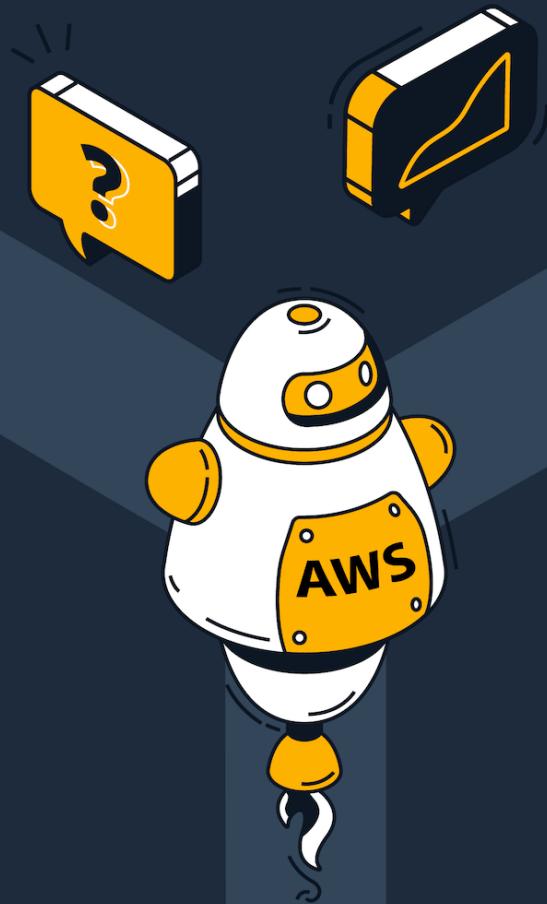


# AWS Fundamentals

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
  - Prologue Version 2
  - 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 give 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.

# Prologue Version 2

We're thrilled to let you know about the second release of our AWS Fundamentals book. Your feedback and suggestions have been a huge help, and we've been working non-stop to give you the most up-to-date and complete info on AWS.

In this update, we've tackled the issues you pointed out to us. We made sure to correct any grammar mistakes and typos that might have made the content hard to understand. We know how important clear, simple language is in tech guides, so we've made sure it's free of errors and easy to get.

Also, you've helped us spot mistakes we didn't catch the first time around, and we've fixed them. We know how crucial it is to get the facts right in tech, and we're sorry for any mix-ups these errors might have caused.

To make the book even more useful, we've added a new section for each service we talk about. These sections go into an extra cool feature of that service that we think you'll find handy.

We've also made sure all the images in the book match up to make it easier to read. By making all the visuals the same style, we hope to give you a smooth reading experience and make it simpler to follow the ideas and examples we're showing.

We want to say a big thank you to everyone who's given us feedback and suggestions. Your thoughts have been super helpful in making this update and making our book even better. Please keep sending your ideas to [hello@awsfundamentals.com](mailto:hello@awsfundamentals.com). Your feedback will help us keep improving our content and make even better resources for the AWS community.

We also want to say that making and publishing an e-book is new for us. We're total beginners at this, but we've done our best to bring you a top-notch update. We're grateful for your patience and understanding as we learn the ropes, and we're dedicated to keep getting better at what we do to give you the best learning materials we can.

Thanks again for your support and trust in our book. We hope this will make your learning journey even better and help you get the most out of AWS!

Best, Sandro & Tobi

# About the Scope of This Book

This book is all about the fundamentals of AWS. The goal is to help you get started with using AWS in the real world.

First, we'll show you how to create your first AWS account, how to set up your root user, and how to ensure that you receive billing alerts.

The next part covers the most important AWS services. AWS consists of more than 255 services, but we have selected the services that you will use in almost any cloud application. Some examples of these 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 provide you with recommendations for the best configuration options, use cases, and a list of tips and tricks to remember.

In the last part, we provide you with an introduction to Infrastructure as Code. We want you to understand the differences between various frameworks, so 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 feels like using superpowers. On one side, you can build applications that are globally available without worrying about infrastructure. On the other side, having the skill of using AWS is in high demand globally.

We want to share our knowledge of both points. By knowing how to build on AWS, you can boost your career and even work on building your own digital product!

We were lucky in the way we learned AWS. During our studies, we worked in companies where experienced employees could teach us the basics directly. We also had the freedom to learn ourselves and make mistakes. Throughout the years, we have honed our skills and gained a lot of insight into different areas. We have seen how AWS has progressed, but the fundamentals have remained the same.

We saw colleagues and friends struggling to learn the core services of AWS and its underlying principles, and how to apply them in 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 for this book.

Learning AWS doesn't need to be hard. It is important to focus on the basics and 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 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 will be 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 way.

This book is also for everybody who has obtained 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 have lost contact with AWS and its configuration options, you can brush up your knowledge quickly and reduce the knowledge gap in your engineering team.

If you have been working with AWS for quite some time but are still unsure about some configuration basics (like when to use long and 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 benefit immensely from reading it.

Firstly, if you are proficient with AWS and have worked in the field for many years, this book may not be for you. We do not require previous knowledge about anything, and that is where we start. By exploring every core service in depth, we aim to provide aspiring cloud engineers with a fundamental tool to start building their own applications or simply to get hired in this field.

This book is also not for people who do not want to work with AWS. If your current focus is Azure or Google Cloud Platform, there is more value in purchasing another book. It can be helpful to understand how AWS handles things, but if you aim to work with another cloud provider, learning their specifics is key.

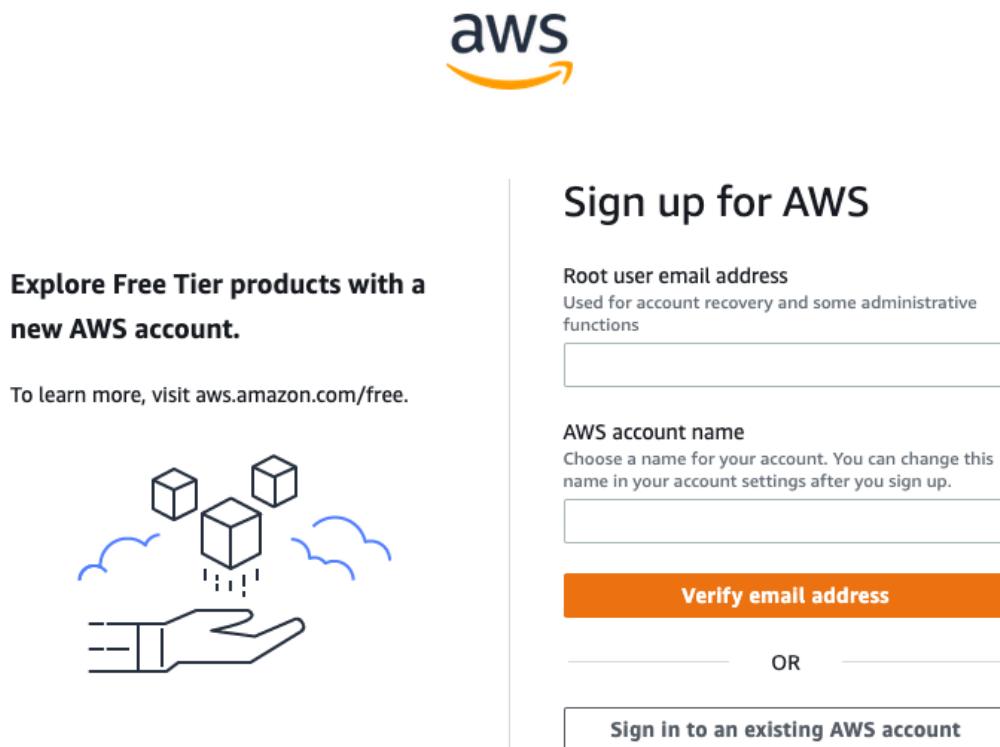
Furthermore, this book does not focus on passing certifications. You will learn the principles required to know how to build applications from scratch and how to apply that knowledge, but passing certifications often requires very deep knowledge that goes beyond the scope of this book. This book is a 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 are focused 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

It's important to read about the fundamentals of AWS, but practical hands-on experience is essential to truly learn anything in the field of engineering. Therefore, it's a must to build and explore your own AWS account, which complements your reading efforts.

To create an account, you need to have a credit or debit card in your name. AWS will charge this card a small amount of \$1 or less, which is only used for validation purposes and won't actually be deducted from your account. After some time, the charge will disappear.



The image shows the AWS sign-up page. At the top is the AWS logo. Below it, the heading "Sign up for AWS" is displayed. To the left, there is a section titled "Explore Free Tier products with a new AWS account." It includes a call-to-action "To learn more, visit [aws.amazon.com/free](https://aws.amazon.com/free)." Below this text is a graphic of three 3D cubes floating above a hand, symbolizing cloud computing. On the right side of the page, there are two input fields: "Root user email address" and "AWS account name." Both fields have placeholder text indicating their purpose. Below the "AWS account name" field is a large orange button labeled "Verify email address". Further down, there is a horizontal line with the word "OR" in the center, followed by another button labeled "Sign in to an existing AWS account".

Besides the payment information, you only need an email address and phone number. The latter 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 how interfaces are structured.

Navigating through services and configurations will get much easier over time, and you'll know 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 can launch unimaginable computing power within seconds.

As you are solely responsible for your account, you should make protecting it properly your first major task in your cloud journey.

This mainly consists of two steps:

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

We urge you not to skip this chapter, even if the desire to start jumping into the services is great. At AWS, there is **no limit to what you can do** and therefore also no limit to what you can get charged for. To understand the significance of this, consider that some EC2 instances can cost up to **\$30,000 per month**.

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

Credentials can be lost or leaked easily, which is why you should rely on a physical second factor to authenticate to AWS with both your root user and your daily IAM users.

Enabling multi-factor authentication (MFA) is very straightforward, and AWS supports both virtual MFA devices (such as the Google Authenticator) and hardware MFA devices like the YubiKey.

Click on the top right of your user and select **Security Credentials** to be 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. <a href="#">Learn more</a>			
	Device type	Identifier	Created on
<input checked="" type="radio"/>	Virtual	arn:aws:iam::157088858309:mfa/root-account-mfa-device	204 days ago

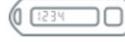
You have several options available, including changing your password, creating access keys for command line access, and assigning an MFA device. If you click on the "MFA device" option, you will be prompted to choose your preferred method.

