should be generated.

- **Stick to providing your build definitions with a buildspec file in your repository** - providing the buildspecs with your code gives you more flexibility, as you can adapt steps without changing any infrastructure or even logging into AWS.
- **Integrate your jobs with SNS and ChatBot to forward build issues** - if issues don't get actively forwarded to places that get much attention, nobody will take care until it's really necessary. This is the same for build pipelines. With AWS Chatbot, you can forward issues (or even successful executions of a pipeline or job) through SNS to your favorite communication channel, e.g. Slack. This will help to create transparency.
- **Add approvals for critical jobs** - running critical jobs, like deploying code to production, should generally be approved by at least one person. This adds an extra layer of review and control to your pipeline. Even though teams are having the goal of automated tests that check every regression (validating that existing features still work), it often remains a goal instead of becoming reality. Mostly, at least a few processes and features need to be tested manually before the new version can be safely brought to the production stage.
- **You can archive build artifacts to S3** - this includes things like compiled code or test results. This can help to get insights into the effectiveness of your development process or to debug unexpected issues in the future.
- **Use build badges to show the state of your builds** - CodeBuild does support build badges that can be used to easily display the current state of your build in different places like README files.
- **Change-based triggers to only run necessary builds** - CodePipelines support triggers that can check that there are changes in certain files or directories. If there are only changes in other places, the trigger won't fire.
- **Parallelize processes with parallel actions** - CodePipeline supports parallel actions to run multiple processes at the same time, such as compiling code and running tests simultaneously. This helps to reduce build times and achieve faster cycle times.

## **Final Words**

AWS CodeBuild and CodePipeline are not the holy grail of continuous integration & delivery solutions due to constraints like a sluggish user interface, comparably slow build times, and a list of missing features.

But nevertheless, it's a mature service, that offers high reliability, and fair pricing and does not come with many operational costs. Due to its native IAM integration, there are no additional efforts to make it as secure as any other resource of your AWS account which comes with a huge plus.



# Observability

Creating software products isn't only about building. It's also about running and operating. This daily task requires to have deep insights into how an application behaves by monitoring key metrics like request rates, response times, error rates, and resource utilization. It's also necessary to detect and diagnose issues quickly and to identify bottlenecks that result in performance issues.

All of those topics can be bundled into the topic of observability and **CloudWatch** is the key player in mastering it.

CloudWatch allows you to collect, analyze, and view system, application, and custom log files. It also monitors various metrics like CPU, memory, and disk usage as well as response times and error rates. It also comes with the ability to send notifications when defined thresholds are breached or errors are detected or to automatically respond with actions.

Additionally, CloudWatch provides a powerful **query language and dashboards** for analyzing and visualizing data. Together, these features make CloudWatch an essential service for every application.



## Amazon **CloudWatch**

# Observing All Your AWS Services with CloudWatch

## Introduction

Amazon CloudWatch is AWS's central logging and metrics service. Each AWS service logs to CloudWatch.

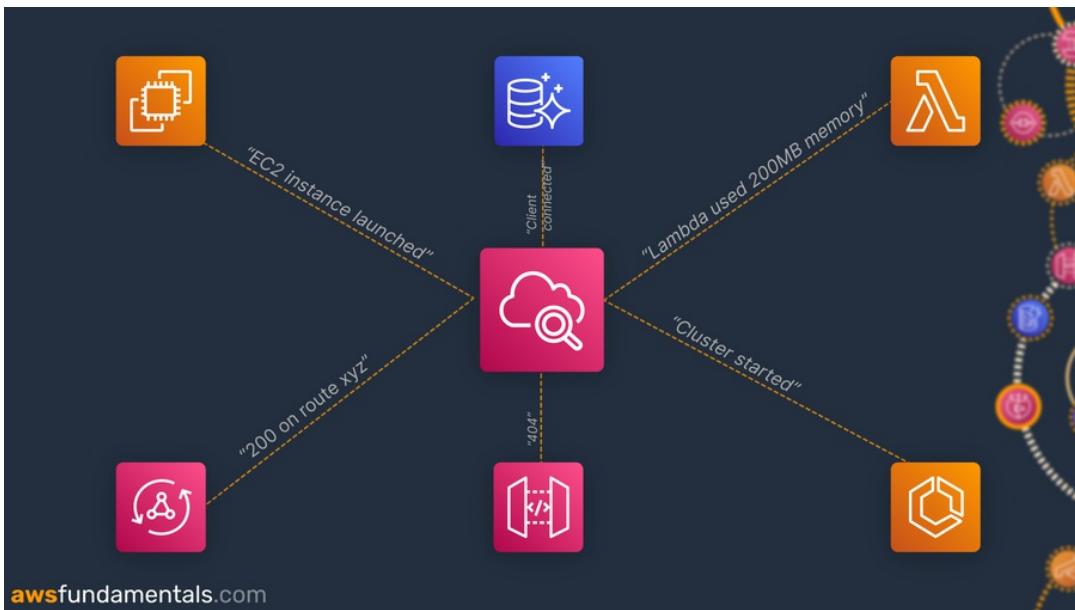
CloudWatch also collects metrics. For example, the number of Lambda invocations, free space in your database, or how much CPU your ECS cluster uses. Based on these metrics you can create **alarms** for certain thresholds. If your free space is below a certain threshold you can get notified.

CloudWatch is one of the **most underrated** services to learn for beginners. You will need to use CloudWatch **a lot**. Especially to understand how your application performs. But also to debug your application. It doesn't matter in which way or how you develop your application on AWS. You will need to use it.



CloudWatch is differentiated into different products. We will mostly focus on **CloudWatch Logs**, **CloudWatch Metrics**, and **CloudWatch Alarms**. Slightly, we will introduce CloudWatch Synthetics.

## CloudWatch Logs Is the Centralized Logging Place for All AWS Services



One of the core functionalities of CloudWatch is **CloudWatch Logs**.

This is the centralized logging space in AWS. Services like Lambda, API Gateway, or ECS log directly into CloudWatch Logs.

This is a huge benefit when working with AWS. You have **one central space** where all your logs are stored.

#### Logs Are Text Outputs of Your Application

Logs are the text output of your application. For example, if you put a print statement (`console.log` for Node.js) into your Lambda function and run it. You will find the log in CloudWatch.

Often you will use a dedicated logger within your application that logs more than just a message. For AWS Lambda you can make use of the amazing powertools developed by AWS. In the following example we have used a JSON logger to log an example message:

```
{
  "message": "This is a message",
  "awsRegion": "eu-central-1",
  "functionName": "ThumbnailLambda",
  "functionVersion": "$LATEST",
  "functionMemorySize": "512",
  "awsRequestId": "ghjkgijk-7466-497e-ac55-3d9c1d9beee0",
```

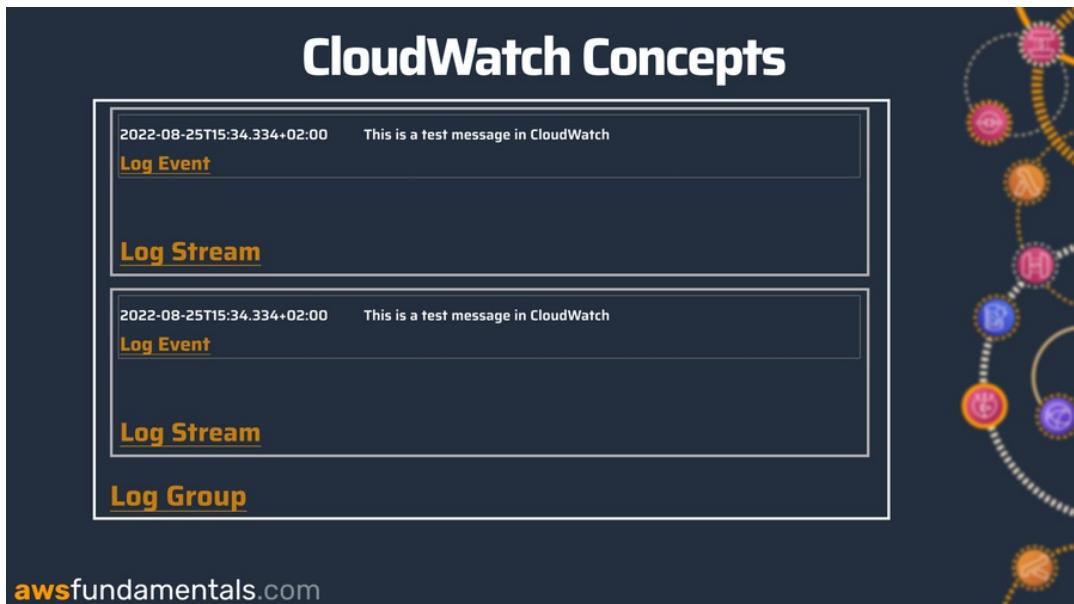
```

    "x-correlation-id": "ghjkgkhj-7466-497e-ac55-3d9c1d9beee0",
    "sLevel": "DEBUG"
}

```

### CloudWatch Follows the Concepts of Log Streams, Groups, and Events

Let's first dive into some concepts of CloudWatch

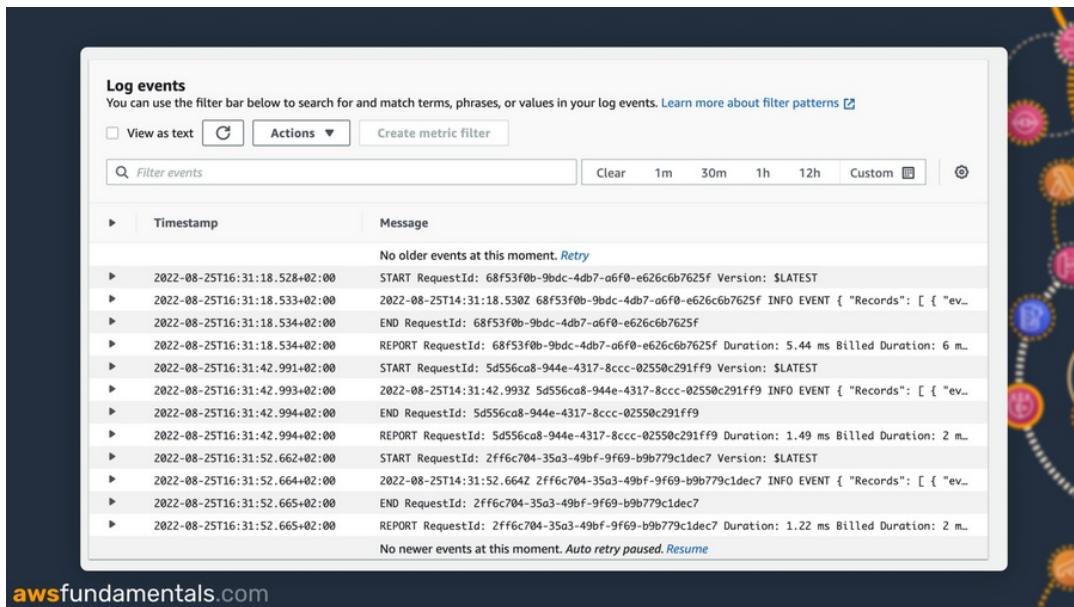


Name	Definition
<b>Log Events</b>	A log event is your actual log statement. It contains the timestamp of your log and the raw log statement you put into it.
<b>Log Streams</b>	A log stream contains one or more log events from the same source. For example, a log stream of a Lambda function can contain more executions of the same Lambda.
<b>Log Groups</b>	A log group is a container that holds multiple log streams. Typically one log group is dedicated to one service. One Lambda function for example has one log group.
<b>Metric Filter</b>	You can create metric filters to get metrics out of your log events.

Name	Definition
<b>Retention Setting</b>	The retention setting defines how long your logs are stored in CloudWatch. CloudWatch also costs money so it is important to not store the data indefinitely.

## Log Events Are the Actual Text Outputs of Your Application

Log events are the actual text output.



The screenshot shows the AWS CloudWatch Log Events interface. At the top, there's a filter bar with options like 'View as text' (unchecked), 'Actions' (dropdown), 'Create metric filter' (button), and a search bar labeled 'Filter events'. Below the filter bar is a timestamp range selector with options: 'Clear', '1m', '30m', '1h', '12h', 'Custom' (with a calendar icon), and a 'Retry' button. The main area displays a table of log events:

	Timestamp	Message
▶	2022-08-25T16:31:18.528+02:00	No older events at this moment. <i>Retry</i>
▶	2022-08-25T16:31:18.530Z	START RequestId: 68f53f0b-9bdc-4db7-a6f0-e626c6b7625f Version: \$LATEST
▶	2022-08-25T16:31:18.534+02:00	2022-08-25T14:31:18.530Z 68f53f0b-9bdc-4db7-a6f0-e626c6b7625f INFO EVENT { "Records": [ { "ev..."}
▶	2022-08-25T16:31:18.534+02:00	END RequestId: 68f53f0b-9bdc-4db7-a6f0-e626c6b7625f
▶	2022-08-25T16:31:18.534+02:00	REPORT RequestId: 68f53f0b-9bdc-4db7-a6f0-e626c6b7625f Duration: 5.44 ms Billed Duration: 6 m...
▶	2022-08-25T16:31:42.991+02:00	START RequestId: 5d556ca8-944e-4317-8ccc-02550c291ff9 Version: \$LATEST
▶	2022-08-25T16:31:42.993+02:00	2022-08-25T14:31:42.993Z 5d556ca8-944e-4317-8ccc-02550c291ff9 INFO EVENT { "Records": [ { "ev..."}
▶	2022-08-25T16:31:42.994+02:00	END RequestId: 5d556ca8-944e-4317-8ccc-02550c291ff9
▶	2022-08-25T16:31:42.994+02:00	REPORT RequestId: 5d556ca8-944e-4317-8ccc-02550c291ff9 Duration: 1.49 ms Billed Duration: 2 m...
▶	2022-08-25T16:31:52.662+02:00	START RequestId: 2ff6c704-35a3-49bf-9f69-b9b779c1dec7 Version: \$LATEST
▶	2022-08-25T16:31:52.664+02:00	2022-08-25T14:31:52.664Z 2ff6c704-35a3-49bf-9f69-b9b779c1dec7 INFO EVENT { "Records": [ { "ev..."}
▶	2022-08-25T16:31:52.665+02:00	END RequestId: 2ff6c704-35a3-49bf-9f69-b9b779c1dec7
▶	2022-08-25T16:31:52.665+02:00	REPORT RequestId: 2ff6c704-35a3-49bf-9f69-b9b779c1dec7 Duration: 1.22 ms Billed Duration: 2 m...

At the bottom of the log table, it says 'No newer events at this moment. Auto retry paused. *Resume*'.

awsfundamentals.com

These events contain a timestamp and the actual message. You can see the start and the end of this Lambda execution indicated by **START** and **END**.

## Log Streams Contain One or More Log Events

A log stream is a bucket for all log events.

The screenshot shows the AWS CloudWatch Log Streams interface. At the top, there are tabs for 'Log streams' (which is selected), 'Metric filters', 'Subscription filters', 'Contributor Insights', and 'Tags'. Below the tabs, a search bar contains the placeholder 'Filter log streams or try prefix search'. To the right of the search bar are buttons for 'Delete', 'Create log stream', and 'Search all log streams'. A dropdown menu is open, showing the option 'Last event time'. The main table lists three log streams:

Log stream	Last event time
2022/08/22/[LATEST]ba18532b513d4f98bfe7125f5b3b4816	2022-08-22 13:13:40 (UTC+02:00)
2022/08/21/[LATEST]d01bafeabc684ed2acc539154dd5c20e	2022-08-21 17:36:05 (UTC+02:00)
2022/08/21/[LATEST]2b7509fd656d4fd4970a9c3fd519efe8	2022-08-21 17:30:29 (UTC+02:00)

At the bottom left of the interface, the text 'awsfundamentals.com' is visible.

In the case of Lambda, one log stream belongs to one **warm Lambda container**. This means one Lambda container that wasn't destroyed. The Lambda chapter explains cold & warm starts in more detail. The log stream holds all log events.

## Log Groups Hold All Log Streams for One Application or Service

Here are all the different Log Groups:

The screenshot shows the AWS CloudWatch Log Groups interface. At the top, there are tabs for 'Log groups' (which is selected), 'Actions', 'View in Logs Insights', and 'Create log group'. Below the tabs, a search bar contains the placeholder 'Filter log groups or try prefix search'. To the right of the search bar are buttons for 'Exact match', 'Actions', and 'Contributor Insights'. A dropdown menu is open, showing the option 'Contributor Insights'. The main table lists eight log groups:

Log group	Retention	Metric filters	Contributor Insights
/aws/lambda/dynamoDbStreamUser	Never expire	-	-
/aws/lambda/thumbail	Never expire	-	-
/aws/lambda/us-east-1.edge-viewer-request	Never expire	-	-
/aws/lambda/us-east-1.prod-awsfundamentals-sst--L...	Never expire	-	-
/aws/lambda/us-east-1.prod-awsfundamentals-sst--L...	Never expire	-	-
/aws/lambda/us-east-1.prod-awsfundamentals-sst--L...	Never expire	-	-
/aws/lambda/us-east-1.prod-awsfundamentals-sst--L...	Never expire	-	-

At the bottom left of the interface, the text 'awsfundamentals.com' is visible.

The name of a log group is prefixed with `/aws` and the service name. For Lambda this is:

`/aws/lambda/<FUNCTION_NAME>`.

Each Log Group contains different Log Streams. The retention settings are associated with the log group. You can define how long your logs are stored.

These are the basics of CloudWatch Logs. Often you can jump into the correct log group by going to your service → heading over to the monitoring tab → clicking on open logs in CloudWatch.

### **CloudWatch Log Insights Allows You to Query Logs like a Database**

CloudWatch Log Insights is an amazing product on top of CloudWatch Logs. It allows you to query your logs with a SQL-like query language.

Tracing logs across different log groups can be quite challenging. Log Insights helps you with that.

In a real production environment, you will have **a lot of logs**. Your logs should contain some sort of correlation id. This can be an autogenerated id that is attached to every log. Or it can be an id from your business context like a user id.

Logs Insights allows you to query multiple log groups with statements like this:

```
fields @timestamp, @message
| filter userId = "user_1"
| sort @timestamp desc
| limit 20
```

The query looks at a first glance similar to SQL.

You describe what you want, add filters, and sort arguments. The engine (logs insights) gives you the results.

For this query, we want to see the `timestamp` and the `message` for the user `user_1`.

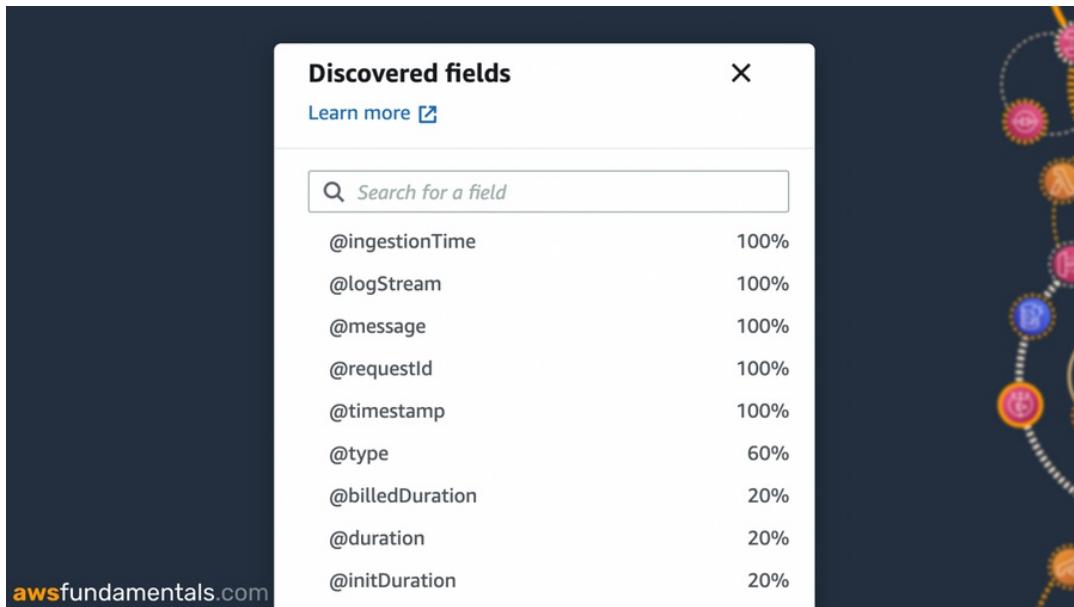
### **Log Insights Shows You Which Fields Are Available for Your Logs**

Most of the time you will query multiple log groups by certain fields. But how do you know which fields are available in the log groups?

Log Insights shows you a recommendation of which fields are available to query.

**You need to execute one sample query first.** Log Insights shows you then all the available fields with the percentage of how often they were available. For example, in 60% of all

logs, a field name `@type` was available.



The screenshot shows the "Discovered fields" interface in AWS CloudWatch Metrics Insights. It lists various fields and their discovery percentages:

Field	Percentage
<code>@ingestionTime</code>	100%
<code>@logStream</code>	100%
<code>@message</code>	100%
<code>@requestId</code>	100%
<code>@timestamp</code>	100%
<code>@type</code>	60%
<code>@billedDuration</code>	20%
<code>@duration</code>	20%
<code>@initDuration</code>	20%

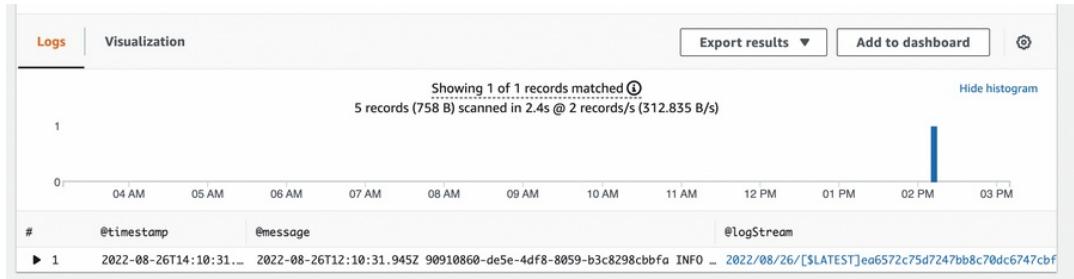
### You Can Build Queries with `fields`, `filters`, `sort`, `limit`, and Many More

Let's build a quick query that filters on the field `correlationId`

```
fields @timestamp, @message
| filter correlationId like "f96eea7e"
| sort @timestamp desc
| limit 20
```

What is happening here?

- `fields` select the fields you want to display → timestamp and the actual message.
- `filter` filters all requests based on the expression that follows. In that case, we filter for a correlation ID.
- `sort` sorts based on the timestamp.
- `limit` limits all requests to 20.



The final result looks like the image above. We have one matching event. By selecting the `@logstream` field we have a direct link to the log stream. This only works if you select **one log group**.

### Log Insights Lets You Query across Multiple Log Groups

The great thing about Log Insights is that you can query across multiple log groups.

For example, if you want to see logs for one particular user. You can select multiple log groups and filter for the `userId`.

Log Insights goes through all selected log groups and shows you the logs. You can choose all log groups in the dropdown field above the editor.

**Logs Insights**  
Select log groups, and then run a query or [choose a sample query](#).

Select log group(s)

/aws/lambda/dynamoDbStreamUser X /aws/lambda/thumbnaill X

Show fewer chosen log groups

```
1 fields @timestamp, @message
2 | sort @timestamp desc
3 | limit 20
```

[awsfundamentals.com](#)

### Log Insights Has Pre-Made Queries Available

Log Insights also has some pre-made queries. These queries are available on the right side. You find examples of different AWS Services like Lambda or ECS.

This query shows you a report of how much memory you over-provisioned in your Lambda functions.

```
filter @type = "REPORT"
| stats max(@memorySize / 1024 / 1024) as provisionedMemoryMB,
  min(@maxMemoryUsed / 1024 / 1024) as smallestMemoryRequestMB,
  avg(@maxMemoryUsed / 1024 / 1024) as avgMemoryUsedMB,
  max(@maxMemoryUsed / 1024 / 1024) as maxMemoryUsedMB,
  provisionedMemoryMB - maxMemoryUsedMB as overProvisionedMB
```

provisionedMemoryMB	smallestMemoryRequestMB	avgMemoryUsedMB	maxMemoryUsedMB	overProvisionedMB
122.0703	52.4521	52.4521	52.4521	69.6182

As you can see it is possible to even create more complex reports. You can use basic aggregations like max, min, or average.

We won't dive deeper into Log Insights right now. But if you want to understand & trace your requests properly, we'd definitely recommend using this service a lot to get better at it.

These were the basics of CloudWatch Logs. Logs are an **essential** part of any cloud application. CloudWatch gives you the capability of understanding your distributed application by consolidating all logs in one place.

## CloudWatch Metrics Stores Metrics about Your Services

Next, we look at the second basic functionality of **CloudWatch, Metrics**.

CloudWatch acts as a metric repository. Each application sends default metrics to CloudWatch. You can use CloudWatch to understand how your application behaves.

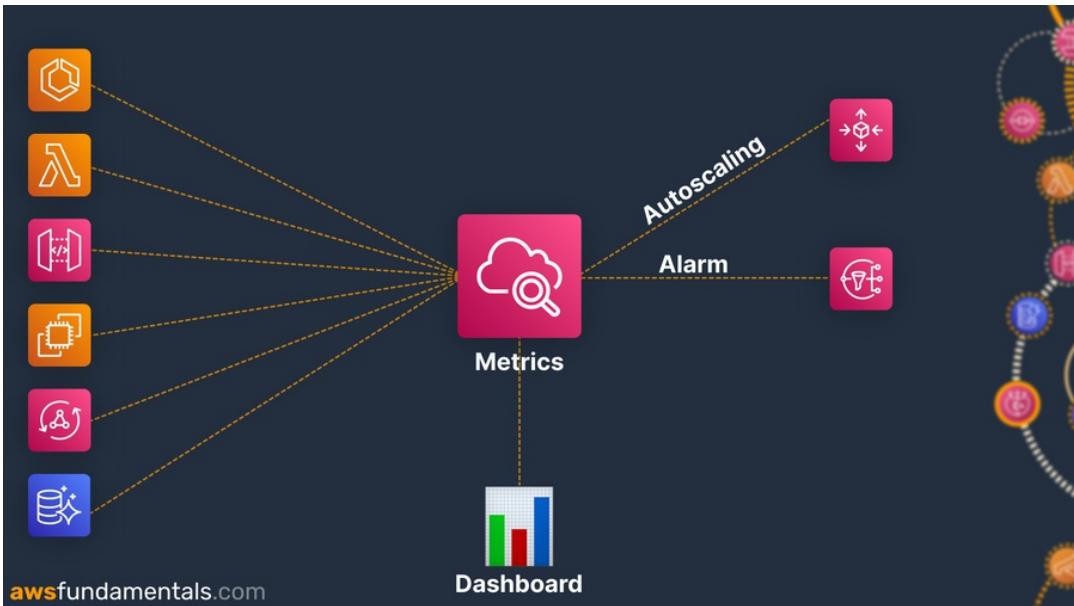
Let's use the Lambda service as an example again. Example metrics are:

- Number of invocations
- Execution time of the Lambda
- Number of errors

You can also use statistical functions like averages, sums, or medians. You can create a dashboard to present these metrics to your stakeholders.

Based on the metrics you can also create **alarms**. Alarms notify you if a metric meets a pre-defined threshold. CloudWatch Metrics lets you even **scale your resources**. For example, if

an EC2 instance is almost out of memory you can scale it with CloudWatch.

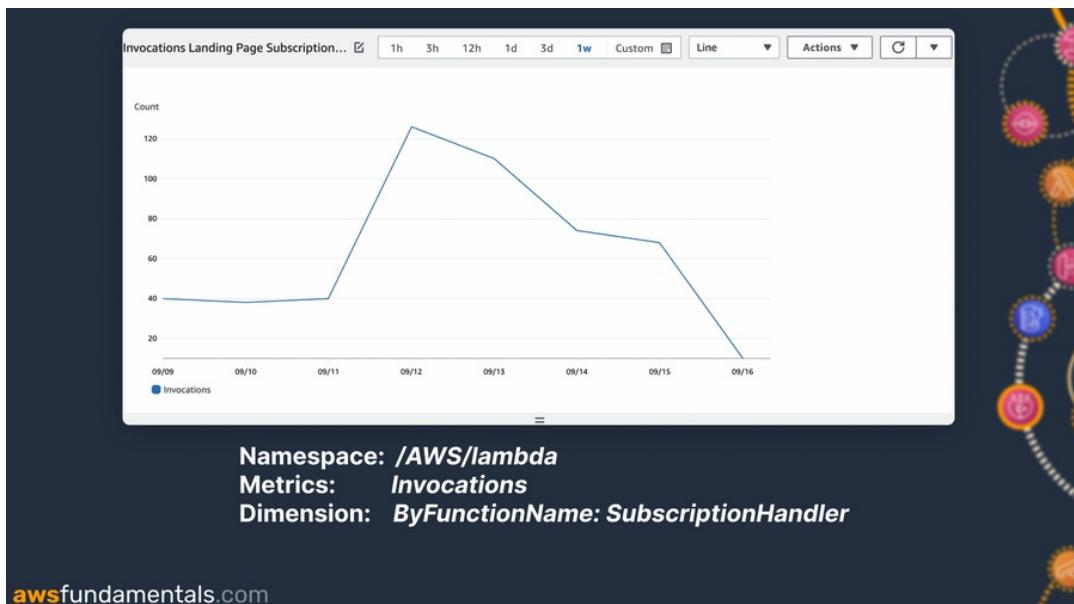


This image shows you how CloudWatch metrics work. You have several services that put metrics into CloudWatch metrics. A CloudWatch alarm either uses SNS or AutoScaling as an action. SNS for notifying you and your colleagues. AutoScaling for scaling resources up and down.

#### **Namespaces, Metrics, Dimensions, Resolutions, and Statistics**

Name	Definition
<b>Namespaces</b>	A namespace is like a bucket for CloudWatch metrics. The same namespaces belong to each other and different namespaces don't. Typically the AWS namespace follows the convention <code>AWS/service</code> for example <code>AWS/EC2</code> or <code>AWS/Lambda</code>
<b>Metrics</b>	A metric is <b>time-ordered data</b> within CloudWatch. It represents a set of data points. A metric has a time and a data point attached. If you think of the AWS Lambda service you can have the <b>23rd of August as a Date</b> and the <b>number of errors (0) as the data point</b> .
<b>Dimensions</b>	Dimensions are name/value pairs for the identity of a metric. Dimensions make it easier to understand metrics. For Lambda these dimensions can be for example <b>by function name</b> or <b>across all functions</b> . It is a way of grouping metrics together.

Name	Definition
<b>Resolution</b>	Metrics can have different resolutions. Resolutions are the <b>granularity</b> of the metrics. The standard resolution has a granularity <b>of one minute</b> . High Resolution has a granularity <b>of one second</b> . You often need to <b>pay extra charges</b> to get the high resolution.
<b>Statistics</b>	Statistics are <b>aggregations of the metrics over a specified time</b> . For example, a sum of errors overall Lambda functions.



The image above shows an example of a Lambda metric.

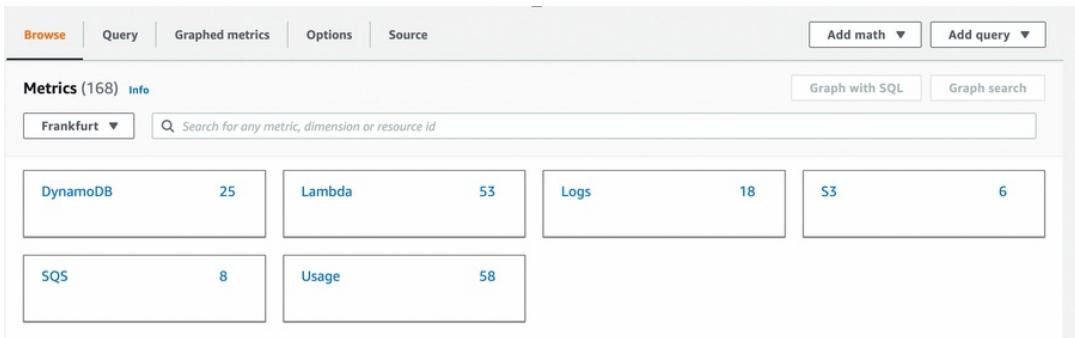
- Namespace: /AWS/lambda
- Metrics: Invocations
- Dimension: byFunctionName: SubscriptionHandler

It displays the count of invocations over one week per day.

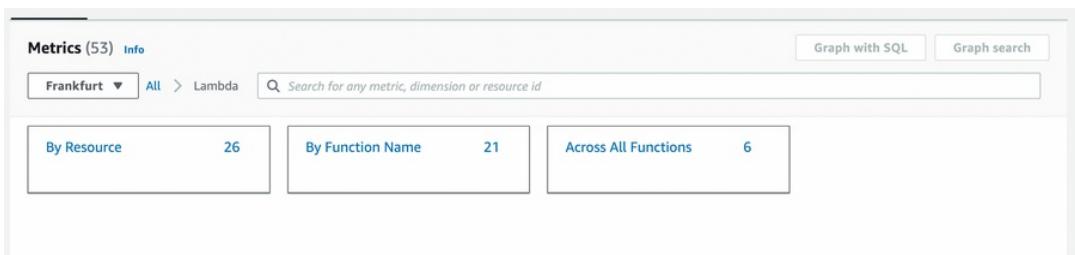
#### Create Metrics with the CloudWatch Browser

The CloudWatch console gives you several opportunities to look at your metrics.

Let's see an example. Click on CloudWatch Metrics and you'll see this window:



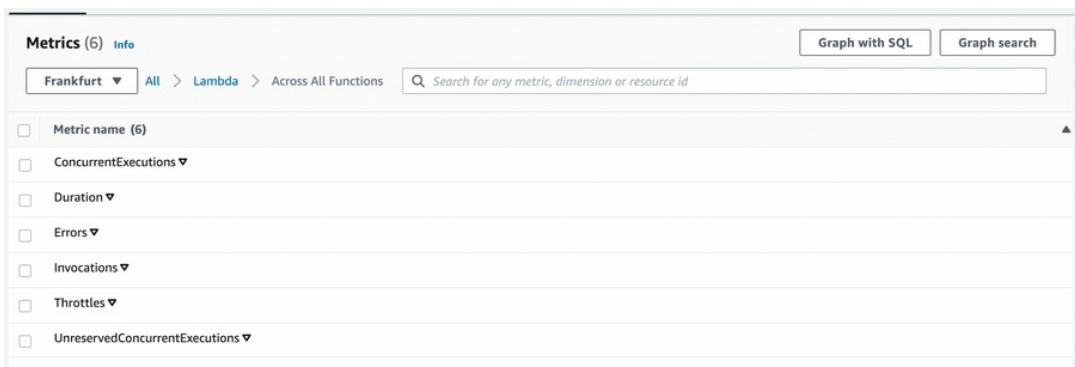
In the **Browse** tab, you can see all the different services that have metrics available. You can click on the service (e.g. Lambda) to see all dimensions.