**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 consecutive one-time passwords, your MFA will be activated and required for the next login.

### A Word about Saving MFA Methods in Your Password Store Application

While it may seem convenient, saving your MFA method in a password store application completely defeats the purpose of having a second factor in the first place. The whole point of a second factor is to require an additional physical or virtual device. As the compromise of an AWS account can have a significant impact on your financial resources, we strongly recommend keeping your second factor a true second factor.

### Staying Away from Using Your Root Account's 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.

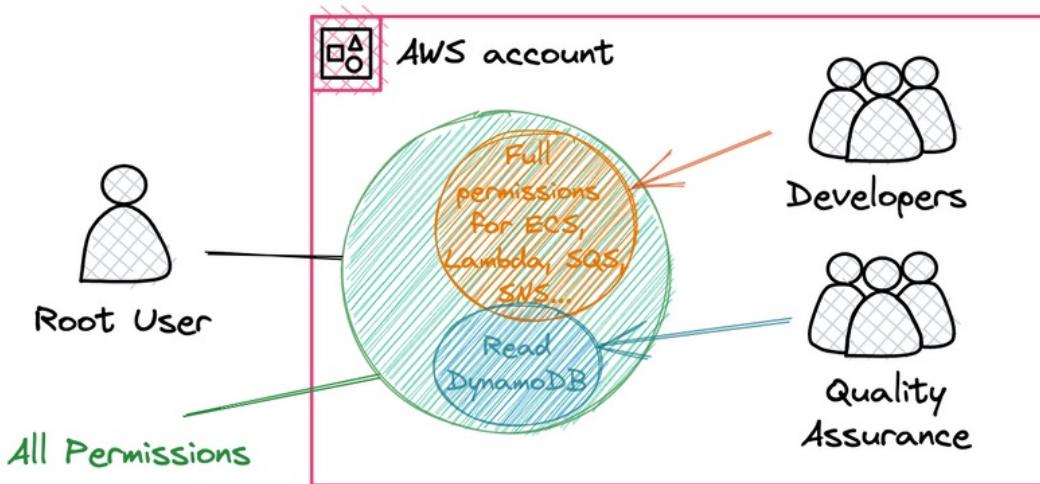
Also, citing the AWS documentation about root users:

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 also manages how you can access your account and what permissions are available.



Let's review 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, such as changing the account name, updating 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 used by a person or by an application, and there can be several for a single account. Unlike the root user, IAM users do not have any permissions initially. 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 a Small or Large Set of Users**

You've probably already realized that, 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 has implemented groups.

A group can have its own permission 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 overseable security cluster that can be better maintained than individual rights one by one.

# Avoiding Cost Surprises

We've already discussed this, but it's worth repeating: there is no enforceable spending limit for your AWS account. Small start-ups have reported horror stories about exploding costs due to recursive Lambda functions, high NAT Gateway data traffic, or exploding logs resulting in terabytes of ingested logs at CloudWatch.

However, there are ways to protect yourself from costs that get out of control:

1. AWS provides its Free Tier, which allows for exploring services without paying much or anything.
2. There are pricing calculators to help estimate costs more accurately.
3. You can set up alarms for cost estimations that 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 offer a certain amount of 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 non-stop running. We'll explore how the 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 they are the ones that matter the most.

AWS 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 can do a lot without spending a dime in the first place. 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.

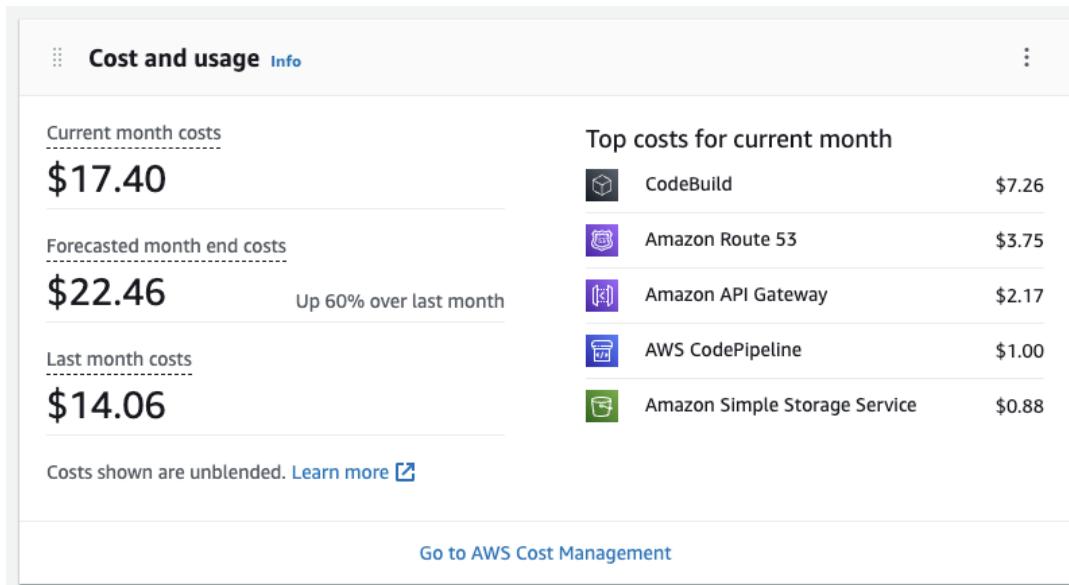
But as with other areas of life, if you're not aware of your spending, you're wandering in the dark, and you can easily get an unwanted surprise at the end of the month.

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

- **CloudWatch** - the cost per used GB of storage is comparably low, but the ingestion of logs can be expensive. If you're logging 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, broken down by service. Therefore, checking the current month's cost is easy, as you will always see it after logging in.

## Getting a Forecast of Your Costs and Receiving Alerts for Exceeding Thresholds

As mentioned earlier, you cannot set spending limits per service or even per account. You will always pay for incurred costs. We have encountered many people who strictly avoid learning anything about the cloud because they are afraid of unexpected costs at the end of the month.

The left panel shows the 'Set alert threshold' configuration. 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 to receive Amazon SNS Alerts. A note about SNS pricing is shown.

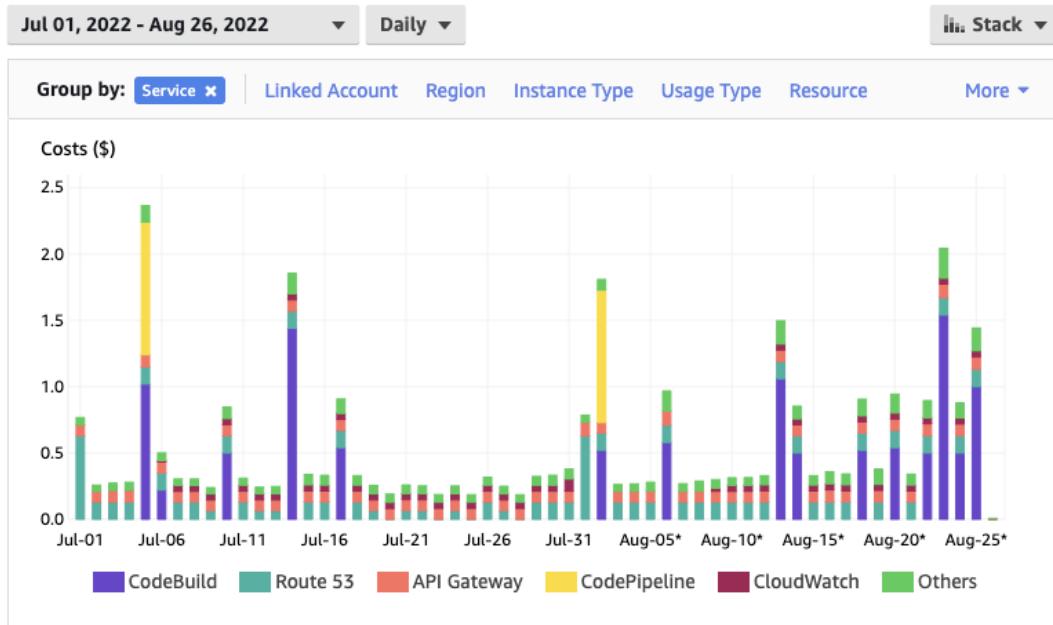
The right panel shows the 'Budget preview' for Aug 2021 - Aug 2022 (MTD) for Unblended costs. It features a bar chart comparing Actual cost, Budget, and Alert #1 (forecasted). The chart shows actual costs peaking at approximately \$40 in Dec 2021, while the budget remains flat at \$50. An 'Alerts' section indicates a forecasted cost alert.

What you can do: Use AWS Budgets to create alerts if cost thresholds are exceeded. AWS also calculates the future costs of your account via estimates based on current and previous usage, which can also be used for budget alerts. The alerts are not in real-time as forecasts are only updated at time intervals, but they 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"/> <a href="#">Find a budget</a>		<a href="#">Show all budgets</a>		<a href="#">Download CSV</a> <a href="#">Actions</a> <a href="#">Create budget</a>		
<input type="checkbox"/>	Name	Thresholds	Budget	Amount used	Forecasted amount	Current vs. budgeted
<input type="checkbox"/>	Default	<a href="#">OK</a>	\$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 deep insights into your cost structure. It enables you to drill down features by service, region, resource, or even instance type. With one glance, you can determine which services contribute the most to your bill and where you could improve 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 of whether it's a component, sub-component, a 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

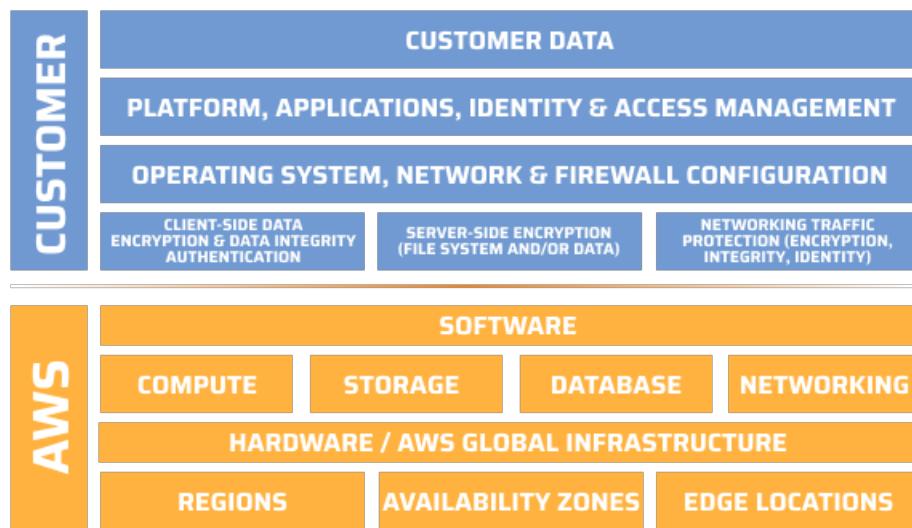
If you're starting out with the cloud and you want to focus on certain core services like Lambda and other related services like SQS, SNS, or DynamoDB, you can take advantage of IAM and adapt your user's permissions to only allow actions on those services.