Dimensions here are

- By Resource
- By Function Name
- Across All Functions

We choose **Across All Functions** because we want to see metrics across all our Lambda functions.



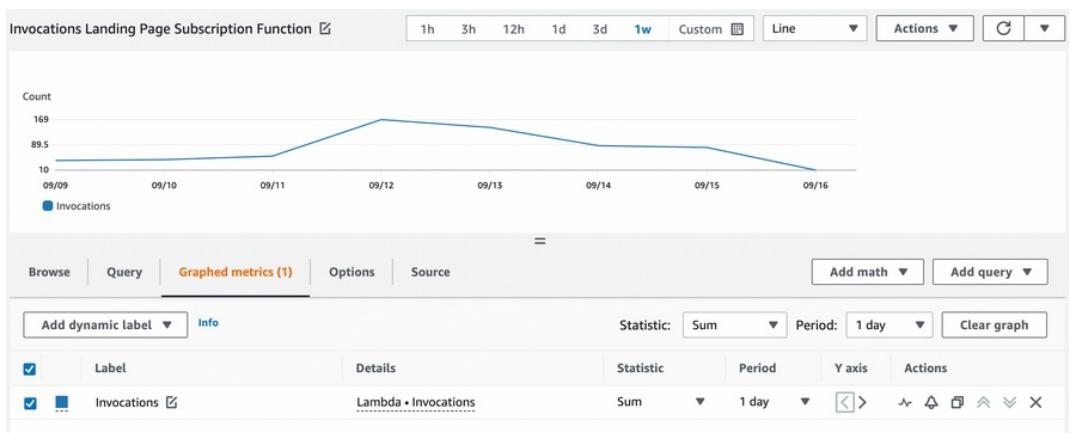
Now, you see the different metrics available for Lambda. These are

- **ConcurrentExecutions:** How many Lambda functions run at the same time

- **Duration:** How long did the functions run
- **Errors:** How many errors happened
- **Invocations:** How often was the Lambda executed

And so on.

By checking the checkbox of a metric it will appear in the graph on the top of the screen. It will also be added to the **Graphed Metrics** tab.



You can then go ahead and adjust the following things:

- The timeframe on the top of the page
- The statistic function (sum, average, min, max, etc.)
- The period on the top of the screen
- Rename labels, axes, and titles
- Changing the type of the chart (line, bar, number, etc.)

With that, you can get a powerful overview of all of your services.

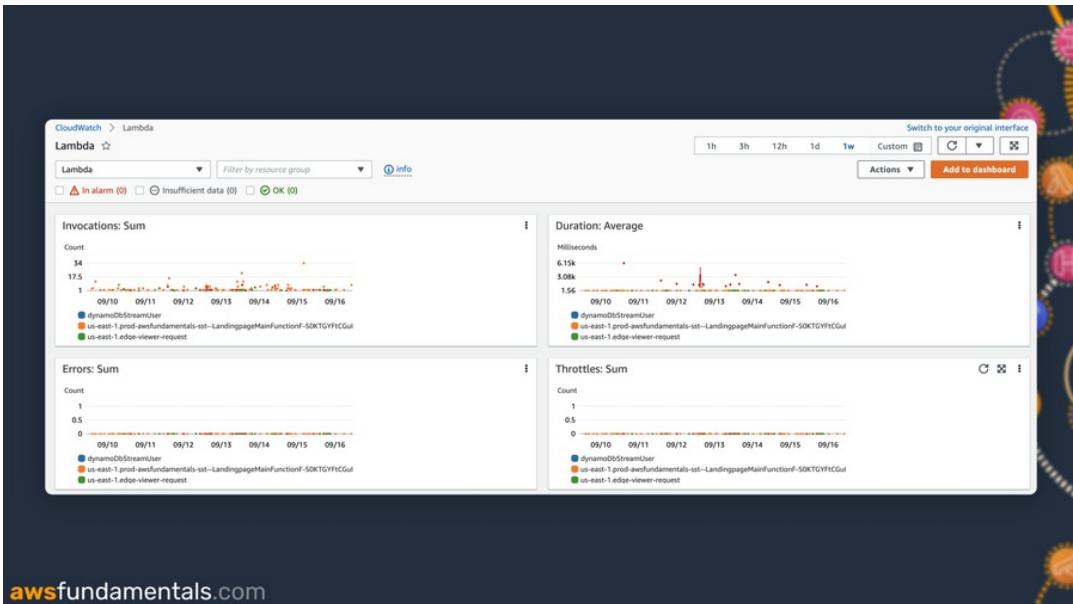
### **Dashboards Give You an Overview of Metrics and Alarms**

Dashboards give you the opportunity to get an overview of lots of different metrics and alarms at the same time.

CloudWatch creates automatic dashboards for you. You can see them if you open CloudWatch

and head to the tab **Automatic dashboards**. They are available for different services like DynamoDB, Lambda, and CloudWatch (itself). But you can also create custom ones.

Let's have a look at the automatic dashboard of the Lambda service:



Dashboards make it much easier to understand how your system behaves. You can also share dashboards across your entire organization. With that everybody gets fast access to metrics.

Dashboards are similar to creating the graphs above for metrics. A dashboard contains several of these graphs.

Dashboards cost 1\$ per month after the first one, meaning **one is free** in the free tier.

### CloudWatch Alarms Notify You on Pre-Defined Thresholds like CPU Usage or Errors

Let's see one common bad scenario. Your server goes down. You only find out because customers are calling you. This is often pretty typical for companies with bad monitoring systems. You want to get informed about downtimes **before** your customer knows about them. CloudWatch Alarms help you with that.

CloudWatch Alarms notify you once a system or service reaches a pre-defined threshold like errors or usage.

It is important to spend time defining your alarm thresholds. By having the correct ones you will know when something bad happens. Even before your customer knows.

Good examples of alarms are:

- Messages available in your Dead Letter Queue
- Errors in your Lambda function
- API is throwing more HTTP 500 status codes than usual

### **The Different Alarms States**

There are two alarm states:

- `IN_ALARM`: The alarm is active → You need to do something
- `OK`: The alarm is not active → All good

It is also possible that the alarm doesn't have sufficient data. You can define if this will trigger the alarm or not. Normally it won't trigger an alarm.

### **Creating Alarms Is Highly Dependent on Your Business Logic**

Creating alarms is an art in itself since it depends on your business logic a lot. But there are some best practices to follow when creating alarms. It is especially important to always keep your limits in mind. Your AWS Account has different levels of services. There are:

1. Service limits
2. Regional limits
3. Account limits

For example, RDS can only have 40 instances per region. Lambda can only have 1,000 Lambdas running at the same time per region. To not run into any issues you need to keep this in mind when setting up your alarms.

Some ideas for alarms:

1. CPU utilization for ECS tasks
2. Visible messages in your DLQ
3. Error in Lambda
4. Number of 500 responses in API Gateway

5. Latency too high in API Gateway
  6. Concurrent Lambda executions

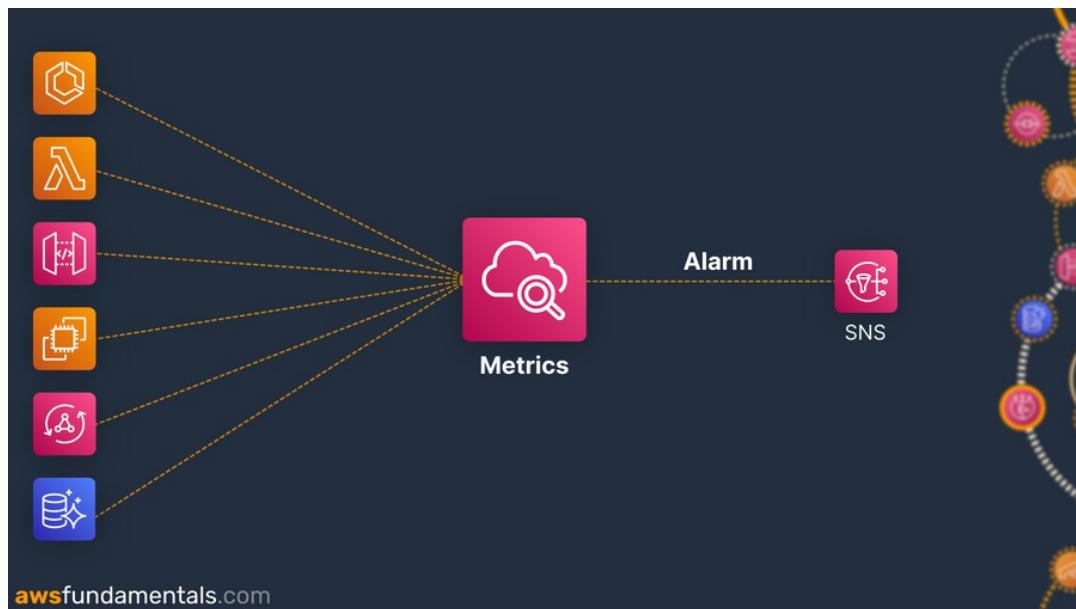
Also, create alarms around limits. One good alarm for Lambda is the number of concurrent invocations. If you work with CDK there are some pretty good constructs out there that help you with that. Check out the monitoring and watchful constructs

## Metric Alarms Are Triggered on One Metric - Composite Alarms Combine Several Alarms Together

There are mainly two different types of alarms.

Type	Description
<b>Metric Alarm</b>	This is an alarm based on one metric.
<b>Composite Alarm</b>	This alarm takes several alarms into account and their states. For example, you can build an alarm that will only be triggered if two of the three metric alarms are in the state <code>IN_ALARM</code>

### **SNS Topics Inform You in Case of Alarms**



The service Simple Notification Service (SNS) was covered in its own chapter.

It allows you to send personalized emails, SMS, or In-App Notifications in case of alarms. CloudWatch uses SNS to inform you of alarm changes. CloudWatch has really good integration with the E-Mail component of SNS. You can add your email address and get notified of alarms.

Since SNS is really flexible, you can also attach a Lambda function to the SNS topic. The Lambda function can do multiple things to alarm you. Some standard use cases are sending notifications via Slack, MS Teams, or Discord. You can also use PagerDuty to get informed about changes.

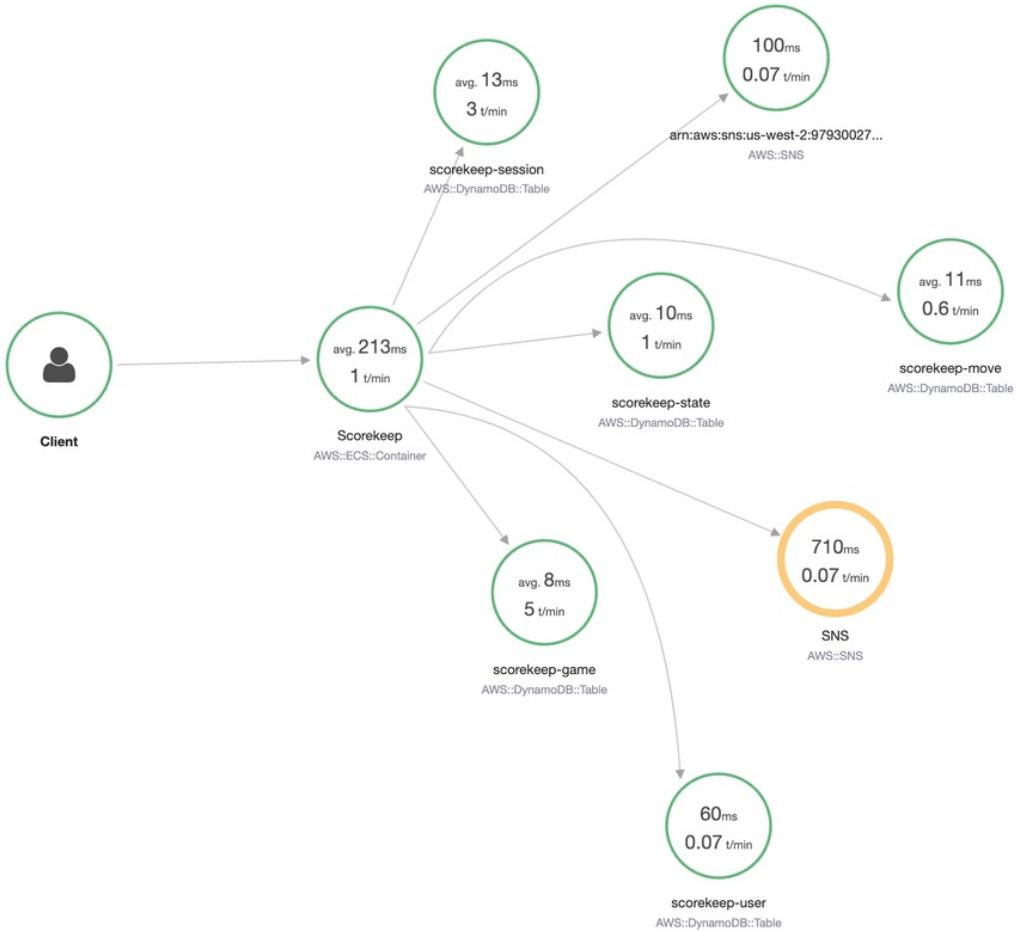
### **X-Ray Gives You the Ability to Trace User Requests Throughout Your Distributed System**

As you can see throughout the book there are many AWS Services that do several things. Understanding a single user request can **get quite challenging**.

For example, if a user reports an issue you need to understand the **whole flow** the user took. That means looking at multiple AWS Services.

X-Ray gives you the opportunity to build distributed traces for all requests. That means each user request will have an `x-ray-trace-id` which you can follow in the system. You then see how users interacted with your system and can also check all logs attached to this trace.

Let's have a look at an example service map that X-Ray built for us:



This example from the AWS Developer Documentation shows you how a client calls ECS and how ECS makes several calls to SNS & DynamoDB. You can see the latency of the requests and drill deeper into them.

X-Ray also saves all of these traces in a table. You can check them one by one to understand the requests further. This is really helpful if you have to debug user sessions. For pure serverless systems, it is quite hard to build up a complete service map. Some services are not fully supported. For example, the integration between SQS and Lambda. By that X-Ray is losing the vision of these services. It still often makes sense to activate X-Ray to understand user requests better. With the upcoming services in Open Telemetry, it can only get better.

## **CloudWatch Synthetics Lets You Test Web Applications in Regular Intervals**

CloudWatch Synthetics is a service that creates so-called canaries. Canaries are scripts that run on a schedule. Their goal is to check endpoints and APIs. They are used for regular health checks of your web applications. The synthetics service falls into the category of **Application Monitoring**. It supports using programmatic web browsers like Puppeteer.

A canary can be of the type

- **Heartbeat monitor** - check URLs regularly
- **API Canary** - check API endpoints
- **Broken Link Checker** - checks broken links
- **Visual Monitoring** - open a webpage in the browser and check elements on that page like buttons
- **Canary recorder** - record sessions in your browser and replace them
- **GUI Workflow builder** - verifies that actions can be done on your web page

CloudWatch Synthetics is a great tool for integrating automated testing into your application. Selenium and Puppeteer are both browsers you can control from source code. By using them you can test your whole web application in code. With this service, you can build a fully-fleshed test suite cost-effective in AWS.

## **CloudWatch Is Priced Based on Ingest and Storage**

CloudWatch can be quite expensive. We said earlier that CloudWatch is one of the most underrated services. This is not only true for learning this service but also for estimating application costs.

Ingesting logs into CloudWatch can get really expensive. Let's see how much logs are costing:

What	How much?
<b>Ingest Data</b>	\$0.50 / GB
<b>Store</b>	\$0.03 / GB
<b>Analyze</b>	\$0.005 / GB scanned

The most expensive point here is **ingesting data**. Storing data can be reduced by setting up a log retention policy. But ingesting logs can only be done from a business logic standpoint. Make sure you understand **which data you need** to log and **which data you don't need to log**. Also don't automatically activate `DEBUG` logs by default. Sample debug logs (like 1%) and deactivate them after.

#### **Standard Metrics Are Free - Custom Metrics Not**

Metrics are different. Default metrics of services like EC2, Lambda, etc. are stored automatically in a standard granularity (1 minute). You only pay for these metrics if you require a more granular view (1 second).

Custom metrics are **really expensive**.

Tier	Cost metric / month
<b>0 - 10,000</b>	\$0.30
<b>Next 240,000</b>	\$0.10
<b>Next 750,000</b>	\$0.05
<b>&gt; 1,000,000</b>	\$0.02

If you have any application with a high load make sure to understand the pricing first. There are really good examples on the AWS CloudWatch Pricing page that shows you how expensive it can be to collect too many metrics.