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 subconscious 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 all infrastructure - to moving more and more responsibilities to the platform provider. This means that 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 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 to operate virtual machines running on EC2.



Another interesting point to note is that the term "hardware" was coined because it was difficult to change. In the past, switching software to new servers or scaling clusters vertically or horizontally was a non-trivial task that required considerable effort. Software, on the other hand, was seen as much easier to modify, as it is simply code running on a platform that can be quickly replaced or adapted.

This world has changed: due to AWS, Azure, and GCP, getting infrastructure almost anywhere 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 is eating the world and taking over more and more difficult tasks and processes, and is integrated into almost every aspect of our lives. 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 not just considered 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.

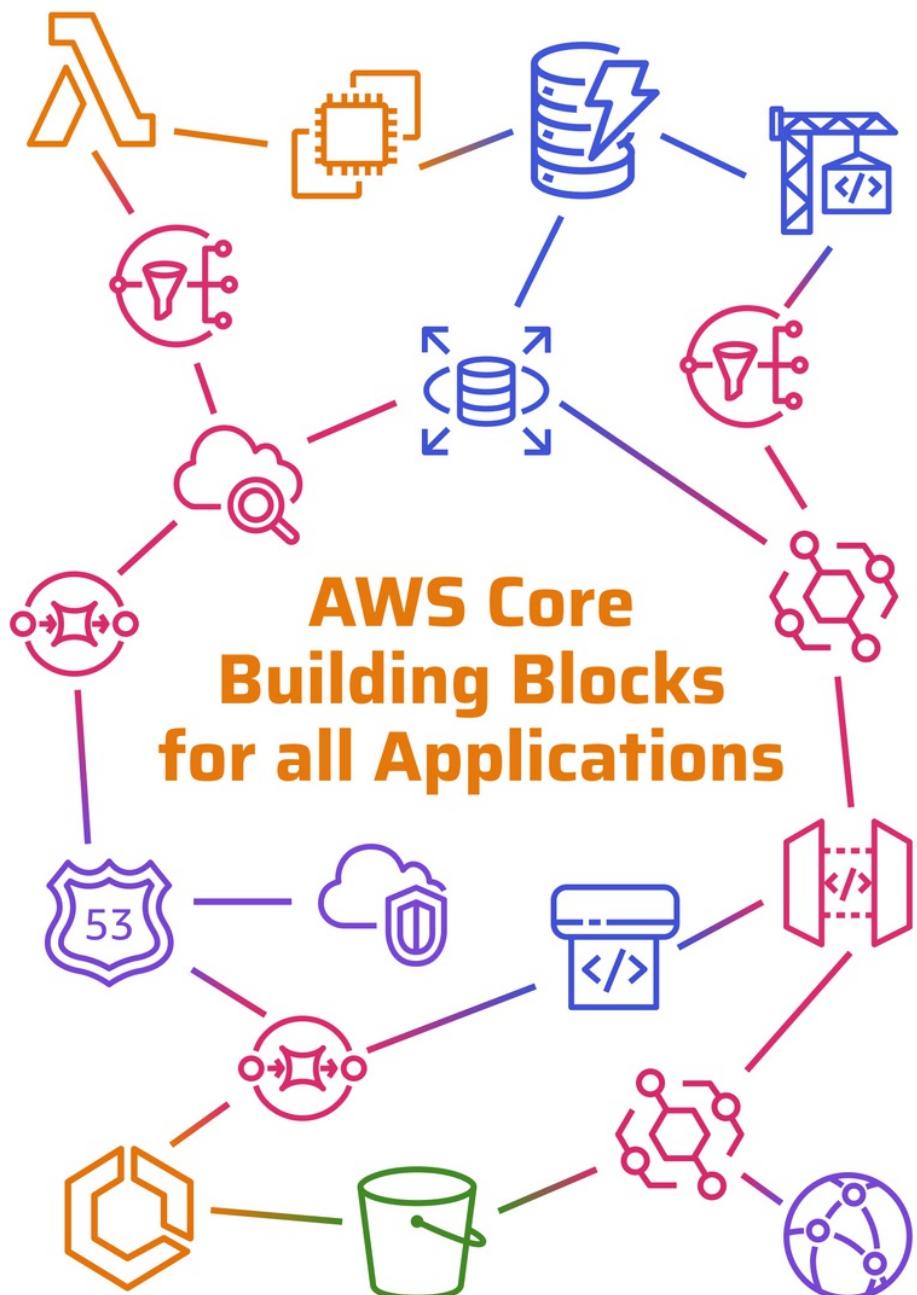
# About going Serverless and Cloud-Native

We strongly believe that the future lies in cloud-native and serverless technologies. But what does this actually mean?

The traditional software development process is often split into two parts: **application development** and **hosting**. Cloud-native, on the other hand, means that application development and infrastructure selection, usage, and integration go hand in hand. This involves leveraging platform technologies that directly benefit the application processes themselves.

The first step that could be observed when public clouds began to take over was that companies gave over responsibility for infrastructure by running containers via cloud providers. With evolving and new services like Lambda - which are even more abstracted, so there's not even a container 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 enables developers to develop not only locally, but also remotely with a fully functioning duplicate of a production system without having to worry about affecting customers. With the rise of serverless technologies and pay-as-you-go models, you also don't have to worry about exaggerated 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 every year. Nevertheless, there are core services that are essential for most applications. In this book, we want to focus on those core building blocks.

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

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 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 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 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.

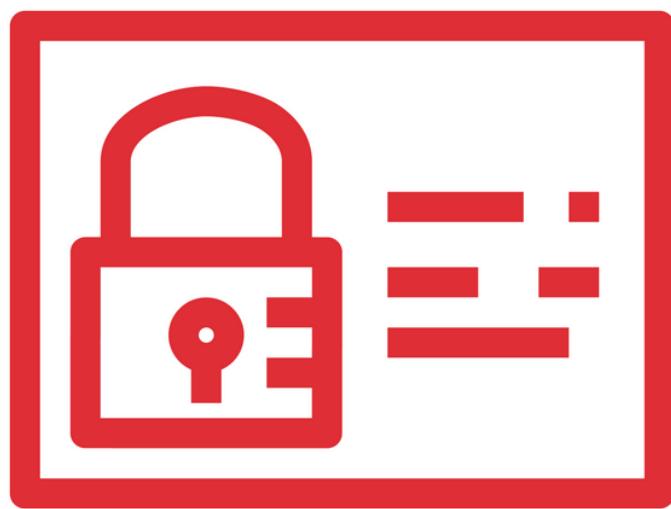
Building and delivering your applications in a reliable way:

- **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

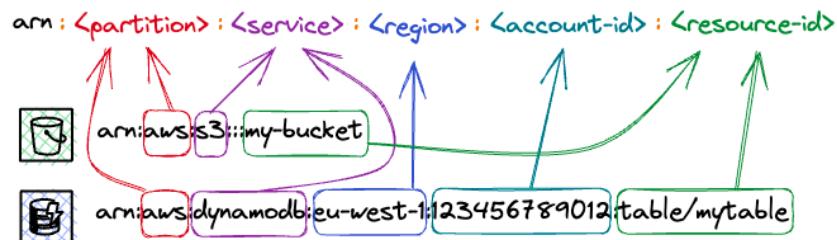
AWS Identity and Access Management (IAM) enables you to manage identities and access AWS services and resources securely. Instead of maintaining dozens of credentials, you'll work with roles and policies that allow fine-grained permissions that can not only be assigned to users but also to resources.

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: to make AWS IAM possible, you need to understand how resources are uniquely identified. This is done via **Amazon Resource Identifiers**, or ARNs for short. Each ARN string 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 also in various contexts, including AWS CloudFormation templates and AWS service APIs. They enable you to specify and authorize access to resources in a secure and standardized manner.

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 introduces 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 more easily manage user rights. Each identity can be associated with one or more policies that specify the permissions for resources granted to that identity. Policies can also be attached to roles that don't represent users or identities but can be assumed by them.

Let's explore the different types of identities 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, as well as their AWS access type(s). It is recommended to use friendly, descriptive names for your users.



You can interact with AWS in two ways:

- **programmatically** by using access keys to make calls to the AWS API
- via the **Management Console** by using a password

AWS IAM users can be configured to be used with one or both of the following methods.

## 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*	john.doe
<a href="#">Add another user</a>	

### 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.

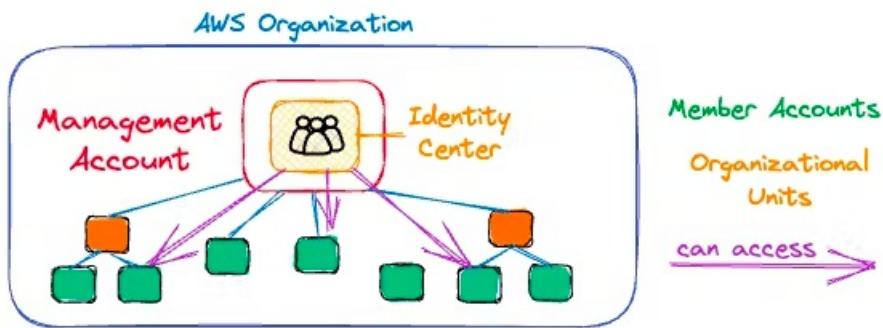
With an enabled MFA for your user, the API access via the Access Key ID and Secret Access Key also changed. Now, you need to request temporary credentials via your keys and the one-time security token that's generated by your MFA application. This is somewhat cumbersome if you want to do this manually,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

## AWS IAM Users Can Be Considered a Legacy Feature

Yes, you read that correctly. Even though we mentioned earlier that you should never work with your AWS account's root user and immediately create your first AWS IAM user, this still holds true. For large-scale projects or enterprises, it's not recommended to work with individual IAM users as it becomes hard to maintain with the growing size of AWS accounts.