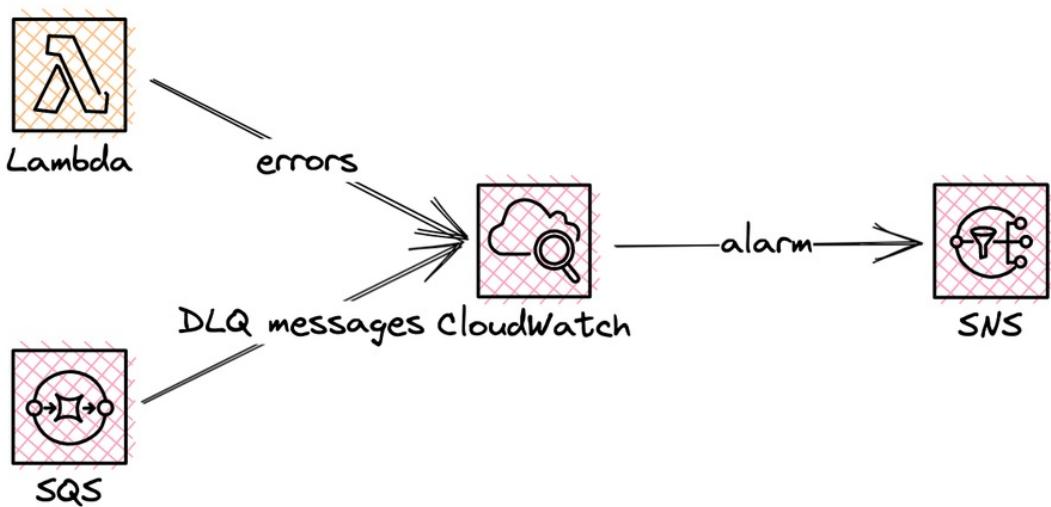
Use the Embedded Metric Log format to get metrics much cheaper by simply logging them to CloudWatch. Check out this link for more details.

#### **Use Cases for CloudWatch**

CloudWatch use cases provide a range of capabilities for monitoring, alerting, and debugging. Let's see three example use cases.

##### **Use Case 1: Alerting in Case of Errors in Your Application**

One of the key use cases for CloudWatch is alerting in case of errors in your applications. All AWS services send metrics to CloudWatch. You can use these metrics to define alarms.

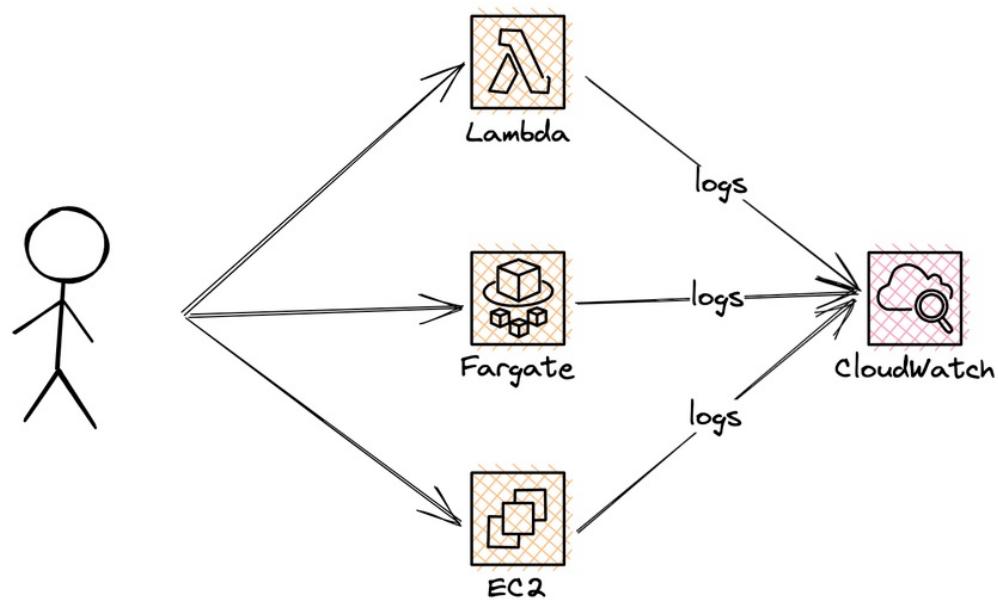


For example, you can create a CloudWatch alarm based on Lambda errors and on messages available in a dead letter queue. Once this alarm is set off an SNS topic will be called. This SNS topic informs the developer or operations team of ongoing errors.

#### **Use Case 2: Debugging and Tracing Logs**

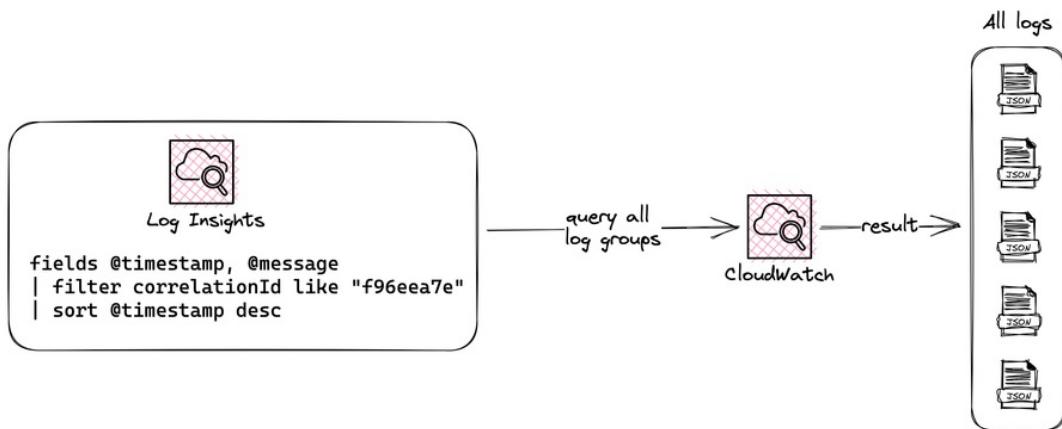
Another useful use case for CloudWatch is debugging and tracing logs across different services. Cloud environments can get quite complex with a variety of services and machines involved.

Users are typically not just using one service but several ones. Debugging a whole user session can be quite challenging.



CloudWatch collects log data from all different sources and persists them in a structured way. This data can be very useful when you try to understand user behavior.

You can use CloudWatch Logs Insights to search, analyze, and visualize log data, making it easier to debug problems and understand user activity.



CloudWatch Insights uses a SQL-like language to query for logs. You can query in multiple log groups. The result is several log statements that match the query. That allows you to see the full picture of a user's interactions with your application.

### Use Case 3: Automate Scaling with CloudWatch Alarms

CloudWatch can also be used to automate the scaling of resources. Typically, this is based on metrics like CPU utilization. But you can use any metric available. Once the metric reaches a certain level you can scale up resources such as ECS or EC2 instances.



This can help you to meet peak loads or to save money.

### Tips for the Real World

Here are some quick tips you can use in the real world:

- Use a **structured logger**. Structured logging allows you to trace and query your logs much more efficiently. Use Lambda Powertools if they are available for your programming language.
- Create alarms and use best practice alarms. There are typical alarms you should create for every service. Examples are the Number of Messages visible in DLQs or errors in Lambda functions.
- Use Log Insights **before incidents happen**. Debugging can be quite challenging. Understand your debugging before an actual incident happens.
- Add **correlation IDs** to your structured logging.
- Create CloudWatch dashboards so that the whole team understands the performance of your application

### Final Words

CloudWatch is the crucial monitoring part that helps to keep your application ecosystem healthy. It's natively integrated with almost any AWS service and gives you deep insights via metrics.

It's important to integrate observability into your ecosystem before incidents start to happen. Finally, you should keep an eye on CloudWatch's costs as it's often a major driver for the end-

of-the-month bill.



**Define & Deploy Your  
Cloud Infrastructure  
with  
Infrastructure-As-Code**



# Define & Deploy Your Cloud Infrastructure with Infrastructure-As-Code

The first part of this book was all about AWS services. We introduced you to the most important ones to build applications.

This second part is all about Infrastructure as Code. There is almost no professional Cloud Developer who doesn't use Infrastructure as Code (IaC) Tools. This is why we start covering this topic in part two.

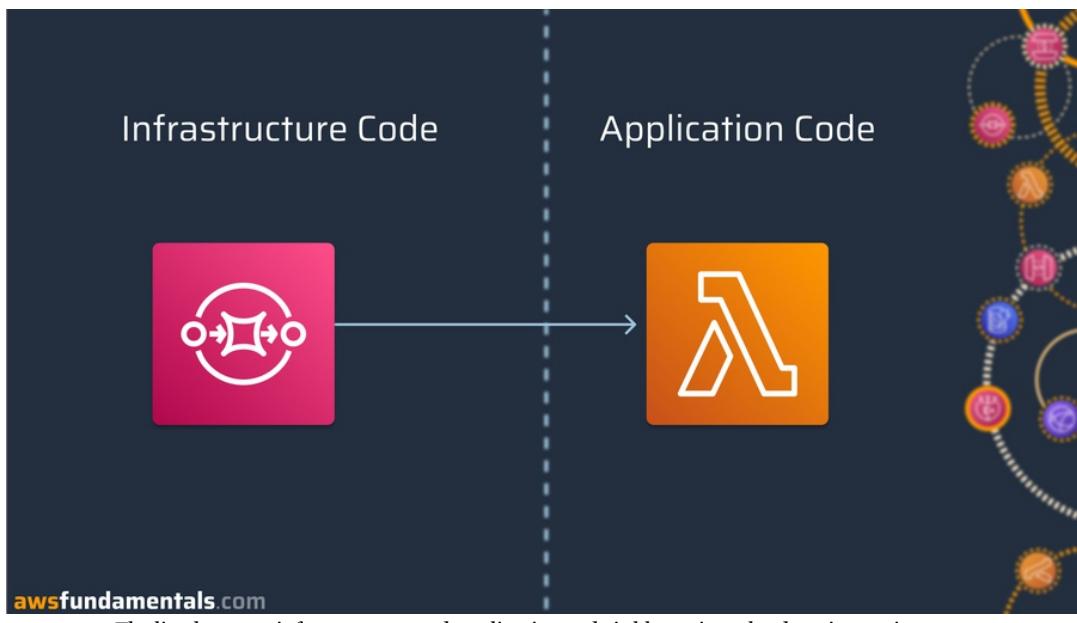
We will cover the basics of IaC and introduce some of the most common tools. But, we're not doing a deep dive on IaC. This book is still about the basics of AWS.

The goal of this part is to teach you the fundamentals that you need to get started. We will introduce the **three common tools** in the industry:

1. CloudFormation
2. Serverless Framework
3. Cloud Development Kit (CDK)

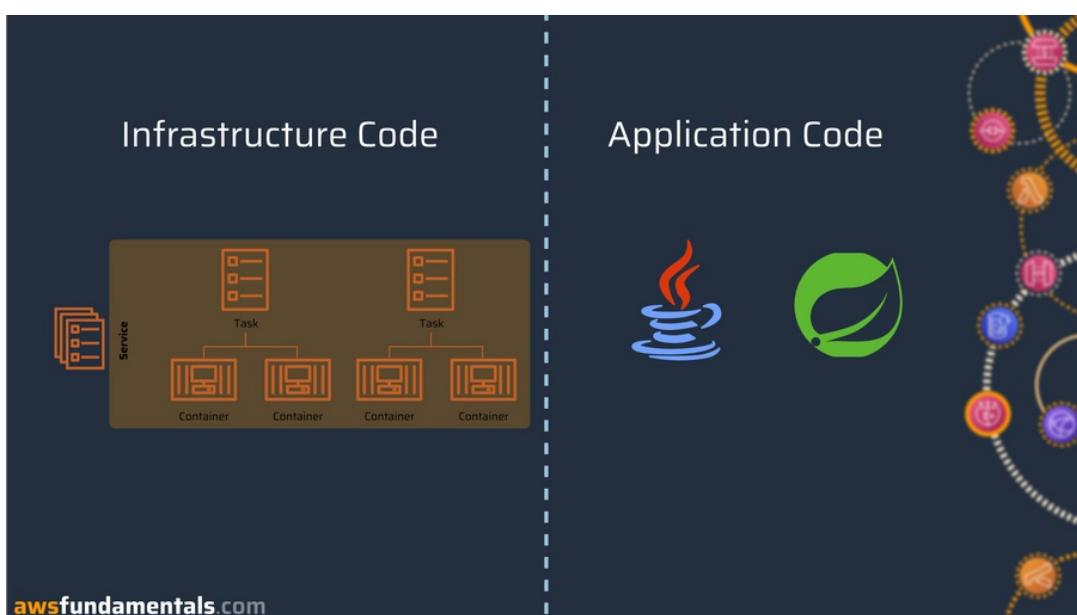
## Application Code Is Infrastructure Code

One difference that comes with cloud computing is application vs. infrastructure code. This line can get **blurry in a cloud-native world**.



Let's take SQS and Lambda as an example. We cannot distinguish one service into only infrastructure or only application code. AWS manages the underlying infrastructure for both services. We use the API of these services to build our applications.

In earlier days the separation was easy. Your Virtual Machines were your infrastructure. And your application server (e.g. Java Spring) was your application code. You could have two different code bases for that.



Now the interaction across infrastructure and application is much more connected.

## What Is Infrastructure as Code?

Infrastructure as Code describes the practice of provisioning your infrastructure with source code. By having source code you have many benefits:

- You can duplicate the same environment in another account or region
- Every change is documented and versioned.
- Infrastructure as Code enables you to follow DevOps best practices. Automated testing, continuous deployment, feature flagging. All of that is possible
- You get a reproducible setup. There are no manual checklists or steps to remember. Executing the code is sufficient.

There are tons of more benefits. You won't use AWS professionally without being able to provision resources via Code.

## History of IaC

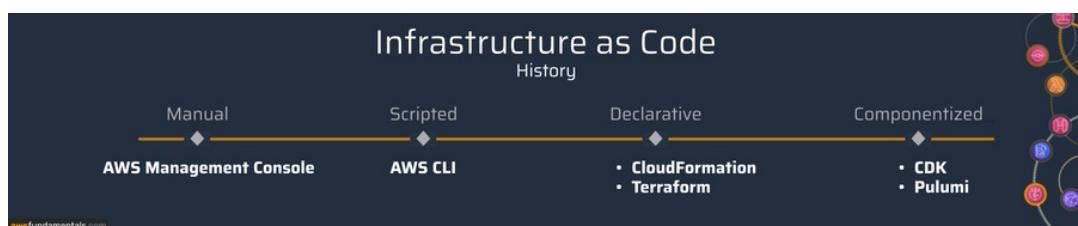
*If you want to understand today you have to search yesterday.*

**Pearl S. Buck, American novelist (1892-1973)**

There are many IaC tools out there. All (or most) of them have their right to be there. To understand their differences we need to take a step back and understand where we are coming from. Each step in the timeline had a **different philosophy**.

Remember that cloud computing is just evolving for the last two centuries. Software development changed many times over the last couple of years.

The image below shows you some of the main categories of different IaC toolings.



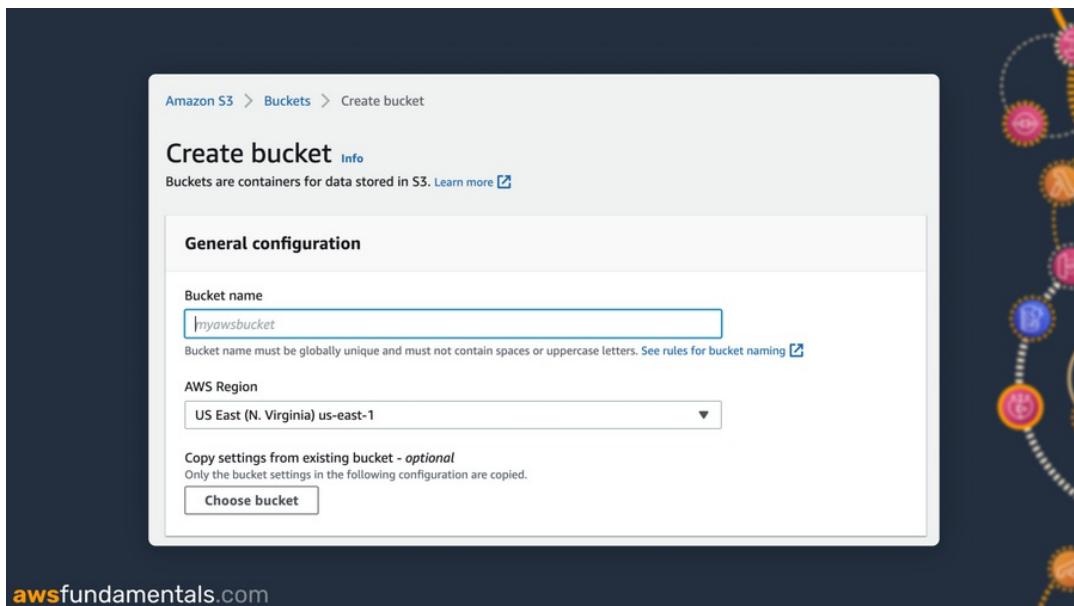
We started with a manual approach (clicking in the console). And now we are in a state where

we can use our favorite programming language with frameworks like CDK.

Let's take a look at each category in more detail.

### Manual - Clicking in the AWS Management Console

Manual provisioning was the start. Back when AWS only had a couple of different services it was easy to do it manually. You logged into the console and created your infrastructure.



You didn't have any code, any history, or any way to do governance on your architecture. Working with each other meant creating checklists and screenshots. Only with those, it was possible to reproduce the same setup.

After AWS launched many more services this was not a workable approach anymore. Launching new regions or development accounts was almost impossible.

### Scripted - Provision AWS Resources with the CLI

The second step in history was creating scripts using the AWS CLI. The CLI is one of the main interfaces to the AWS API. You can create an S3 bucket for example with the CLI like that:

```
aws s3api create-bucket --bucket awsfundamentalstestbucket
```

Many people started using the CLI in automated script files.

The issue with the CLI is that there is no rollback logic or any concrete state. Updating

infrastructure was hard. You needed to write a logic for existing infrastructure as well. This was not a good approach as well.

The main benefit was that there was a semi-automated way of **creating infrastructure**. But not updating any infrastructure

### Declarative - Describe the Infrastructure You Need

The next step was using a **declarative method**. A declarative approach defines **what the final state looks like**. We don't care **how** it will be done, we only define the final state.

For example, we can define that we need an S3 bucket. How the tool takes care of providing us with that bucket is not of interest.

AWS introduced the service **CloudFormation**. CloudFormation allows you to provision resources, handle errors, and roll back states. A CloudFormation template is a configuration file in a YAML or JSON format. For example, a file could look like that:

```
{
  "AWSTemplateFormatVersion": "2010-09-09",
  "Description": "Sample CloudFormation Template",
  "Resources": {
    "s3Bucket": {
      "Type": "AWS::S3::Bucket",
      "Properties": {
        "BucketName": "awsfundamentals-testbucket-xyz"
      }
    }
  },
  "Outputs": {
    "BucketName": {
      "Value": {
        "Fn::GetAtt": ["s3Bucket", "Arn"]
      }
    }
  }
}
```

CloudFormation knows which infrastructure is already provisioned. It then goes ahead and only adds new infrastructure.

The underlying CloudFormation engine made a huge difference from that time. Developers are still using CloudFormation a lot. Especially corporations that started out with CloudFormation are still using it a lot. There are also many frameworks that build on top of CloudFormation. Examples here are CDK, Sam, or Amplify.

Terraform is also in this category. Terraform went a different way and is using AWS APIs directly and saving their own state. Terraform is not using CloudFormation underneath.

If you want to understand the basics of IaC you need to learn CloudFormation or Terraform. Even if you want to use a framework like CDK. It is needed to understand the basics of CloudFormation.

These two are right now one of the most used IaC tools out there.

### **Benefits:**

- History of code changes
- Reproducible infrastructure deployment
- Build new environments easily

### **Cons:**

- Learning new Syntax
- Huge YAML and JSON configuration files
- Not easy to debug
- No proper software engineering constructs could be used or just in complex ways

### **Componentized - Use Your Programming Language to Build Abstractions**

The final stage is where we are today. The current stage is the **componentized stage**. Componentized refers to building reusable abstractions that developers can reuse. This is something a software developer is doing on a daily basis.

There is one big difference between the other stages. Now developers are able to build infrastructure in **a proper programming language**. Languages like Python, TypeScript, or Java are often supported. Under the hood often CloudFormation is still used.

Popular frameworks in that space are the Cloud Development Kit (CDK) and Pulumi. The good thing is that these tools still use CloudFormation underneath. That means things like drift detection, state updates, or rollbacks are still working.

Using a proper programming language is one of the major benefits here. Developers use these languages and IDEs (Integrated Developer Environments) daily. IDEs allow you to do typical tasks like debugging or refactoring. These tasks are similar to "normal" software engineering.

Another great benefit is that the creation of resources is still declarative. In CDK for example, you only tell which resource you would like to have, e.g. an S3 Bucket. CDK (or CloudFormation) figures out how to create this resource.

Creating a bucket in CDK looks like that:

```
new s3.Bucket(this, 'MyFirstBucket', {});
```

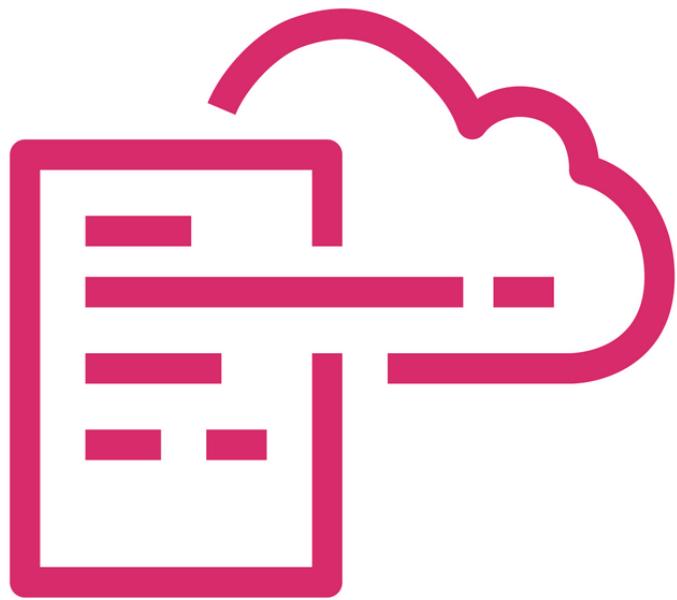
## Benefits

- Proper programming languages can be used
- Building & Sharing abstractions is encouraged and possible
- Using your IDE tools (refactoring, tests, etc.) is possible
- CloudFormation is still underneath → Declarative approach

## Cons

- You can build too many abstractions
- Using it without understanding CloudFormation can be an issue
- You can build systems that rely on some external state like a REST API

After understanding what IaC is, why we need it, and seeing some of its histories let's dive into the first set of tools.



## AWS **CloudFormation**

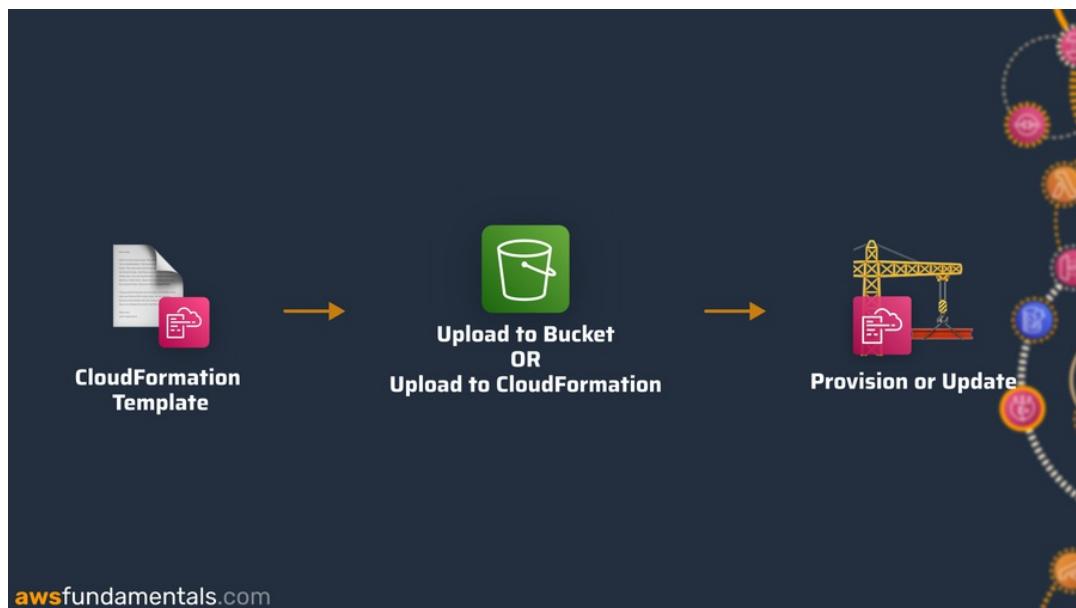
# CloudFormation Is the Underlying Service for Provisioning Your Infrastructure

## Introduction

CloudFormation is not only a tool for provisioning infrastructure via code. It is also the engine that powers AWS to provision cloud services.

CloudFormation templates are templates written in JSON or YAML language. CloudFormation uses these templates to provision your infrastructure. CloudFormation takes care of creating or updating your infrastructure as you defined it.

CloudFormation follows a declarative approach. That means you only tell CloudFormation which resource you want. It finds ways of creating the services and infrastructure. You don't need to define the way how to create it you only define the final state you want to have.



You can pass the templates to CloudFormation via an S3 bucket or upload them to the service. You will see an overview of all the services that CloudFormation creates. After that, it starts provisioning it.

The great thing about CloudFormation is that your infrastructure management got much easier. Let's say you've got this example infrastructure of a web application:

1. Backend Database with **DynamoDB**

## 2. API Server with **API Gateway and Lambda**

### 3. Single Page Application in **S3 Bucket and CloudFront Distribution**

You can create a CloudFormation Template for that and launch it. If you don't need it anymore you can remove all services with one click on CloudFormation.

It is also possible to replicate the whole architecture easily. You can use the same stack and deploy a development or staging environment.

## **CloudFormation Concepts - Templates, Stacks, Change Sets**

CloudFormation follows the concepts of Templates, Stacks, and Change Sets.

### **Templates Define All Resources CloudFormation Should Provision**

The CloudFormation template is a JSON or YAML file. It doesn't matter which file format you use. A template is the main building block of a CloudFormation app.

Let's start with an example.

The following template creates an S3 bucket with the name `awsfundamentalstestbucket-xyz`. After creating the bucket it outputs the ARN of the bucket.

We will explain this in more detail and deploy it to an S3 account in the next chapter.

```
{
  "AWSTemplateFormatVersion": "2010-09-09",
  "Description": "Sample CloudFormation Template",
  "Resources": {
    "s3Bucket": {
      "Type": "AWS::S3::Bucket",
      "Properties": {
        "BucketName": "awsfundamentalstestbucket-xyz"
      }
    }
  },
  "Outputs": {
    "BucketArn": {
      "Value": {
        "Ref": "s3Bucket"
      }
    }
  }
}
```

```

        "Value": {
            "Fn::GetAtt": ["s3Bucket", "Arn"]
        }
    }
}
}
}

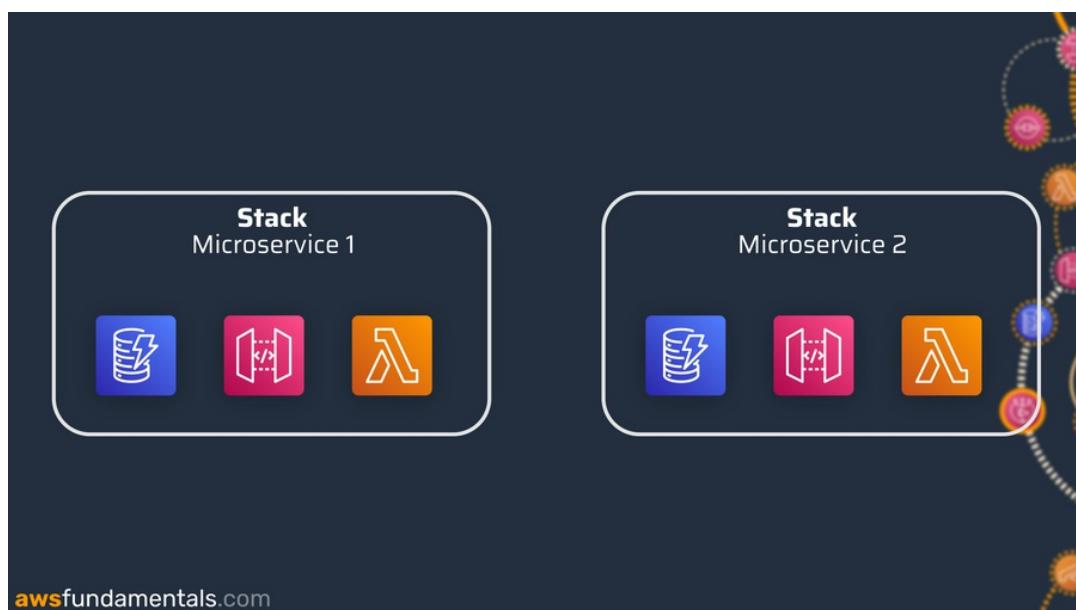
```

## Stacks Are Deployable Units

Another main concept in CloudFormation is the usage of **Stacks**. One stack is a **deployable unit** in CloudFormation. One stack contains a collection of many cloud resources. If we look at our template above one stack would be the S3 bucket. After uploading the template to CloudFormation we give it a name. The service provisions all resources then.

There are different approaches to designing stacks. A common approach is to group one **microservice** into one stack. One microservice could contain the following services:

1. Database
2. API Gateway
3. Lambda Function
4. Frontend



Another approach is to distinguish between **stateful and stateless** resources.

1. **Stateful** - DynamoDB, RDS, SQS

2. **Stateless** - API Gateway, Lambda

It all depends on your architecture. But these are common approaches.

You can do several things with stacks, such as:

- Delete the whole stack
- See the drift in stacks. This shows you the difference between the currently deployed infrastructure and your template.

You can create stacks in different ways. Either in the CLI with `aws cloudformation create-stack` or within the console by clicking on the button **Create stack**. We will create a stack together later on.

### **Change Sets Only Deploy Changes to Your Current Infrastructure**

Change Sets are responsible for changing already deployed infrastructure. CloudFormation detects your changes and only creates changes to your infrastructure. With a change set, you see the exact changes and how they would impact your infrastructure.

A good example is RDS. If you want to activate backups for RDS you don't want to destroy your whole database to activate a backup. With CloudFormation you can create change sets to only update your current infrastructure. Modern frameworks like CDK are doing that but CloudFormation gives them the power to do it. With that, you will activate backups without losing all your data.

### **Define Resources, Outputs, Parameters, and Variables in Templates**

If you work with CloudFormation you will work a lot with templates. Let's go through a template and see some of its basics.

```
{  
    "AWSTemplateFormatVersion": "2010-09-09",  
    "Description": "Sample CloudFormation Template",  
    "Resources": {  
        "s3Bucket": {  
            "Type": "AWS::S3::Bucket",  
            "Properties": {  
                "BucketName": "my-s3-bucket"  
            }  
        }  
    }  
}
```

```

        "BucketName": "awsfundamentalstestbucket-xyz"
    }
}
},
}

"Outputs": {
    "BucketArn": {
        "Value": {
            "Fn::GetAtt": ["s3Bucket", "Arn"]
        }
    }
}
}

```

This is the template we saw earlier already. This template creates an S3 bucket with the name `awsfundamentalstestbucket-xyz`.

## Template Version

`AWSTemplateFormatVersion` this is the version of the CloudFormation template you use. The version is a date when it was released. This is pretty common in the AWS universe and similar to IAM Policies or the usage of SDKs. Try to use the same version across all your CloudFormation templates.

## Description

`Description` is a string that describes this stack. This is really helpful if you work in larger teams.

## Resources Contain All Resources You Want to Provision

```

"Resources": {
    "s3Bucket": {
        "Type": "AWS::S3::Bucket",
        "Properties": {
            "BucketName": "awsfundamentalstestbucket-xyz"
        }
    }
}

```

```
}
```

Resources define the infrastructure that you want to provision. This is the heart of the CloudFormation template.

In that example, we have the key `s3Bucket`. This can be freely chosen.

A resource needs to define the following things:

- `Type` - Types follow a similar pattern like `AWS::ProductIdentifier::ResourceType`. For an S3 bucket, this is `AWS::S3::Bucket`. For an EC2 instance, this is `AWS::EC2::Instance`. You find a list of all resources here
- `Properties` - This object is dependent on the type of infrastructure you want to provision. You need to define the properties the AWS Service offers you. In our case, we only define the `BucketName`

## With Outputs, You Can Print Properties of Your Created Resources

You can define the outputs of properties of your created resource. For example, after creating your S3 bucket you can output the ARN (Amazon Resource Number) of this bucket.

```
"Outputs": {
  "BucketName": {
    "Value": {
      "Fn::GetAtt": ["s3Bucket", "Arn"]
    }
  }
}
```

For other resources like an SQS queue, you can output the queue URL.

Outputs can be imported into other stacks for further usage (like giving IAM permissions) or they can be accessed in the CloudFormation console. Often you need to add them to another application, which is a more convenient way to receive this information.

## Parameters Allow You to Add Information before Deploying Your Stack

For some use cases, it is also important to receive user input **before** deploying the resources.

For example, if you deploy infrastructure to different stages like development & production.

You often want to change some properties for these stages. In the development stage, you want to name your resources with the prefix `Dev-`. In the production stage, you want to add the prefix `Prod-`. Or you want to set different tags.

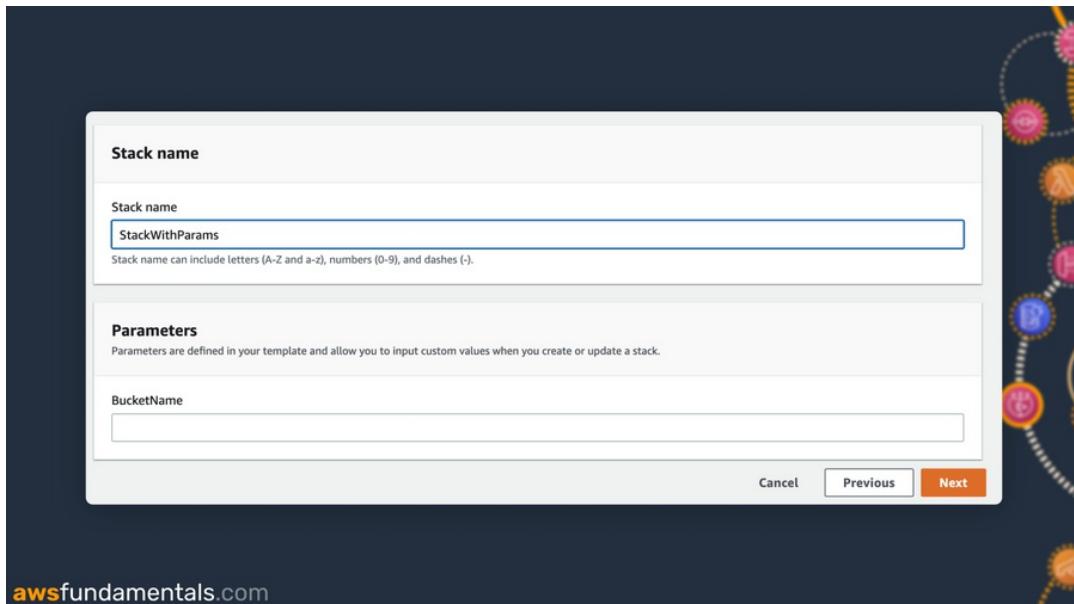
You can use parameters for that. CloudFormation will ask you for this information **before you can deploy your stack**. Let's extend our example stack with a dynamic bucket name. The user should enter the name of the bucket **before** we create the bucket.