We won't go into detail, as this is not a beginner topic, but serious projects are often split into

multiple accounts to leverage their natural security boundaries. This is done via AWS Organizations and AWS Identity Center. With AWS Organizations, you can manage and operate multiple accounts with central management and rules. AWS Identity Center helps to create and administer users across all those organization accounts easily. This also allows for a single set of credentials per user, but still logging on to multiple accounts via single sign-on.



By creating your users in the Identity Center of your organization's management account, you can use them like any other IAM user across the entire organization. You can even use dedicated identity management services such as Azure Directory or Okta to provision your users in your AWS account. The advantage of this approach is clear: you are not bound to AWS IAM, but rather have an independent set of users that can also be used with any service that allows for identity federation.

As previously stated, this is **not a beginner topic**, but it's good to have heard it by now.

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

With an increasing number of users in an AWS account, maintaining individual permissions can become tedious. It's recommended to cluster users into dedicated permission 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:** have enhanced IAM permissions to create and manage users while the company or team grows, shrinks, or adapts in its structure.
- **developers:** have permissions to create AWS resources to build and deploy applications on AWS.

- **quality assurance:** have permissions to access delivery pipelines, databases, and reports.
- **business analysts:** have 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 combined permissions of all of the assigned groups.

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

Roles have some similarities to users: they are 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 that 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 have the permission to do so.

This is a great feature to offer short-lived credentials to external services that are not under 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 whether to `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, to retrieve objects from the `mydata` bucket. The **Resource** element specifies the 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**

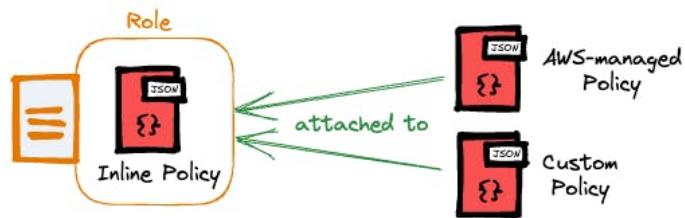
By creating tailor-made policies, you can assign only necessary permissions for an identity. Unlike AWS-managed policies, you have full control over granted or denied permissions and don't need to worry about permission updates.

IAM roles allow us to group a set of permissions without associating them with a specific identity. Instead, we can assign these roles to one or more principals, such as a user, service, or group, granting them the relevant permissions.

Roles help us adhere to sound security protocols by assigning only the necessary permissions to users or services to carry out their functions. This is commonly known as the principle of least privilege.

Permissions granted by a role are defined via policies assigned to the role, which can be added in the following ways:

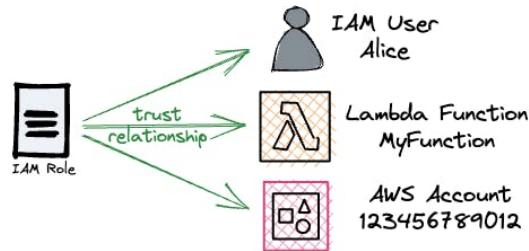
1. **Inline**, directly included in the role definition.
2. Via an existing **managed policy** attached to the role.



Option two can either be a custom-managed, tailor-made policy or one managed by AWS.

If you use AWS-managed policies, keep in mind that they may receive updates. As a result, there may be conflicts with permissions or excessive permissions may be granted.

**Trust policies define which principal is allowed to use a role.** Roles can be adopted by various entities, such as IAM users, AWS services, AWS accounts, or entire AWS organizations.



AWS mandates that we specify the precise principals authorized to access them by defining a trust policy for the role.

A trust policy for a Lambda function would look like this:

```
{
  "Version": "2012-10-17",
  "Statement": {
    "Effect": "Allow",
    "Principal": {
      "Service": "lambda.amazonaws.com"
    },
    "Action": "sts:AssumeRole"
  }
}
```

We see the three required elements in this case:

- **Effect** - By default, all permissions are denied. This policy will grant additional permissions by setting it to `Allow`.

- **Principal** - The policy allows the principal to utilize an AWS service, specifically AWS Lambda in our case.
- **Action** - The principal is granted permission to allow the service to assume the role.

### 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 easily and clearly grant a set of permissions.

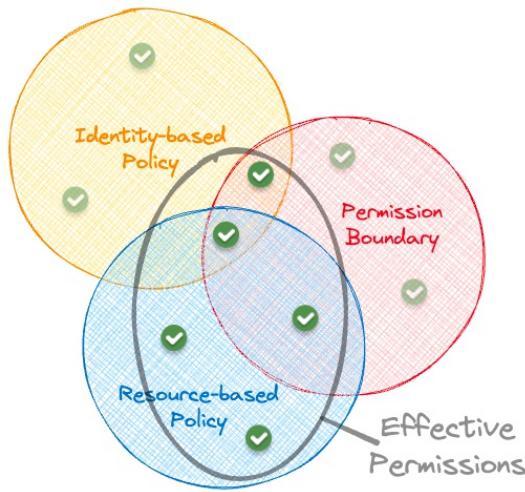
One way of doing this is by using conditions in your policy statements, for example, with the `StringEquals` comparator.

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

The statement above grants all permissions for ACM, API Gateway, and CloudFront to resources that have the tag `App` with the value `my-first-app`. By correctly assigning the tag to our certificate, API Gateway deployment, and CloudFront distribution for our target app, we can use this 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: has a policy attached that grants **Query** and **UpdateItem** permissions for **all DynamoDB tables** in the AWS account.
- Role B: has a policy attached that allows the creation (**CreateTable**) of DynamoDB tables.
- A permission boundary 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 **cannot be restricted** via permission boundaries.

### More Policy Types for Advanced Use Cases

Additionally, there are less well-known policy types that you'll likely not come across frequently:

- **Organizations Service Control Policies (SCPs)**: You can attach AWS accounts to an AWS Organization to have centralized billing and permission management. This allows you to make 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 the resource-based and 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 varies 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 against best practices and testing them beforehand.

### **Policy Creation Support via Github Copilot in your IDE**

GitHub Copilot is an excellent tool for creating IAM policies directly within your IDE. When working with Terraform, you can easily create new policies using two approaches:

1. **Leveraging code suggestions** - Start by defining a new resource type and give it a detailed name.
2. **Descriptive comments** - Write a comment that includes your intentions for the policy.

Both methods work, although the second option tends to produce better results. You can easily iterate both approaches with additional comments within the sections.

One of the primary benefits of using GitHub Copilot is that it allows you to stay within your IDE. While there are ChatGPT integrations available, they typically require you to open an additional tab to view the response from OpenAI. Furthermore, these responses often include explanations that may not be necessary for the task at hand, although they can be removed with appropriate prompts.

### **Creating Policies from Scratch with Natural Language and ChatGPT**

With ChatGPT, there are no rules to follow when inputting our requirements. We have the

freedom to describe our needs in any way we wish. We can use a single sentence, break it down into multiple statements, list it out, or use just a few keywords. ChatGPT will take care of the rest.

We're not restricted to using only one resource per prompt. We can generate multiple resources from a single prompt.

Moreover, ChatGPT is context-aware. This means that we can write additional prompts to fine-tune our previous results.