For that, we add the block `Parameters` to the template. We also define an input parameter of type `String`. The `Properties` object references the bucket name with the variable `BucketName`.

```
{  
    "AWSTemplateFormatVersion": "2010-09-09",  
    "Description": "Sample CloudFormation Template",  
  
    "Parameters": {  
        "BucketName": {  
            "Type": "String"  
        }  
    },  
  
    "Resources": {  
        "s3Bucket": {  
            "Type": "AWS::S3::Bucket",  
            "Properties": {  
                "BucketName": {  
                    "Ref": "BucketName"  
                }  
            }  
        }  
    },  
  
    "Outputs": {  
        "BucketArn": {  
            "Value": {  
                "Fn::GetAtt": ["s3Bucket", "Arn"]  
            }  
        }  
    }  
}
```

```
    }  
}  
}
```

If we create a stack in CloudFormation now we can see that we need to specify the `BucketName` before we can deploy this stack.

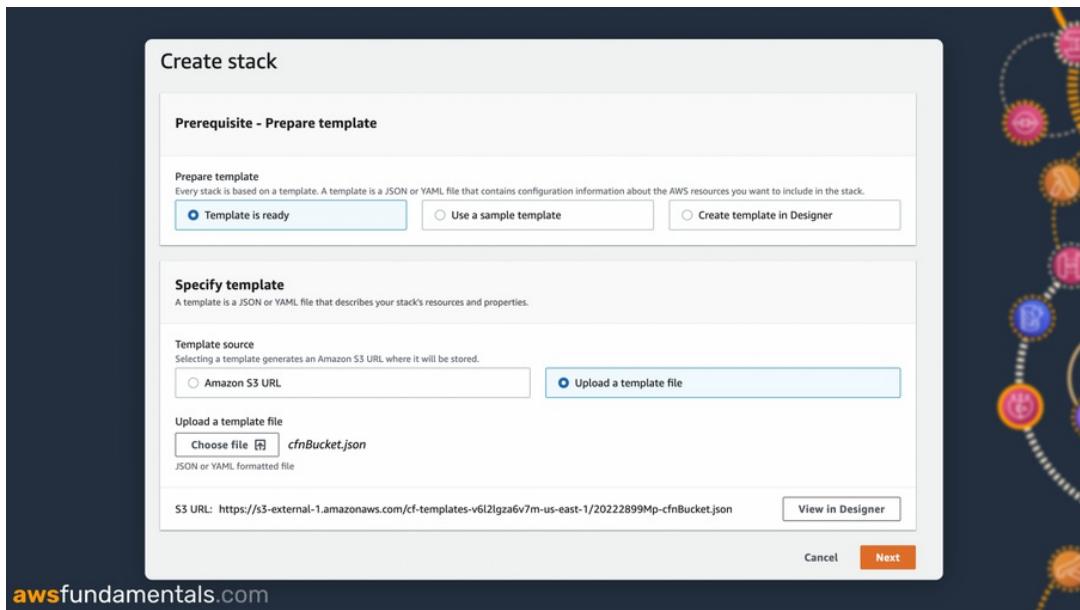


## Let's Deploy

Let's now deploy our example template. We'll show you how to deploy the template with both the web console and the CLI. Let's start with the console.

### Management Console

Go to the service CloudFormation and click on Create Stack → With new Resources.



We have three options for templates :

1. **The template is ready** - Use your own template
2. **Use a sample template** - Use a pre-made template by AWS (really good to get some inspiration)
3. **Create a template in Designer** - AWS has a visual designer for creating templates.

We choose option one and upload the template directly.

This is the template we will use:

```
{
  "AWSTemplateFormatVersion": "2010-09-09",
  "Description": "Sample CloudFormation Template",

  "Parameters": {
    "BucketName": {
      "Type": "String"
    }
  },

  "Resources": {
    "s3Bucket": {
      "Type": "AWS::S3::Bucket",
      "Properties": {
        "BucketName": {"Ref": "BucketName"}
      }
    }
  }
}
```

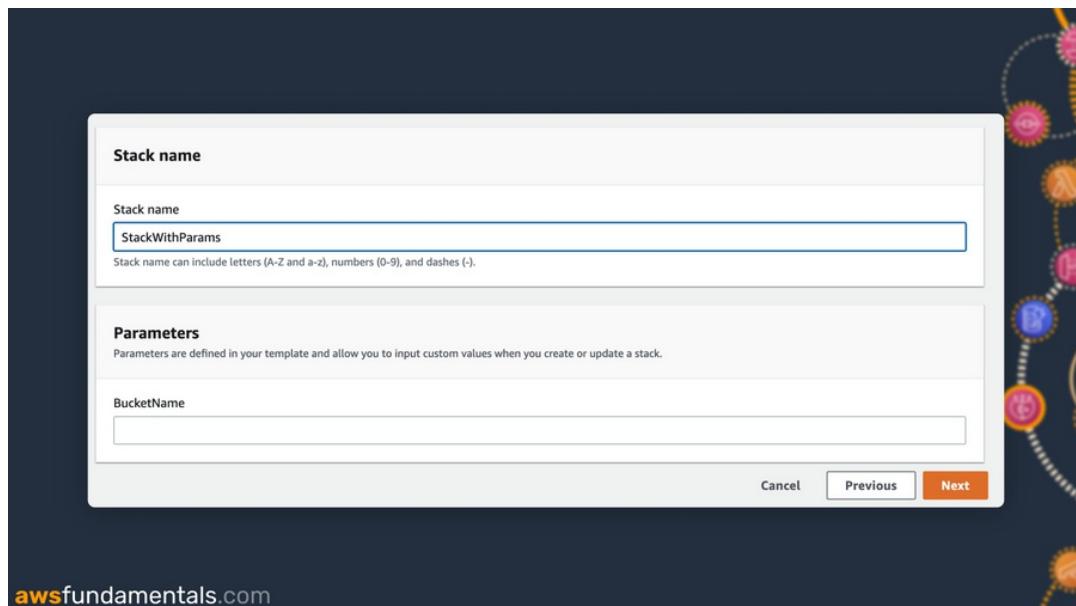
```

    "Properties": {
        "BucketName": {
            "Ref": "BucketName"
        }
    }
},
{
    "Outputs": {
        "BucketArn": {
            "Value": {
                "Fn::GetAtt": ["s3Bucket", "Arn"]
            }
        }
    }
}
}

```

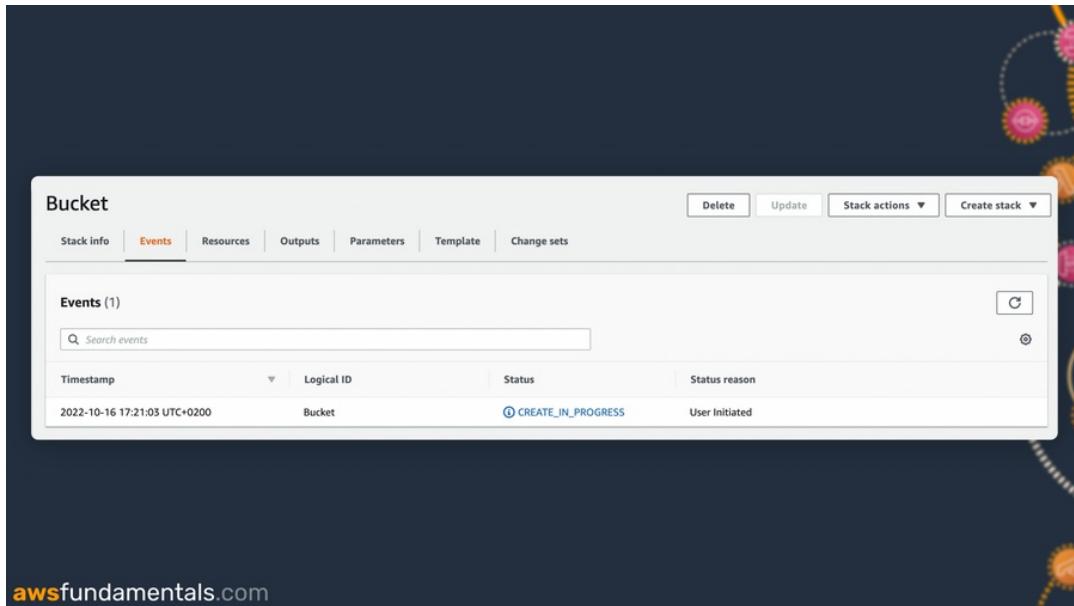
Please copy that, save it as a JSON file and upload it here.

After you click on next you can upload your template. In the next screen, you can enter your defined parameter. For this one, we defined to use of a dynamic bucket name. Enter whatever name you want to. Remember, S3 bucket names need to be **unique across all AWS Accounts in the world**.



This is the `Parameters` section of your template.

After you entered the bucket name click on Next. You can add some more metadata and IAM permissions in the next screens but we skip that and click on **Create Stack**.



Now you see the overview of your stack. When you work with CloudFormation you will look at this screen a lot so let's go through it in a bit more detail. We'll highlight the tabs here.

- **Stack info** - shows you metadata and descriptions of your stack
- **Events** - shows you what happens in the background for provisioning the resources.  
There are different statuses in there like
  - `CREATE_IN_PROGRESS`
  - `CREATE_COMPLETE`
  - `CREATE_FAILED`
  - `DELETE_COMPLETED`
  - `DELETE_FAILED`
  - `DELETE_IN_PROGRESS`
  - `ROLLBACK_COMPLETE`

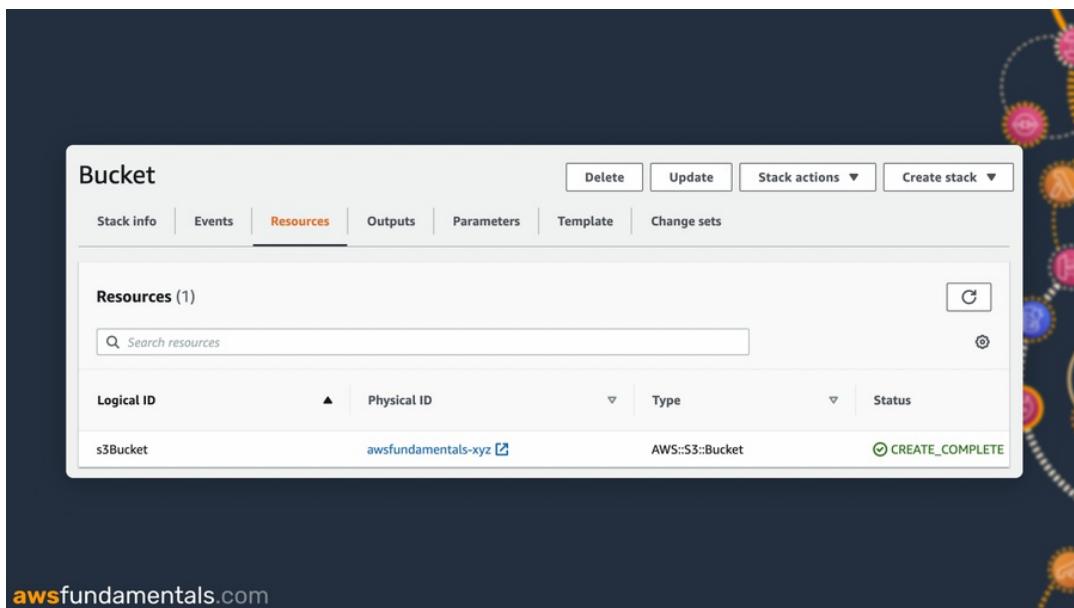
I think the status codes are really self-explainable

- **Resources** - here you see all resources available. In our stack, we will only see one

resource of the type `AWS::S3::Bucket`

- **Outputs** - these are the outputs we defined earlier. We see the `BucketArn` here
- **Parameters** - the user-defined parameters
- **Template** - the actual template that was deployed
- **Change Sets** - any change sets we've applied

The creation of the CloudFormation template should be done after a couple of seconds. You will see it once your stack is in the state `CREATE_COMPLETE`



You can now click on the Physical ID `awsfundamentals-xyz` and your bucket will be open.

Congrats, you deployed your first IAC resources.

## Deploy with AWS CLI

Let's do the same deployment of the template via the AWS CLI. We saved the file in the current folder as `cfn.json`. You can save it wherever you want you just need to update the path here.

Please keep in mind to configure your named profiles correctly on your CLI that you have permission to do the things with the CLI.

We execute the following command and tell you what it is doing in detail.

```
aws cloudformation create-stack \
--stack-name bucketTestStackParam \
--template-body file://cfn.json \
--parameters ParameterKey=BucketName,ParameterValue=Aws-testbucket-123
```

`aws cloudformation create-stack`: This is the actual *command* you want to execute.

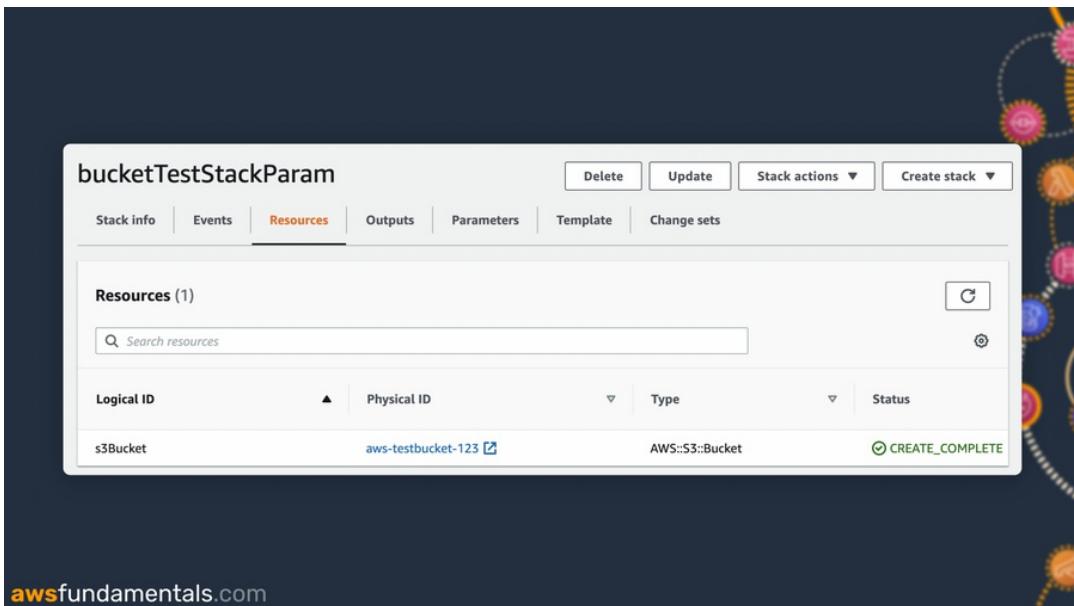
The rest are parameters we pass to the `create-stack` command

`--stack-name bucketTestStackParam`: The name of the stack

`--template-body file://cfn.json`: The path to your file

`--parameters`: The parameters you defined. In that case the `BucketName`

After executing this command you can see the created stack in the CloudFormation console.



Everything you can do in the console, you can also do with the CLI.

## CloudFormation Is Free

Provisioning AWS resources with CloudFormation is free. You only pay for the resources you have actually provisioned. There is a concept in CloudFormation that you provision third-party resources and you would need to pay for those. But we won't cover this here.

## Final Words

CloudFormation is **the** basic IaC service. It is often more common to use a framework on top of CloudFormation. Frameworks like CDK, Serverless Framework, SAM, Serverless Stack, or Amplify. These are all valid choices. Most of them use CloudFormation as the provisioning engine. Terraform is one of the main exceptions here that don't use CloudFormation.

Even when you use a framework, make sure to understand the basics of CloudFormation. Understanding how the process of templates and stacks actually works is important. Most tools create CloudFormation templates.

If you want to get started with IAC tooling we recommend you study some of the **basics of CloudFormation**. Create a few templates, and play around with them.

If you want to get practical with IAC focus on learning Terraform or CDK. This is the present and future of cloud development.

That is why we will cover the same basics in the next chapters. Let's continue with Terraform.



# Using Your Favorite Programming Language with CDK to Build Cloud Apps

## Introduction

Now we see the last IaC tool and one we are most bullish about, the Cloud Development Kit (CDK). CDK is at the top of our history pyramid. It is a so-called **componentized framework** for provisioning your resources.

The main difference between CDK and declarative tools like CloudFormation is the programming language. In CloudFormation you create configuration files in YAML or JSON. In CDK you can use a proper programming language such as Python or TypeScript. After writing your code, CDK will transform this code into CloudFormation templates. This has two amazing benefits

1. You can use all benefits of your **programming language**
2. You can use the powerful **CloudFormation** service

For the examples, we will use **TypeScript**. You can also use the examples in another language of your choice.

## Supported Programming Languages Are TypeScript, Python, Go, C#, and Java

Natively the CDK is developed in TypeScript. All other languages port back to TypeScript.

## Benefits of CDK

CDK allows us to use our own programming languages. This enables us to build abstractions. And it also allows us to use our most-beloved tool, the IDE. The IDE is our Integrated Development Environment. For example, VS Code, Webstorm, or IntelliJ.

Let's see some benefits in more detail.

## The IDE Improves the Developer Experience

The CDK offers you a great improvement in development experience. Developers are already familiar with one or two different programming languages. The only hurdle is how to provision cloud resources in a good way.

Developers know their programming languages in and out. There is no new syntax or primitives to learn. This makes it much easier to get started. The hurdle of learning the syntax of Terraform for example is completely gone.

### You can use your IDE

Developers use & love their IDEs. We know shortcuts, plugins, and our way around all different tasks. Debugging, Refactoring, and coding are all integrated seamlessly.

IDEs are optimized for coding with your dedicated programming language. Using an established programming language allows us to use the full extent of an IDE. We don't need extra plugins or configurations to make it work.

### CDK Wants You to Build Abstractions

CDK allows you to build reusable abstractions, called **constructs**. You can share constructs in your company or even with the whole world (see [constructs.dev](#)).

This is often an argument **against the CDK** because you can abstract way too much. While this is true, this also holds for any other programming language. Don't over abstract and keep it simple.

Abstractions can be super helpful. For example, if you use a lot of message queues. You often have a standard way of building this. You use SQS in combination with Lambda and connect it to your monitoring system. You can create one construct for that and share it with your team colleagues. With that, you make sure that everybody uses it in the same way.

```
new QueueToLambda(this, 'BackgroundTask1, {lambda})
```

### CDK Still Follows a Declarative Approach

In CDK you define **what you want to see and not how it will be created in AWS**. CDK still follows a declarative approach. It abstracts away how CloudFormation takes care of provisioning your data.

In the end, you will have a CloudFormation template that you can check out and look at.

### **CDK Is Stable Because AWS Develops It**

One important factor when deciding on an IaC tool is how stable it will be in the future. The great thing about CDK is that AWS is the main maintainer of CDK. It is open-source and has a great community. All CloudFormation constructs are available with `CFN_` constructs automatically. Most common constructs are available in higher-level `2` constructs by AWS. Later more on that.

This is an important step in deciding which tool to use. You don't want to use any IaC tool that won't port new properties or services fast.

### **CDK Uses CloudFormation**

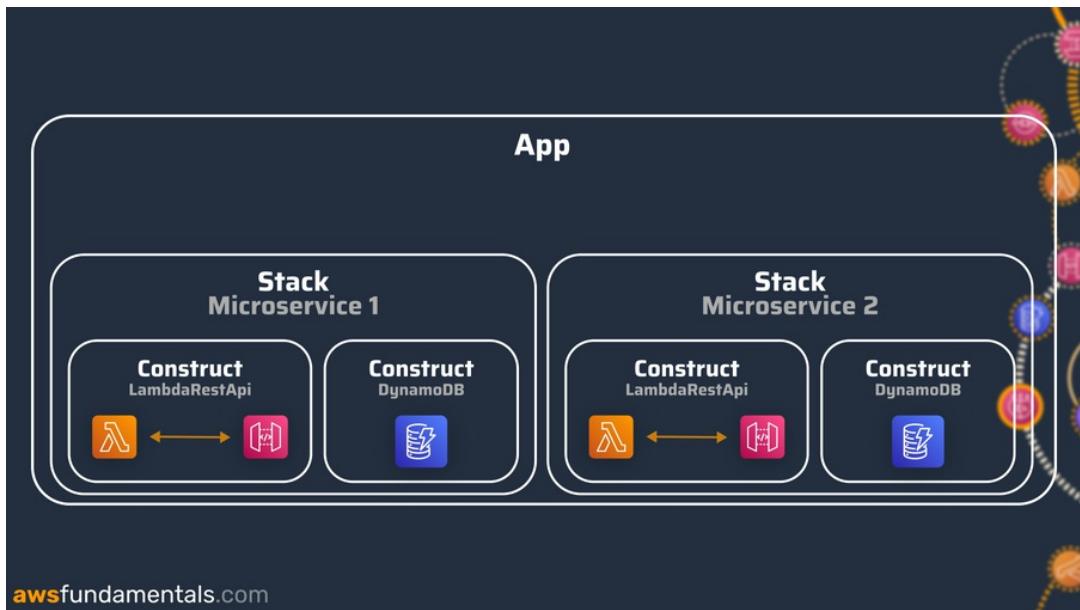
In the background, all CloudFormation concepts are still there. Stacks, Change Sets, and everything related to that. This can be a good thing and is sometimes also a bad thing. But it means you don't need to learn another provisioning methodology. You can instead build on top of CloudFormation.

The main concepts of having a stack or a change set still apply. This is also why it is so important to **learn the basics of CloudFormation**. Let's get started with the actual CDK.

### **CDK Has the Main Concepts of Apps, Stacks, and Constructs**

The CDK follows the concepts of Apps, Stacks, and Constructs.

- **Construct** - the core cloud resource, for example, one S3 Bucket
- **Stack** - a deployable unit of one or more constructs
- **App** - one or more stacks



## Constructs Define the Actual Cloud Resources

We'll start with the smallest unit in CDK, the **Construct**.

Constructs are the main building block in CDK. You should model your application by creating constructs for the different cloud resources you need. Constructs **can be reusable but don't need to be**. That means you can also create a construct that is only used once.

Constructs are differentiated into different levels.

Level	Definition
<b>Level 1</b>	This is the exact CloudFormation construct. All CloudFormation services are automatically published as Level 1 constructs. That means everything that is available in CloudFormation (should be) available as Level 1 Construct. Level 1 constructs always started with <code>Cfn</code> like the <code>CfnBucket</code>
<b>Level 2</b>	Level 2 constructs have some abstractions built-in on top of the level 1 construct and additional APIs. These APIs make working with the services in CDK much easier. A good example is the <code>Bucket</code> construct.
<b>Level 3</b>	Level 3 constructs are mainly community-driven constructs. <a href="https://constructs.dev">https://constructs.dev</a> offers amazing constructs developed by the community. Level 3 constructs are more patterns than actual resources.

### Level 1 Bucket Construct

```
const bucket = new s3.CfnBucket(this, "MyBucket", {  
    bucketName: "MyBucket"  
});
```

`CfnBucket` is the CloudFormation construct. You can pass all parameters you can also pass via CloudFormation

## Level 2 Bucket Construct

```
new s3.Bucket(this, 'MyFirstBucket');
```

`Bucket` is the Level 2 construct of an S3 bucket. It takes the `CfnBucket` and assigns some default values (like autogenerated names) + more accessible APIs.

## Level 3 - CloudFront and S3 Integration

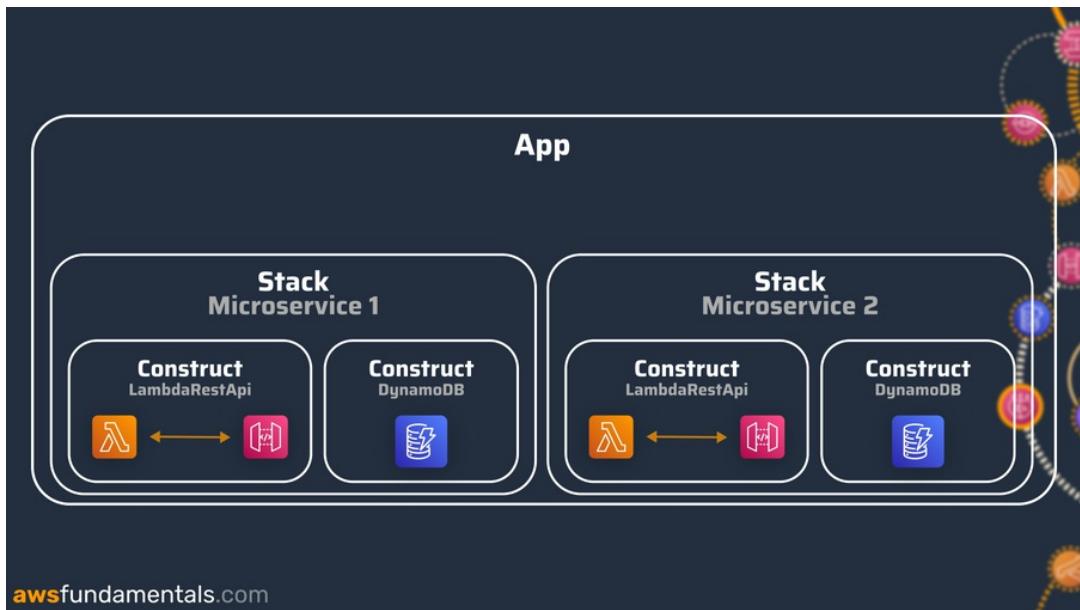
```
new CloudFrontToS3(this, 'cloudfront-s3', {});
```

Level 3 constructs are more like patterns than simple resources. The `CloudFrontToS3` construct, for example, is connecting a CloudFront distribution (CDN) together with an S3 bucket. This is often used for hosting single-page applications.

**Tip:** Try using as many Level 2 constructs as possible when developing your infrastructure with CDK. Don't start over abstracting constructs right from the beginning. See your patterns emerging and build on top of that.

Be really careful with community-driven Level 3 constructs and try to understand them **properly** before you add them. Remember that you deploy services into **your** AWS Account.

It is not always easy to distinguish between Level 2 and Level 3 constructs. Let's see this example.



We can see two types of constructs here.

One is a simple DynamoDB table (`DynamoDB`). The second one is an AWS API Gateway with a Lambda Proxy Integration (`LambdaRestApi`). The latter one is kind of a pattern already and could be seen as a Level 3 construct. But AWS also offers this one as its own construct in `LambdaRestApi`.

### Stacks - Deployable Units

Stacks bundle together multiple Constructs and build them into one deployable unit. The concept is similar to stacks in CloudFormation. One stack contains **one or more Constructs**.

Here is an example stack in CDK:

```
class HelloCdkStack extends Stack {
  constructor(scope: App, id: string, props?: StackProps) {
    super(scope, id, props);

    new s3.Bucket(this, 'MyFirstBucket', {
      versioned: true
    });
  }
}
```

In this stack, we only define the construct `Bucket` and deploy this one.

## Model Your Stacks Either by Services or Stateless Vs. Stateful

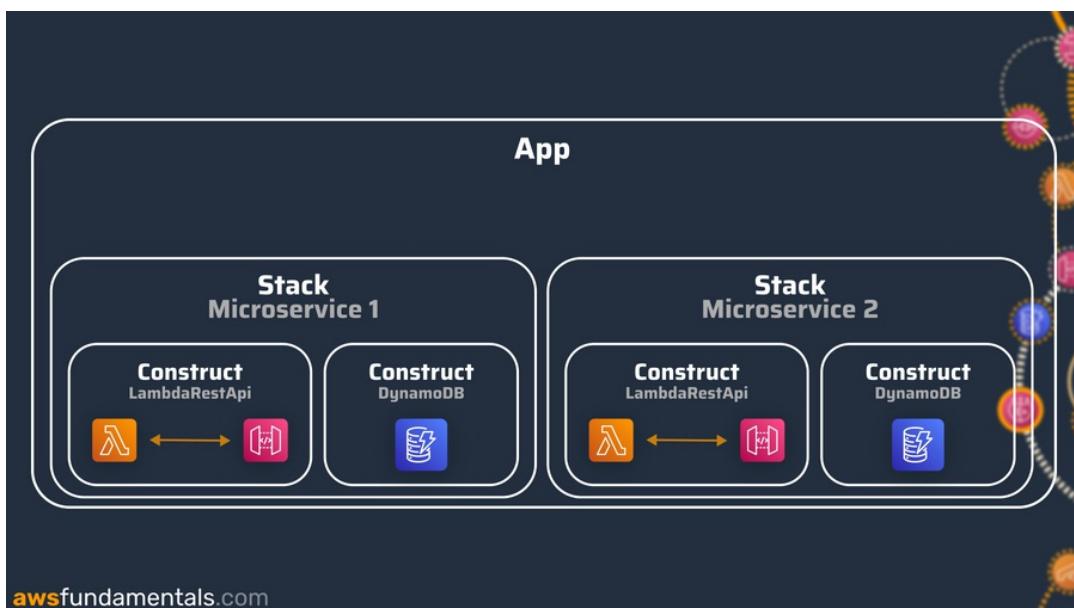
Modeling your stacks is not easy. It is not always straightforward what belongs together and what does not. We showed you some examples in the CloudFormation chapter already.

The general recommendation of AWS is: ***Model with Constructs not Stacks***

That means focusing especially on building your constructs within one stack. Don't create stacks for every cloud resource you need.

You still need to model your stacks.

### Stack per Microservice



One approach to model your stacks is by having **microservices**. One microservice equals one stack. The example image above is one microservice with an API, a Lambda function, and a database.

### Stack per Stateful vs. Stateless

Another approach to model your stacks is by using the distinction between stateful and stateless. One stack would be for stateful resources and another one for stateless resources.

You can determine if a stack is stateful or stateless by thinking about if you can delete & redeploy a stack **without any impact on your users**. If this is the case you've got a stateless stack.

**Stateless** services could be services like

- Lambda
- SNS
- EventBridge

**Stateful** services on the other hand cannot be removed. Typically services here are:

- DynamoDB
- RDS
- SQS

Importing and exporting values between stacks is possible. Don't overdo that or you can find yourself sometimes in bad CloudFormation states. Keep especially an eye out for circular dependencies. Find your resources that belong together and put them into one stack. Model the rest with constructs.

## Apps - Bundle Multiple Stacks Together

This brings us to the last concept, **the App**. The App is the entry point of your CDK application (hence the name). The CDK app combines all stacks.

```
class HelloCdkStack extends Stack {  
  constructor(scope: App, id: string, props?: StackProps) {  
    super(scope, id, props);  
  
    new s3.Bucket(this, 'MyFirstBucket', {  
      versioned: true  
    });  
  }  
}  
  
const app = new App();  
new HelloCdkStack(app, "HelloCdkStack");
```

The app bundles all stacks together and has several phases from constructing, validating, synthesizing, and actually deploying your infrastructure. The details here are not important to get started with the CDK.

## CDK CLI - Initialise, Synthesize, and Deploy Your App

CDK comes with its own CLI. Typical tasks for the CLI are synthesizing, deploying, removing stacks, and much more. Let's go through some of the most important commands

### `cdk init`

Init lets you build a new CDK app. You can pass a `language` parameter to choose the language of your choice.

For example, with `cdk init sample-app --language typescript` you can build a new TypeScript app. You will get a folder `sample-app` with default constructs, and a file structure that can be used to get started with. Your CDK app is available in the `bin` folder and you can see your stack in the `lib` folder.

### `cdk synth`

`cdk synth` is one of the most important commands. This command takes your source code and synthesizes it into a CloudFormation template.

Let's say you created your app with the `init` command stated above. Once you run `cdk synth` a `cdk.out` folder will be created with the name of your stack.

In that case, we will see the following file: `cdk.out/SampleAppStack.template.json`. By looking at this file you see the created CloudFormation file.

This command **needs** to be successful before you can deploy anything. So this is a pretty good command to put into your deployment pipeline or pre-commit hooks.

### `cdk bootstrap`

You need to bootstrap or prepare your account to be able to use CDK. This needs to be done per region and per account.

The bootstrap command prepares your account by creating S3 buckets and assigning IAM permissions so that you are able to deploy your stack. Each environment (region & account combination) needs to be bootstrapped independently.

You can bootstrap your environment by executing the command `cdk bootstrap`

## `cdk deploy`

`cdk deploy` will deploy your CDK app to your defined environment. You need to **bootstrap your environment before you are able to deploy**.

The deploy command is deploying all of your stacks. You can also pass a stack name to only deploy a single stack. `cdk deploy SampleAppStack`

Execute the command `cdk deploy` and your deployment locally starts. IAM changes will always be shown with a warning that you are aware of what you are changing.

## `cdk watch`

CDK Watch was introduced with the latest major version upgrade 2 of CDK.

It is an amazing addition to the serverless development workflow. `cdk watch` will watch your file changes and once you save a file the deployment will start automatically. This is pretty well-known in web development.

The main addition that came with the `watch` command was the ability to hot-swap resources. Hot-swapping means that a resource (a Lambda function, Step Function definition, or VTL template) won't use the normal route of being deployed via CloudFormation. But it will rather be swapped by calling the API directly. This is much faster compared to CloudFormation. This was introduced to improve the local development experience.

Sounds hacky? It is! But it is much faster.

Never do this in production

This functionality enables a much better development process **but it introduces a drift in your CloudFormation deployment**. This is intended. Only do this in development.

## `cdk diff`

This command shows you the diff between your current deployed stack and your local CDK code. This is really good to see what you want to deploy and it is also a good idea to add this to your Pull Requests.

Let's add a bucket to our sample stack.

We add this code to `lib/sample-app-stack.ts` in a newly created sample CDK application.

```
new Bucket(this, "NewBucket")
```

Now we execute the command `cdk diff` and get the following result

```
Stack SampleAppStack
Resources
[+] AWS::S3::Bucket NewBucket NewBucketE3966448
```

This shows us that a new S3 bucket will be added with the name `NewBucketE3966448`.

### **cdk destroy**

`cdk destroy` removes your stacks and destroys all included resources.

## **CDK Makes Working with IAM Simple**

One amazing capability of CDK we want to highlight is how you can grant permissions to roles or users in CDK. As we know AWS follows the **least privilege principle**. That means your roles should grant **as least privileges as possible**.