## 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 Access Analyzer's JSON editor interface. At the top, there are tabs for 'Visual editor' and 'JSON', with 'JSON' being selected. 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. The policy defines two statements: one allowing S3 access to all buckets and another allowing DynamoDB access to specific tables. Below the JSON code, there is validation information: 'Security: 0', 'Errors: 1', 'Warnings: 0', and 'Suggestions: 1'. A red underline highlights an error at line 19, column 16, indicating an invalid ARN account ID. A tooltip for this error states: 'Ln 19, Col 16 Invalid ARN Account: The resource ARN account ID 01234567891 is not valid. Provide a 12-digit account ID.' At the bottom of the editor, there are buttons for 'Search errors', 'Learn more', 'Feedback', and a close button.

```
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 Warnings: 0 Suggestions: 1

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 the 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.

## **AWS Organizations and Single-Sign-On for Comfortably Working with Dozens of AWS Accounts**

This may go beyond the fundamentals and is often not done correctly even by large enterprises, but it's an important topic that we want to cover quickly: AWS Organizations.

AWS Organizations enables you to easily manage many accounts under a central organization. It includes account management and consolidated billing capabilities in a single place and tools like authorization and management policies to meet any security and compliance needs or requirements.

### **Centralized Consolidation with AWS Organizations**

AWS Organizations provides a unified platform for managing and regulating multiple accounts within a single entity.

Among its robust features are:

- **Account Administration** - Allows for the creation and management of multiple accounts within one central location. This includes transitioning existing accounts into an organization, generating new member accounts, and implementing automation rules for new account creation.
- **Consolidated Billing** - The organization's member accounts are exempt from handling billing, as it's transferred to the management account. However, each AWS account can still utilize the Free Tier.
- **Policy and Permission Implementation** - Permissions and compliance policies can be applied across different accounts from a single location, enabling cross-account access.
- **Identity Federation** - Centralized management of identities and permissions is possible with AWS Organizations, including federation with a corporate directory for Single Sign-On via AWS Identity Center.

These are just a few examples of its potent capabilities.

### **Management Accounts, Organizational Units, and Service Control Policies Are Used to Structure Your Organization and Separate Resources**

Let's delve into the key concepts of AWS Organizations. Those are important to understand its inner mechanics.

## **Management Account - The Organization's Owner**

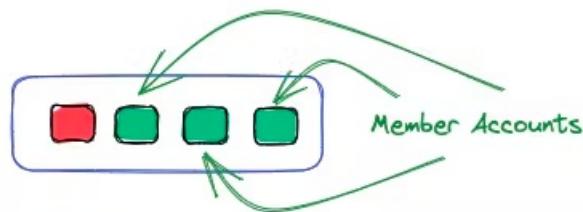
The management account is the initial account that was used to create the organization. It has full access to all AWS resources and services.



Due to its unlimited permissions, it should only be used for the initial setup and account management tasks. But more on the best practices for working with the management account in the later chapters.

## **Organizations - A Container for Multiple AWS Accounts**

An organization functions as a collective hub for various AWS accounts. It facilitates centralized management, enabling the application of policies and permissions uniformly across all associated accounts.



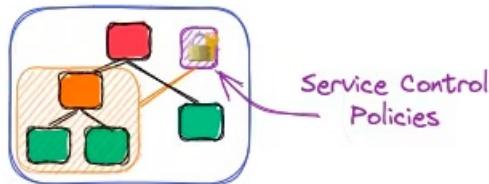
## **Organizational Units - A Logical Grouping of Accounts**

Organizational units serve as an additional instrument for nuanced management. Each unit, essentially a logical container, can encompass accounts and other organizational units. This arrangement supports the structuring of applications, projects, or even the hierarchical layers within your organization.



## **Service Control Policies to Restrict Access to Services and Resources**

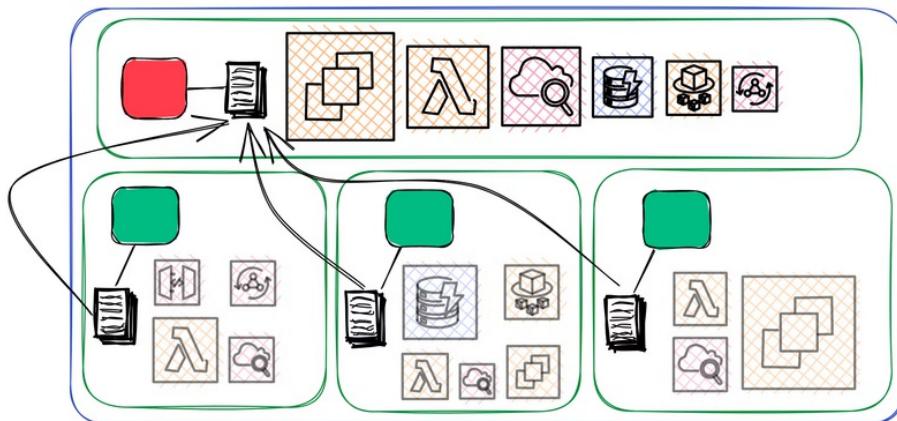
Service Control Policies (SCPs) can be applied to an organization, organizational unit, or individual account. They allow you to restrict access to AWS resources and services.



They help you to apply fine-grained permissions across your set of accounts so you can fulfill compliance rules and enforce strict security setups.

### **Consolidated Billing - Billing in a Single Place**

With Consolidated Billing, payment for all member accounts will be moved to the management account. The management account can also access all billing and account information, including member account activity within the organization.



This not only simplifies the billing process but also allows you to gain deep insights into the spending of your organization.

### **Federated and Delegated Access**

Working with multiple accounts and IAM roles can be tedious. With AWS Organizations you can manage permissions and roles for all users in one place.

With Identity Federation and IAM Identity Center, users can access multiple accounts within the organization with a single set of credentials. You're able to define roles they can assume and therefore the final permissions they will receive.

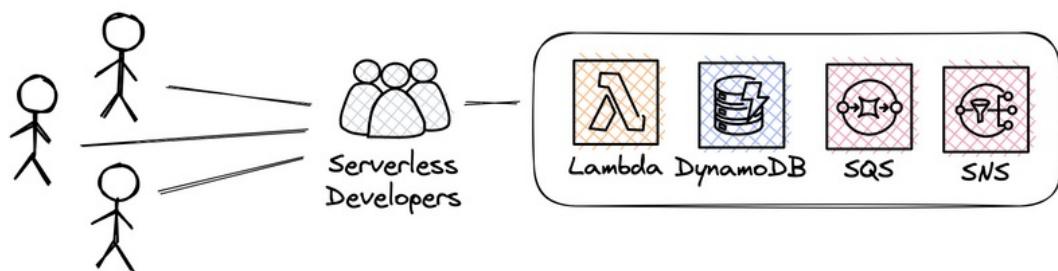
## Use Cases for AWS IAM

If taken seriously, this list could be endless, 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.

### Use Case 1: Creating Users with Restricted Permissions That Are Managed via a Group

Working with AWS can feel terrifying at first, 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 excessive costs when not handled properly, can help calm your nerves.



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 expensive EC2 instances or RDS clusters are created and never cleaned up.

### Use Case 2: Creating an S3 Bucket That's Only Accessible for Users with Activated MFA

Amazon S3 is a great service to store all kinds of unstructured data, including 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 is known, human failure is how many data leaks are created and how sensitive data gets exposed to the internet. Due to the sheer unlimited amount of accounts a single person has 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 who 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

When working with IAM, it's crucial to understand all of its fundamentals, but there are a few important things to remember:

- **Try to grant the least privilege** - Every time you create a new policy or 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, so rotating them regularly helps to reduce the risk of misuse.
- **Use groups to manage permissions** - With groups, you can manage the rights of

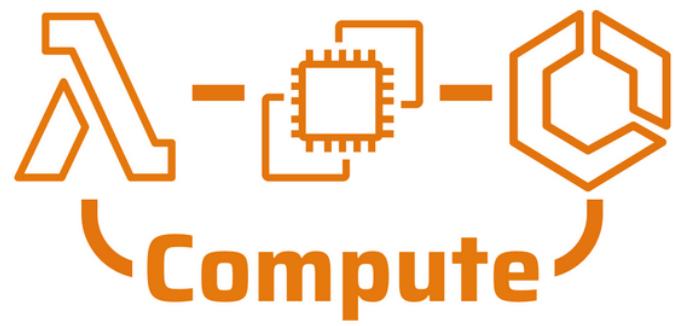
multiple users in a single place, reducing operations while increasing security.

- **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 detect and prevent misuse of keys.
- **Leverage IAM Roles as much as possible** - When using AWS services for your workloads, it's crucial to utilize roles with customized permissions. This ensures that users have only the permissions they need to perform their specific tasks, and AWS will handle the issuance of temporary credentials to prevent concerns about database password leaks.

## Final Words

Mastering IAM is like finding the holy grail. It is a complex topic, and there will always be days when you get stuck on some permissions issue, regardless of your experience.

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



# 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 to choose from. Let's go through a brief history of how computing has evolved over the past years.

On-premises 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.

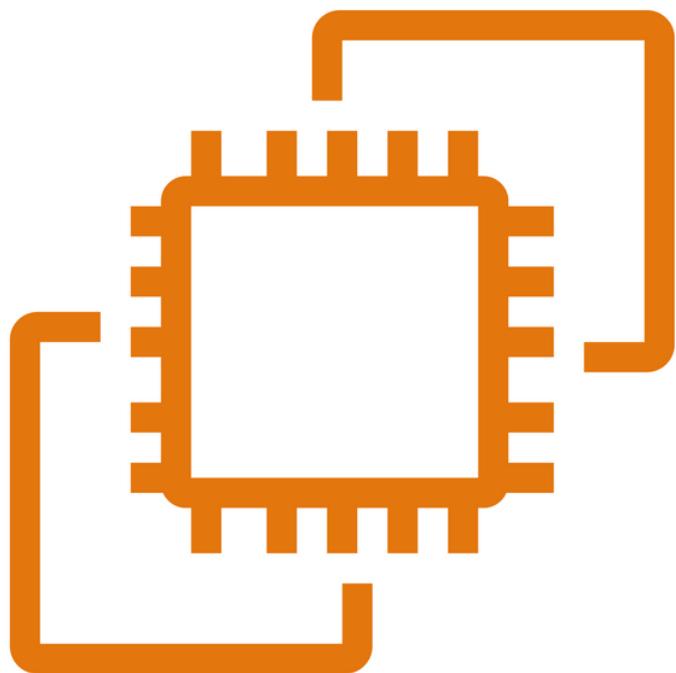


With the advent of cloud computing, it's now possible to rent virtual machines, such as with **EC2**. This allows scaling compute resources up and down as needed, without requiring significant up-front investments. However, this virtualization still requires management and maintenance by its users, such as 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 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 these 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 was one of the first services launched by AWS back in 2006. With EC2, you can rent virtual machines to run your own applications, allowing you to scale your application's capacity up or down as needed. This makes 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 that are perfectly designed for every specific need.

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

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

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 computationally intense operations like machine learning.
2. **Flexibility & Control** - Lambda limits you to certain runtimes and completely abstracts away the 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 computation times and the number of requests, as with AWS Lambda, you'll be billed for your chosen instances that can be a perfect fit for your needs in terms of instance types and sizes.

Overall, EC2 and other container approaches can be used in conjunction with serverless solutions like Lambda. EC2 is well-suited for steady and high computing demands, while Lambda offers the best approach for small, often event-driven workloads that can benefit from the rapid scaling and flexibility of the serverless model.

## **Fundamentals for Working with EC2**

Before working with Amazon Elastic Compute Cloud (EC2), it is important to remember its fundamental concepts.

### **Virtual Machines**

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

Each 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. They contain 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 cannot ensure the integrity or security of the AMIs, so it is 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 they will have a major impact on performance and the bill you will 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 that 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** - provides 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 is stored with no hierarchy and is 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 we 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.

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 left panel displays a search bar and a 'Quick Start' section with icons for Amazon Linux, macOS, Ubuntu, Windows, Red Hat, and SUSE. Below this is a detailed view of the selected 'Amazon Linux 2 AMI (HVM) - Kernel 5.10, SSD Volume Type' (ami-01cae1550c0adea9c). The right panel is titled 'Summary' and contains fields for 'Number of instances' (set to 1), 'Software Image (AMI)', 'Virtual server type (instance type)' (t2.micro), 'Firewall (security group)' (default), and 'Storage (volumes)' (1 volume(s) - 8 GiB). A tooltip for the 'Free tier' is visible, 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 GiB of EBS storage, 2 million I/Os, 1 GB of snapshots, and 100 GiB of bandwidth to the internet.' At the bottom are 'Cancel' and 'Launch Instance' buttons.

In addition to the image, you can also select your preferred architecture, including x86 and ARM. ARM runs on the latest Graviton2 processors by AWS, which offer a better performance/price ratio. However, they may cause conflicts with the software you require that is not supported.

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 also shows family details: t2, 1 vCPU, 1 GiB Memory, and pricing information for On-Demand Linux and Windows. A 'Compare instance types' link is available. 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 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. It includes fields for VPC selection (set to 'vpc-03e559f382af5deda'), subnet selection ('No preference (Default subnet in any availability zone)'), and auto-assign public IP ('Enable'). A 'Firewall (security groups)' section allows selecting an existing security group, with 'Select existing security group' chosen and a dropdown showing 'default sg-03e3f8df769310c30'. There is also a 'Compare security group rules' link.

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

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

The screenshot shows the 'Configure storage' section. It specifies a root volume of 8 GiB using the gp2 storage type. A note indicates that free-tier eligible customers can get up to 30 GB of EBS General Purpose (SSD) or Magnetic storage. An 'Add new volume' button is available, and the current status shows 0x File systems.

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 that 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"/> Find instance by attribute or tag (case-sensitive)				
<input type="checkbox"/>	Name	Instance ID	Instance state	Instance type
<input type="checkbox"/>	awsfundamentals	i-0388d84f1c8fcd730	Pending	t2.micro

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 and private key set or use an existing one. When generating a new pair, AWS will store the public part on the instance (concretely 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

**i** 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 without needing to manage any key pairs. This is achieved with EC2 Instance Connect. It's a good idea to make use of the session manager as it doesn't require you 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 get either the public IP address or the domain name of the EC2 instance we created before. These 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.

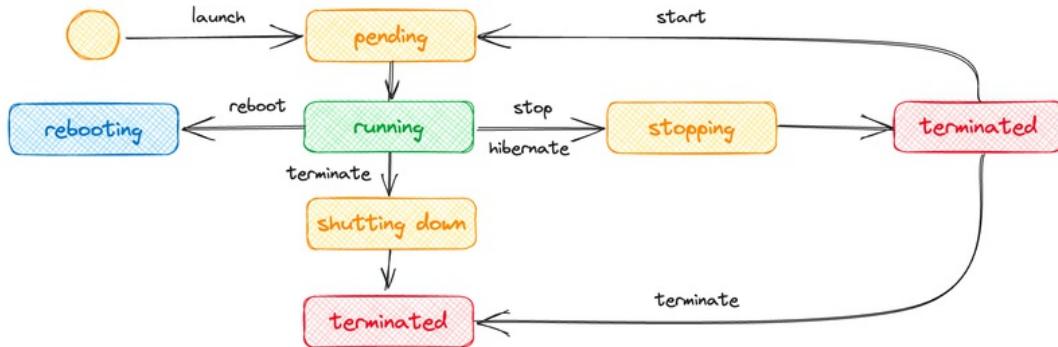
**Side note on this process:** Depending on the operating system that is running on your EC2 instance, you may have to use another username. For example, the username for Amazon Linux instances is **ec2-user**, while the username for Ubuntu instances is **ubuntu**. You can find the correct username for 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 needed again 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 instance storage, block store volumes persist over time and can be detached from and reattached 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.

Like 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 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 (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 remapped 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.

In summary, 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 vary drastically in pricing. If you plan to use EC2 regularly, it's your 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 are a variety of services that help you 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 detect problems with the software or physical hardware, such as 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.

## **Getting Detailed Insights: Monitoring Instances with the Amazon CloudWatch Agent**

Monitoring and optimizing the performance of your Amazon Elastic Compute Cloud (EC2) instances is a critical aspect of AWS infrastructure management. To gain valuable insights into the health and performance of your EC2 instances, as well as troubleshoot issues proactively, Amazon Web Services (AWS) offers a powerful tool: the Amazon CloudWatch Agent. In this subchapter, we will delve into the importance of the CloudWatch Agent, how to install and configure it on Amazon Linux 2, its benefits, and advanced configuration options. We'll also discuss the crucial aspect of managing data ingestion and retention to optimize costs.

### **Importance of Monitoring and Collecting Metrics**

Effective monitoring of EC2 instances is essential for maintaining a reliable and efficient cloud environment. Without the CloudWatch Agent, you miss out on real-time visibility into system-level metrics, logs, and custom metrics. This deficiency hampers your ability to identify performance bottlenecks, allocate resources optimally, troubleshoot problems efficiently, and centralize logs for comprehensive analysis.

The CloudWatch Agent bridges this gap by enabling the collection and publication of crucial performance data. It also empowers you to gather custom metrics specific to your applications, offering deeper insights into their behavior. In essence, the CloudWatch Agent is the linchpin of comprehensive monitoring capabilities for ensuring the smooth operation of your EC2 instances.

### **Benefits of Using Amazon CloudWatch Agent**

The Amazon CloudWatch Agent empowers you with a myriad of benefits for monitoring your EC2 instances effectively:

- 1. Effortless Monitoring:** With the CloudWatch Agent, monitoring becomes a breeze. You can effortlessly track critical metrics such as CPU utilization, memory usage, disk space, network utilization, and more, providing a holistic view of your instances' performance.
- 2. Centralized Logging:** The agent allows you to centralize your logs in CloudWatch Logs. This centralized approach streamlines log management, making it easier to search, analyze, and troubleshoot issues across your entire infrastructure.
- 3. Custom Metrics:** Beyond default metrics, the CloudWatch Agent supports the collection of custom metrics. This flexibility allows you to monitor application-specific metrics,

providing tailored insights into your application's behavior and performance.

4. **Optimization and Alarms:** By leveraging the CloudWatch Agent's capabilities, you can optimize instance performance, identify bottlenecks, set up alarms to proactively respond to issues, create customized dashboards, and seamlessly integrate with other AWS services.

## Installation and Configuration on Amazon Linux 2

Installing the CloudWatch Agent on Amazon Linux 2 is a straightforward process, but it requires careful configuration. Let's go through the steps:

### Prerequisites for Installing Amazon CloudWatch Agent

Before you begin, ensure that the EC2 instance you're working on has the necessary permissions by attaching the `CloudWatchAgentServerPolicy` to its IAM role. If you're unfamiliar with AWS Identity and Access Management (IAM) roles and policies, you can refer to AWS's comprehensive guides on the topic.

### Step-by-Step Installation Guide

1. **Connect to Your EC2 Instance:** Access your Amazon Linux 2 instance using SSH or your preferred remote access method.
2. **Update Package Manager:** To ensure you have the latest package information, update the package manager's cache with the command:

```
sudo yum update -y
```

3. **Install the CloudWatch Agent Package:** Use the following command to install the CloudWatch Agent package:

```
sudo yum install -y amazon-cloudwatch-agent
```

4. **Configuration File:** Once the installation is complete, the agent's configuration file can be found at `/opt/aws/amazon-cloudwatch-agent/etc/amazon-cloudwatch-agent.json`. Edit this file using a text editor of your choice to specify the metrics and logs you want to collect and define the destination in CloudWatch.
5. **Start the CloudWatch Agent:** To start the CloudWatch Agent and apply your configuration, run the following command:

```
sudo /opt/aws/amazon-cloudwatch-agent/bin/amazon-cloudwatch-agent-ctl -
```

```
a fetch-config -m ec2 -c file:/opt/aws/amazon-cloudwatch-
agent/etc/amazon-cloudwatch-agent.json -s
```

6. **Verify Agent Status:** Confirm that the CloudWatch Agent is running correctly by checking its status:

```
sudo /opt/aws/amazon-cloudwatch-agent/bin/amazon-cloudwatch-agent-ctl -
m ec2 -a status
```

If the agent is running successfully, you should receive a message indicating its status.

### **Configuration Options and Customization**

The Amazon CloudWatch Agent offers a plethora of configuration options and customization features to tailor the monitoring experience to your precise requirements. Here are some key features:

1. **Metrics Collection:** Configure the agent to collect system-level metrics, including CPU utilization, memory usage, disk space, and network activity. You can also define and collect custom metrics specific to your applications.
2. **Logs Collection:** Specify which log files or log groups to collect and send to CloudWatch Logs. Customize log file paths, patterns, and filters to capture specific log data.
3. **Log Streaming:** Configure the agent to stream logs directly to CloudWatch Logs or an intermediary service like Amazon Kinesis Data Firehose for further processing or archival.
4. **Metric Filters:** The agent supports metric filters, allowing you to define rules for extracting specific data from logs and creating custom metrics based on that data.
5. **Alarms and Notifications:** Set up alarms based on metric thresholds and configure notifications to trigger actions when specific conditions are met. This enables proactive monitoring and alerting for critical events.
6. **Configuration File:** The agent relies on a configuration file that permits you to define settings such as metrics, logs, log file paths, log patterns, and more. This file can be modified to customize the agent's behavior to your exact specifications.

These configuration options and customizations grant you the flexibility to fine-tune the CloudWatch agent, ensuring you capture the right metrics and logs for effective analysis and troubleshooting.

## Collecting Metrics and Logs

To successfully run the CloudWatch agent on your server, you must create a configuration file tailored to the agent's requirements. This file encompasses all the necessary settings and parameters needed for the agent to function optimally.

### The CloudWatch Agent Configuration File

The JSON configuration file serves as a blueprint for the agent's data collection efforts, covering:

- Default and custom metrics.
- Logs.
- Traces.

You can create this file in two ways: using a user-friendly wizard or manually. The wizard offers simplicity in creating the configuration file but provides limited customization. On the other hand, manual creation offers greater control and flexibility, allowing you to specify metrics not available in the wizard.

When using the wizard, execute the following command:

```
sudo /opt/aws/amazon-cloudwatch-agent/bin/amazon-cloudwatch-agent-config-wizard
```

The wizard will guide you through the process, asking questions about the installation context, log file handling, retention periods, and default metrics. It will also detect credentials in your credentials file (`~/.aws/credentials`), making configuration easier.

After completing the wizard, the configuration file can be found at `/opt/aws/amazon-cloudwatch-agent/etc/config.json` on Amazon Linux. If you've created it manually, it may have a different name.

### Starting the Agent

To initiate the CloudWatch agent on an Amazon EC2

instance via the command line, use this command as an example:

```
sudo /opt/aws/amazon-cloudwatch-agent/bin/amazon-cloudwatch-agent-ctl -a fetch-config -m ec2 -s -c /opt/aws/amazon-cloudwatch-agent/etc/config.json
```

Replace the configuration file path with the actual file name you used for your configuration parameter.

## Collecting Metrics

By default, the CloudWatch agent collects numerous metrics that can be immediately utilized. These metrics cover a range of aspects, including CPU utilization, memory usage, and disk space, each providing crucial insights into system performance.

For instance, some default metrics include:

- `cpu_time_active`: Measures the time the CPU is active in any capacity.
- `cpu_time_idle`: Measures the time the CPU is idle.
- `disk_free`: Represents free disk space.

These metrics are visually displayed in the CloudWatch metrics section, where you can create custom dashboards to visualize your most critical data.

Moreover, the CloudWatch Agent isn't confined to predefined metrics. You have the freedom to develop custom metrics tailored to your unique use cases, enabling you to monitor specific aspects of your applications or systems effectively.

## Setting Up Log Collection

The CloudWatch Agent allows you to specify which logs should be forwarded to CloudWatch Logs. Let's look at an example configuration:

```
{
  "agent": {
    "metrics_collection_interval": 60,
    "run_as_user": "root"
  },
  "logs": {
    "logs_collected": {
      "files": {
        "collect_list": [
          {
            "file_path": "/var/logs/application.log",
            "log_group_name": "/aws/ec2/my-app/application",
            "log_stream_name": "{instance_id}"
          }
        ]
      }
    }
  }
}
```