In CloudFormation or Terraform you used to create IAM Roles and Policies for each service and try to combine them as best as possible. This was really painful because you first needed to understand **which privileges to assign**. And on the other hand, you needed to create the policies by hand. This time is over with CDK.

With CDK Level 2 constructs, you have the ability to grant privileges between common resources. Let's see an example.

We have two constructs. One DynamoDB table and one Lambda function need access to this table.

```
const table = new Table(this, "User", {
    partitionKey: {
        name: "id",
        type: AttributeType.STRING,
    },
    billingMode: BillingMode.PAY_PER_REQUEST,
});

const lambda = new NodejsFunction(this, "function", {
```

```
    entry: "lambda/index.ts",
    handler: "handler",
  }) ;
```

The Lambda function should get and put items in the table. The role looks like that:

```
{
  "Effect": "Allow",
  "Action": [
    "dynamodb:PutItem",
    "dynamodb:GetItem"
  ],
  "Resource": "TABLE"
}
```

In CDK we can assign the role directly to the lambda function with the following command:

```
table.grantReadWriteData(lambda);
```

This gives the exact permissions to the Lambda **without creating the role and policy manually**. This is less error-prone and much simpler compared to creating everything by hand and assigning it to the Lambda function.

If you will ever work on a larger code base you will see that this makes a **huge difference**. You don't need to handle IAM permissions at all anymore. Of course, there are some cases where you still need to assign roles manually. But for most of the default cases, you don't.

## Construct Hub Allows You to Share Constructs Globally

One of the main benefits of using CDK is the ability to create abstractions. Building abstractions leads to sharing code. AWS introduced the constructs hub which is a place to share all different types of CDK Constructs.

The hub is available at <https://constructs.dev>.

It gives you some really nice additions for any part of your development workflow. One construct I really like using is for monitoring & alerting. There are best practices in regard to creating alarms for your constructs. For example, a dead letter queue should always have an alarm on the number of visible messages.

The Monitoring construct is doing exactly that. It scans your stack for different types of

constructs (Lambda, API Gateway, Queues, etc.) and creates automatically alarms for you.

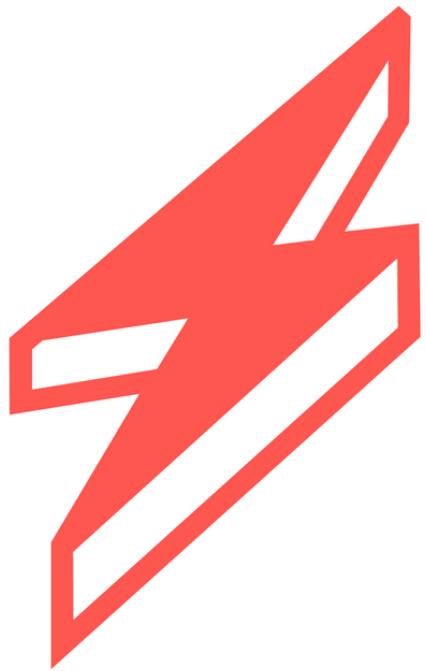
There are many constructs like that out there. Make sure to check them out.

## **Final Words**

This was the introduction to CDK. We use CDK on a daily basis and are really amazed by it. By using a proper programming language you can make the development of cloud-native applications so much easier.

There are definitely still things that need to be improved a lot when it comes to developer experience or drift detection. But the general way the CDK is amazing so far.

I highly recommend you go through the CDK development guide, and workshops, and build some small applications with it. I am sure you will not regret it!



# Serverless Framework

# Leveraging the Serverless Framework to Build Lambda-Powered Apps in Minutes

## Introduction

If we're honest: creating the needed infrastructure for a simple service that is powered by Lambda and is exposed to the internet via API Gateway is not a trivial task, even though this is a very common use case. Yes, we can click together things in the AWS console, but that's not something we want to do for serious projects as it's not easily reproducible and very error-prone.

Even when using CloudFormation, there are a lot of code, resources, and correct references needed to achieve this goal. What if there'd be a framework that solely focuses on everything around Lambda?

And that's what Serverless Framework does. And it does it beautifully. It abstracts away a lot of the boilerplate configuration you need for Lambda, API Gateway, and other native integrations with AWS services. It also does the packaging for you and comes with a powerful CLI that also enables you to easily execute Lambda function code deployments without touching the infrastructure of your stack.

## **Serverless Framework Comes with Different Providers, Formats to Define Your Infrastructure and a Huge Set of Templates**

The entry barrier for Serverless Framework is as low as it gets. Everything you need to do to install Serverless either globally or locally via NPM (`npm i -g serverless`) or YARN (`yarn global add serverless`) and you're already good to go.

If you're having **multiple projects** with **different versions of Serverless**, it will automatically detect if there's a local version in `node_modules` to use.

## **The Different Providers & the Command Line Interface to Interact with Serverless and Your Cloud Provider**

We're focusing on the Serverless AWS provider, which requires you to have the AWS CLI installed. But the Serverless Framework is not limited to the AWS cloud.

If you haven't set it up beforehand, a quick reminder: all you need to do is to run `aws`

`configure` and provide your Access Key ID & Secret Key.

## Describing All Your Infrastructure in the YAML Format

Your Serverless configuration resides by default in your `serverless.yml` file. It's in the YAML format and this is the place where you define your functions, events, and all other AWS resources you might want to deploy.

The magic behind Serverless: it will translate very small abstracted commands into powerful CloudFormation templates in the background.

## There Are Alternative Configurations Formats

Maybe you don't like YAML. Serverless got you covered as you can also define your configuration via

- JSON (`serverless.json`)
- JavaScript (`serverless.js`) or even
- TypeScript (`serverless.ts`)

When working a lot with JavaScript/TypeScript, this is pretty convenient as IDEs mostly have better support than YAML.

```
import type { AWS } from '@serverless/typescript';

const serverlessConfiguration: AWS = {
  service: 'aws-nodejs-typescript',
  frameworkVersion: '*',
  provider: {
    name: 'aws',
    runtime: 'nodejs12.x',
  },
  functions: {
    hello: {
      handler: 'handler.hello',
      events: [
        {
          http: {
            path: '/hello',
            method: 'GET'
          }
        }
      ]
    }
  }
};
```

```
        method: 'get',
        path: 'hello',
    }
}
]
}
}

module.exports = serverlessConfiguration;
```

## Making Use of the Starter Templates for Beginning with a Solid Base

If you're running `sls` Serverless will ask you if you want to use a starter template.

```
→ sls

What do you want to make? (Use arrow keys)
❯ AWS - Node.js - Starter
AWS - Node.js - HTTP API
AWS - Node.js - Scheduled Task
AWS - Node.js - SQS Worker
AWS - Node.js - Express API
AWS - Node.js - Express API with DynamoDB
AWS - Python - Starter
AWS - Python - HTTP API
AWS - Python - Scheduled Task
AWS - Python - SQS Worker
AWS - Python - Flask API
AWS - Python - Flask API with DynamoDB
Other
```

Starter templates are real-world architecture examples. It's a great way of bootstrapping your base application without needing to fiddle together the boilerplate foundations. It also saves you from copying and pasting code from tutorials and stumbling through the documentation first.

As the semantics of the configuration are really descriptive, this offers you a great way of exploring a lot of base templates that you can combine and reuse everywhere.

You can also browse (and full-text search) all existing templates directly at [serverless.com](https://serverless.com).

## The Mastery of Abstraction in Every Aspect

Serverless offers a lot of constructs that help you not only easily bootstrap otherwise complex infrastructure setups, but also rely on best practices that are directly integrated.

### Services - A Dedicated Unit of Configuration

A service is a unit of organization and is wrapped by your `serverless.yml` configuration file. It's later translated into a CloudFormation template to create your resources.

By running `sls` you'll create a new service configuration.

### Functions - Your Compute Layer Running on AWS Lambda

Functions are your compute layer, running on Lambda. They are independent both in execution and deployment. Often, they are single purpose but they don't have to be.

```
package:  
  individually: true  
  
functions:  
  myfunction:  
    package:  
      artifact: myfunction.zip  
      handler: index.handler  
      architecture: arm64  
      name: myfunction  
      runtime: nodejs16.x  
      memorySize: 1024  
      timeout: 10  
      logRetentionInDays: 14
```

The example will create a Lambda function and adequately configure it based on our settings. This is only a small example of the properties you can use with functions.

Serverless Framework also offers easy configuration for:

- VPC attachment

- Layers for externalizing dependencies
- Destinations
- Dead Letter Queues for failed invocations
- Versioning of functions
- X-Ray & tracing

... and much more.

## Packaging Your Deployment Units

Our previous example referenced an externally packaged deployment unit. But Serverless can also take care of packaging your function's code. Instead of using `artifact` you can use `patterns` to tell Serverless which files should be included in your deployment unit ZIP file.

```
package:
  patterns:
    - '!node_modules/**'
    - 'myfunction.js'
```

Run `sls package` to only package your function and create the CloudFormation template, but without actually invoking the deployment of your stack. This helps to validate your configuration beforehand by having a look into the hidden `.serverless` folder which will now include your packaged deployment units and the CloudFormation template that will be applied with `sls deploy`.

## Adding Triggers for Your Functions

Functions are triggered by **events**, which can be e.g. HTTP calls through API Gateway or messages via SNS. What would otherwise be a tedious task, Serverless creates the necessary infrastructure to attach those events to your functions easily.

A look at what needs to be created for API Gateway:

- the gateway itself
- mock responses for CORS

- Gateway resources & methods
- the Lambda integrations attached to your methods

This is what your API Gateway integration will actually look like if done via Serverless. An event, coming via HTTP Gateway for path `/api/*`

```
events:
  - httpApi:
      path: /api/{proxy+}
      method: ANY
```

Serverless Framework took over all of our hassle.

### **Custom Resources That Are Defined in Plain CloudFormation**

The Serverless Framework focuses on Serverless architectures, which mainly evolve around Lambda functions and events. That's why the deepest support is targeted at Lambda, API Gateway, EventBridge, and IAM.

But even if there's no construct available for the AWS service you want to add: Serverless allows you to write CloudFormation directly in your Serverless configuration file. Later, everything will be converted into a CloudFormation template.

Just use `resources` and start writing native CloudFormation code. See an example below which also makes use of CloudFormation references as well as Serverless variables.

```
resources:
  Resources:
    LatencyBasedRecord:
      Type: AWS::Route53::RecordSet
      Properties:
        ResourceRecords:
          - ${self:custom.regionApiDomain}
        SetIdentifier: ${self:provider.region}
        HostedZoneName: ${self:custom.certificateName} .
        Name: ${self:custom.apiDomain} .
        TTL: 60
        Region: ${self:provider.region}
        Type: CNAME
      HealthCheckId: !Ref RegionHealthCheck
```

## Custom Variables for Managing Configurations for Different Stages

You can define your custom variables within a **custom** block and reference them everywhere by using `$(...)`. It's also possible to use the return values that are provided by CloudFormation with references (`!Ref xxx` or `{ Ref ... }`).

```
layers:
  common:
    # [...]

  custom:
    lambdaRuntime: nodejs16.x

functions:
  myfunction:
    package:
      artifact: dist/myfunction.zip
      handler: index.handler
      name: myfunction
      runtime: ${self:custom.lambdaRunetime}
    layers:
      - { Ref: CommonLambdaLayer }
```

## Deploying Your Infrastructure and Code to AWS

Serverless will deploy your resources via CloudFormation for having a managed state and also enjoying other benefits that CloudFormation offers like automated rollbacks in case of erroneous deployments.

Using `sls deploy` will always deploy all of your resources, including the code of your functions. Since v4, Serverless also allows you to use CloudFormation's direct deployments, which come with no downsides and sped up deployments. It will become a default in v4.

```
provider:
  name: aws
  deploymentMethod: direct
```

You can append `--verbose` to have detailed output about the steps Serverless is taking and

also show the outputs of CloudFormation.

If not specified otherwise, Serverless will always fall back to stage `dev` and region `us-east-1`.

Pass a custom stage name or region via the CLI (`deploy --stage prod --region eu-west-1`) or configure it in your template.

```
provider:  
  name: aws  
  stage: preview  
  region: eu-central-1
```

You don't change infrastructure as often as you'll change the code of your functions. Also, deploying with CloudFormation takes time.

Serverless also got your back here: you can deploy your functions independently.

```
sls deploy function --function myfunction
```

Serverless will package your deployment unit (or directly use your referenced unit), compare the hash with the currently uploaded function, and deploy the function directly via the AWS CLI if they do not match. This is blazingly fast and depending on the package size and the upload speed of your function, only a matter of a few seconds.

Especially in the development process, this is tremendously helpful.

## Externalizing Your Dependencies via Layers

Packaging & deploying everything at all times including your dependencies can take a long time, even with a fast uplink connection.

Lambda offers the option to create Layers that will include your dependencies. They can be attached to one or several Lambda functions and each function can have up to 5 Layers in parallel.

Serverless covers this with a few lines of code.

```
layers:  
  common:  
    package:  
      artifact: dist/layer.zip  
      name: common  
      description: 'Common Layer for Node.js'
```

```

compatibleRuntimes:
  - nodejs16.x
retain: false
compatibleArchitectures:
  - x86_64
  - arm64

functions:
  hello:
    handler: index.handler
  layers:
    - !Ref CommonLambdaLayer

```

As with functions, you can also rely on Serverless packaging mechanisms.

## Extending the Capabilities with Official and Community Plugins

Serverless is already a mighty tool with only the onboard tool being shipped out of the box. But it also offers a lot of community plugins for anything you can think of which extend its functionalities even more.

Naming just a few:

- **Resource Tagging** (`serverless-plugin-resource-tagging`) - easily add default tags for all of your template's resources.
- **Domain Manager** (`serverless-domain-manager`) - add custom domain names for your API Gateways without any effort.
- **If/Else** (`serverless-plugin-ifelse`) - create resources based on conditions, e.g. if you don't want to have certain resources only in specific regions or stages.
- **AWS Alerts** (`serverless-plugin-aws-alerts`) - create CloudWatch alerts in just a few lines of code.
- **Step Functions** (`serverless-step-functions`) - create Step Functions event workflows.

The ecosystem for plugins is gigantic and you'll find for almost any use case a proper plugin. No need to reinvent the wheel.

## Marrying with Other Infrastructure-as-Code Tools like Terraform

Serverless Framework focuses on Serverless services that evolve around Lambda. But a fairly large project will often require you to have a lot of other core services as well. Yes, you can manage everything via the custom CloudFormation templates within your configuration file, but this quickly becomes a burden to maintain.

Large infrastructure ecosystems are easier to maintain with tools that allow better organization of your code via blueprints that are easily reusable. One of those tools that master those requirements is Terraform.

We can use the best of both worlds quite easily without adding any workarounds, struggling with a difficult setup, or hardcoding anything. The only thing that's required: we need to synchronize information between Terraform and Serverless Framework via the parameter store of the Systems Manager.

Even though we didn't include Terraform in this book, we'll quickly go into an example. The following sample shows how we'd export the unique identifier, so the ARN, of a resource via Terraform to the parameter store. In this case, it's an SNS topic.

```
resource "aws_ssm_parameter" "chatbot_arn" {
    type      = "String"
    name      = "slack-alerts-chatbot"
    value     = aws sns topic.chatbot.arn
    overwrite = true

    lifecycle {
        ignore_changes = [value]
    }
}
```

After we exported the ARN to the parameter store with Terraform, we can import it into our Serverless Framework template.

```
custom:
  terraform:
    chatBotAlertTopicArn: ${ssm:slack-alerts-chatbot}
```

The only preconditions: we always need to deploy Terraform first and only depend on Serverless resources to Terraform resources - not the other way around. Else we'd get a dependency cycle that can't be resolved anymore.

## **Final words**

Serverless Framework came to stay. It does a great job of quickly spinning up small or large Lambda-powered applications.

We're in love with it for several years and just expect it to become more powerful in the future.

# Credits & Acknowledgements

The biggest thank you is to you, the reader. Thank you so much for buying this book and with that supporting our work.

We never imagined that we would be sharing our work publicly on social media or creating a book in the first place. However, we are grateful for the opportunity to share our thoughts and experiences with others.

We would like to express our appreciation to the following individuals and organizations:

- Our friends and family which supported us throughout the writing process and encouraged us to pursue our dreams.
- Our colleagues and fellow creators that inspired us with their creativity and hard work.
- The team at Freepik, for providing the beautiful illustration on their website that we have used on our cover image. You can find the illustration at the following link:  
**[https://www.freepik.com/free-vector/devops-development-operations-banner\\_7979494.htm](https://www.freepik.com/free-vector/devops-development-operations-banner_7979494.htm)**

We are deeply thankful for the support and encouragement we have received from all of these individuals. Their contributions have been invaluable in helping us bring this project to fruition.

Thank you.

# About the Authors



Tobias loves to build and break things while teaching his learnings along the way. He's a passionate software engineer & educator with several years of professional experience and worked for small tech startups as well as large car manufacturers. While migrating existing production workloads, he got the necessary hands-on to explore and understand the tricks & peculiarities of AWS. This also enabled him to build resilient, large-scale applications from scratch. He's stumbled into all the traps aspiring cloud enthusiasts can fall into so you don't have to.



Sandro is the platform lead at Hashnode where he builds the next generation of developer blogging platforms. There he uses a serverless-first architecture on AWS to build a scalable & performant platform for millions of users. Sandro is also an AWS Community Builder, Co-Founder of ServerlessQ, and writes about AWS on his blog.

Sandro works with AWS for the past 5 years and build production-grade applications in the areas of machine learning, data lakes, container-based services, and web applications. Currently, he focuses mainly on event-driven architecture.