```
        "filters": [
            {
                "type": "include",
                "expression": "(.*)"
            }
        ],
        {
            "file_path": "/var/log/httpd/access_log",
            "log_group_name": "/aws/ec2/my-app/access-errors",
            "log_stream_name": "{instance_id}",
            "filters": [
                {
                    "type": "include",
                    "expression": "\\\\\"\\\\\\s(4|5)\\\\\\d{2})"
                }
            ]
        }
    ]
}
```

In this example, we configure the agent to collect data from two log files: `application.log` generated by our application and `access_log` provided by HTTPD, which logs successful and unsuccessful HTTP requests.

Each log file's configuration includes:

- `file_path`: The path to the log file on the instance.
  - `log_group_name`: The name of the log group in CloudWatch Logs where the logs will be stored.
  - `log_stream_name`: The name of the log stream in CloudWatch Logs where the logs will be stored. `{instance_id}` is a placeholder that will be replaced with the actual instance ID.
  - `filters`: An array of filters to apply to the log data, specifying which log entries should be

included or excluded based on regular expressions.

These configurations ensure that relevant log data is forwarded to CloudWatch Logs, making it accessible for analysis and troubleshooting.

### **Filtering and Transforming Log Data**

The CloudWatch Agent offers powerful filtering capabilities to help you focus on the log entries that matter most. In the provided example, we filter log entries with HTTP status codes starting with 4 or 5 to exclude entries indicating client or server errors.

You can selectively include or exclude log entries based on specific patterns or criteria, effectively filtering out irrelevant or sensitive data. Additionally, you can use regular expressions to transform log data before sending it to CloudWatch Logs, such as extracting specific fields or modifying the format of log entries. These capabilities provide fine-grained control over your log data and enhance your ability to extract meaningful insights.

### **Important Disclaimer: Keep an Eye on Data Ingestion and Retention**

While the CloudWatch Agent is a powerful tool for monitoring EC2 instances, it's essential to manage data ingestion and retention carefully to control costs effectively in your AWS environment. Consider the following key points:

- **Log Retention:** CloudWatch charges based on the amount of data ingested and stored. Setting appropriate log retention periods ensures that you retain logs for the necessary duration without incurring unnecessary costs for retaining logs that are no longer needed.
- **Data Volume:** The more data you ingest into CloudWatch, the higher your costs will be. Avoid excessive data ingestion by filtering out less critical logs and preventing unnecessary charges from accumulating, particularly if you have a high volume of logs.
- **Data Filtering:** Not all logs and data have the same level of importance or need to be retained for the same duration. By setting proper retention periods and filters, you can optimize your data storage costs and focus on storing only the most critical information.

### **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, they most likely don't include all the software you need.

As described earlier, shared AMIs are used 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 considering use cases for EC2, the possibilities are endless depending on your needs. There is nothing you can't do.

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

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

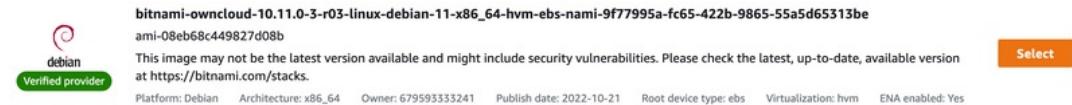
One of EC2's major benefits is the option to run tasks on compute-optimized instances. With spot instances, you'll also benefit from significant discounts. By selecting `Spot Requests`, you can create a bid for a desired number of spot instances, which will automatically launch the instances when they 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 cloud storage, similar to Dropbox. With AMIs, you don't have to build it yourself but can rely on a marketplace or community image, such as ownCloud.



There are many great Amazon Machine Images (AMIs) available for various use cases. You can even run your own email server or host a mail server for Minecraft.

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

With EC2, you can run any operating system you want. This is especially useful for system-bound development chains like those for iOS applications, which require macOS.

By launching an EC2 instance with macOS, you can build your own delivery pipeline for iOS applications with the Continuous Integration/Continuous Delivery (CI/CD) tool of your choice. Keep in mind 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 also be used to run simple proxy servers that allow you 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 to minimize unnecessary expenses. If you're using EBS volumes, remember that you'll still be 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 in your cluster. Simply run, stop, and manage containers.

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

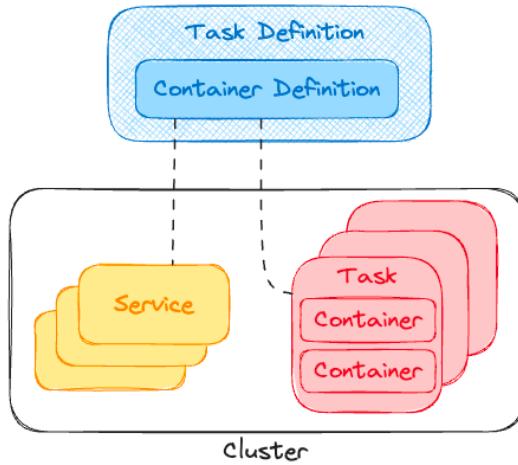
- Don't worry about the underlying infrastructure with the Fargate launch type. You'll only need to determine which container image you want to run and what workload capabilities you require in terms of memory or virtual CPUs.
- It's fully integrated with AWS IAM. You can define fine-grained permissions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

As we believe 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 is 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 encounter many key terms that may not be intuitive or easy to grasp at first. However, they are crucial to understanding how all the internals of ECS work 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** - it contains all the necessary information to be executed on any machine, such as certain versions of libraries or languages. Additionally, you can 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 face operating challenges regarding the management and orchestration of your container landscape, such as deployments, starting and stopping, and scaling of your containers, among others.

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

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

A task definition is a blueprint for launching 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: EC2, Fargate, or External?

- **Roles** - You'll need 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 that will be injected into environment variables from AWS Secrets Manager). On the other hand, a task role grants permissions to the actual application that runs inside your containers (e.g. for 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).
- **vCPU & memory allocation** - The compute resources that you want to assign to the task definition. The values that can be assigned depend on your launch type and the values vCPU and memory depend on each other (you can't use any vCPU value with any memory value). Generally, EC2 offers way higher compute resource capabilities than Fargate.
- **Environment variables** - Key-value pairs that are injected into your service. These are mostly used to inject stage-dependent 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 via the task execution role and provides them to your container instance. Don't forget to pass the necessary read permissions for your secrets to this 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 the actual execution of a task definition. It comprises a set of containers that run together on the same host. Tasks are defined using the Docker-Compose file format, which enables you to specify the container image, environment variables, port mappings, and other options for each container in the task.

You can also launch tasks directly. The task will remain active until it is either manually

stopped or it exits on its own. In either case, 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 ensures that a desired number of tasks are always running. If a task stops because a container exits for any unexpected reason (e.g. due to runtime errors) and the number of healthy tasks falls below your threshold, ECS will take care of it and start a new task as a replacement.

### **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 have gone over the fundamentals of containers and ECS's ability to orchestrate them. 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**

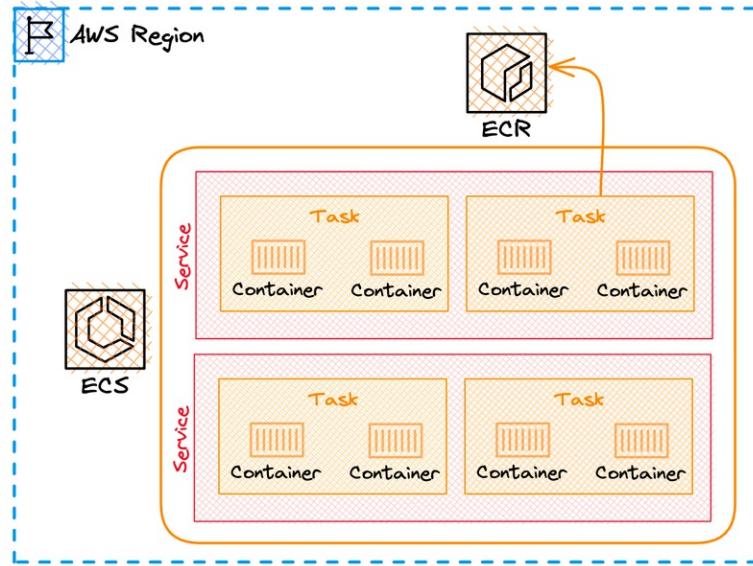
Deploy EC2 instances in your clusters to run your containers. With EC2, you have full control over the underlying infrastructure, including the instance type, operating system, and security groups.

It's a good 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**

Fargate is a serverless pay-as-you-go option where you don't have to 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 (which are entirely abstracted and therefore removed from your responsibility), and runs the tasks on those instances.

It's the recommended launch type for small to large workloads that may require low overhead and experience occasional bursts. If you're not completely certain about your requirements, always choose the Fargate launch type to minimize operations and liabilities.



#### **External - Orchestrating Your On-Premises Containers**

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

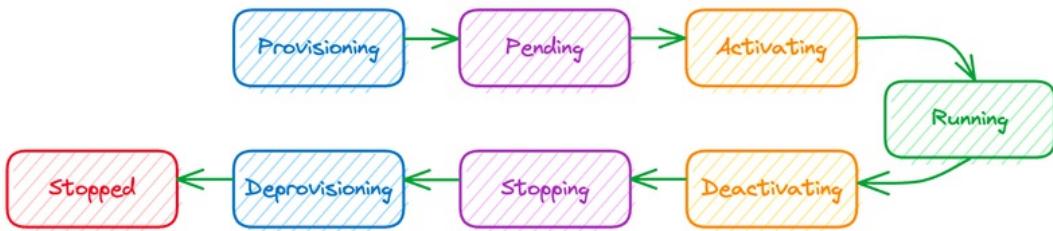
#### **Task Scheduling - Running Your Tasks**

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 Passes through Different States in Its Lifecycle**

A task passes through different lifecycle states, regardless of whether 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.



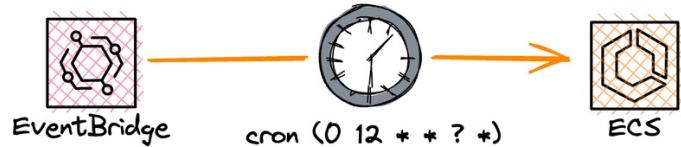
The lifecycle of a task includes the following states:

- **Provisioning** - Preconditions are in progress, for example, 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 take place before moving to the running state, for example, 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, for example, 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, for example, 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 requires 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 that you can trigger.

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

Afterward, 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 the Amazon **Elastic Container Registry (ECR)**. It's a fully-managed container registry that allows you to host and share an unlimited number 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 pay for the amount of data and transfer costs to the internet. This means that when Fargate is pulling images so it can run your container images inside

tasks, this won't introduce any costs.

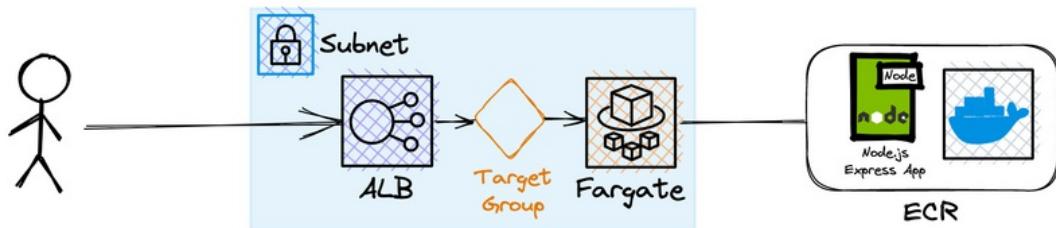
### Creating Our First ECS Service That Runs a Node.js Application in Fargate

After we have acquired all the necessary knowledge, we can proceed to build our first service that runs 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 from 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 is 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 and 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 will also create a target group that monitors our Fargate tasks for their healthiness.
7. Submit an HTTP GET request to our application to receive our expected response.

A very simple architecture diagram, without going into the details of task, task definition, service, and cluster 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 that it's not available unless we're authenticated with valid credentials for our AWS account.

The screenshot shows the 'Create repository' wizard in the Amazon ECR console. The top navigation bar shows 'Amazon ECR > Repositories > Create repository'. The main section is titled 'Create repository' and has a sub-section titled 'General settings'. Under 'Visibility settings', the 'Private' option is selected, with the note: 'Access is managed by IAM and repository policy permissions.' The 'Repository name' field contains '157088858309.dkr.ecr.eu-west-1.amazonaws.com/awsfundamentals'. A note below says: '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.' Under 'Tag immutability', the 'Disabled' option is selected. A note at the bottom states: 'Once a repository has been created, the visibility setting of the repository can't be changed.'

After clicking `Create`, you will be taken to the new repository. Click on the top right corner where it says `View push commands` to get all the necessary commands needed to work with your new repository.

It will look something 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. Install NPM on your machine. For macOS, this can be easily done via a package manager like homebrew and `brew install npm`.
2. 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 have 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 will be 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.

Amazon Elastic Container Service > Create cluster

### Create cluster Info

An Amazon ECS cluster groups together tasks, and services, and allows for shared capacity and common configurations. All of your tasks, services, and capacity must belong to a cluster.

#### Cluster configuration

Cluster name

awsfundamentals

There can be a maximum of 255 characters. The valid characters are letters (uppercase and lowercase), numbers, hyphens, and underscores.

#### ▼ Networking Info

By default tasks and services run in the default subnets for your default VPC. To use the non-default VPC, specify the VPC and subnets.

VPC

Use a VPC with public and private subnets. By default, VPCs are created for your AWS account. To create a new VPC, go to the [VPC Console](#).

vpc-03e559f382af5deda

default

Subnets

Select the subnets where your tasks run. We recommend that you use three subnets for production.

Choose subnets

subnet-055ae88627c3f85b8 X  
eu-west-1a

subnet-0f743066101484e0c X  
eu-west-1c

subnet-0ad655648225de94d X  
eu-west-1b

There is no need to worry too much about VPC configuration and subnets. Let's stick with the default settings that come 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

**▼ 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 to avoid managing EC2 instances, which greatly reduces operational complexity. Since Fargate eliminates the need for server management, it is categorized as a Serverless technology.

Amazon Elastic Container Service > Clusters

**All Clusters Info**

Clusters (1)					<input type="button" value="C"/>	<input type="button" value="Create cluster"/>		
<input type="text"/> Search clusters					<input type="button" value="&lt;"/>	<input type="button" value="1"/>	<input type="button" value="&gt;"/>	<input type="button" value=""/>
Cluster	Services	Tasks	CloudWatch monitoring	Capacity provider strategy				
awsfundamentals	0	No tasks running	<input checked="" type="radio"/> Default	No default found				

Once you see the name of your cluster, the creation process is complete. Currently, there are no active tasks or services, but we will address this in the following paragraphs.

## Setting up a Task Definition

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

Here's how to create a new task definition:

1. Set a task definition name.
2. Add container details, including the image URL from a previous step and the port to expose. For our case, the port should be `3000`.
3. It's possible to run multiple containers in a single task definition. This allows for inter-container communication, such as container A talking to container B via `localhost`. All communication stays inside the task. You can also define which containers are essential, meaning they must be healthy to keep the task itself healthy.

Since 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/awsf"/>	<input checked="" type="checkbox"/> 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"/>
<a href="#">Add more port mappings</a>			

Next, we need to configure our environment.

1. For our sample application, we can choose very low vCPU and memory settings. We don't have any requirements for computation since we're only serving a simple application.
2. There's no need to configure a task role. The task role is used in our application only if we're using the AWS-SDK to invoke other AWS services. As we don't integrate with other services, like a DynamoDB table, 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)**

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

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

<b>CPU</b>	<b>Memory</b>
<input type="button" value=".5 vCPU"/> <input type="button" value="▼"/>	<input type="button" value="1 GB"/> <input type="button" value="▼"/>

**► 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 **Next**, you will be directed to the overview page. Clicking **Create** will create your new task definition.

### Establishing and Launching a New Service

An ECS service enables you to run and maintain a specified number of task definition instances 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.

To achieve this, go to your cluster and click 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="text" value="awsfundamentals"/>	<input type="text" 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 will use the basic compute configuration and select the previously created task definition in the deployment configuration. At present, there is only one revision. Each update to the task definition will publish a new version. For instance, if we update the image URL after publishing a new version (= new image tag) to ECR.

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

The screenshot shows the AWS Lambda function configuration interface. In the 'Load balancing' section, the 'Load balancer type' is set to 'Application Load Balancer'. Under 'Create a new load balancer', the 'awsfundamentals' name is specified. The 'Choose container to load balance' dropdown contains 'awsfundamentals 3000:3000'.

**▼ 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. To keep it simple, we'll use HTTP so we don't need an AWS Certificate Manager certificate.

However, the default security group attached to our load balancer doesn't allow any incoming traffic. To allow incoming traffic from the internet, we need to add a new rule that allows traffic from `0.0.0.0/0`. To do this, go to the security groups pane in VPC.

**Remember:** this is just an example. In a real-world scenario, it's recommended to strictly separate resources into security groups that only have the required allow rules they 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> 80	<b>Protocol</b> 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> root	<b>Protocol</b> HTTP
----------------------------------	-------------------------

<b>Health check path</b> <a href="#">Info</a> /	<b>Health check protocol</b> HTTP
--	--------------------------------------

<b>Health check grace period</b> <a href="#">Info</a> 10	seconds
---	---------

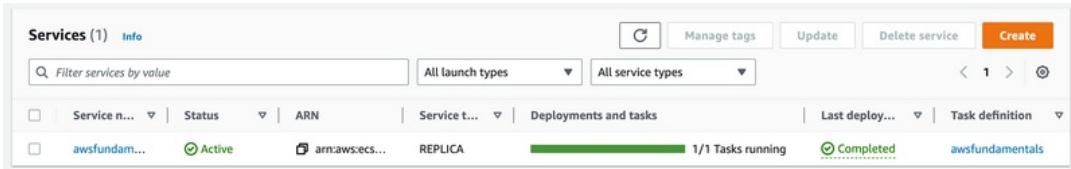
The target group is used to route requests from the load balancer to our ECS tasks. We must specify the protocol for the requests themselves, as well as the protocol and port for the health checks. Health checks regularly call our application to check whether it is in a healthy state. If the health checks consecutively fail, the tasks will not be added to the load balancer's set of routed tasks.

To complete the service creation, click [Create](#).

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

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	<a href="#">View nested</a>
<input type="radio"/> ECS-Console-V2-TaskDefinition-9cc07f35-c39c-478c-a5ee-bc6f84bc1560	<span>CREATE_COMPLETE</span>	2022-12-22 06:58:54 UTC+0100	<a href="#">View nested</a>
<input type="radio"/> Infra-ECS-Cluster-6176d621-8dea-41c2-9dee-dd0de6c43fad	<span>CREATE_COMPLETE</span>	2022-12-22 06:21:35 UTC+0100	<a href="#">View nested</a>

Once the stack finishes the updates, we are done. We can go back to our service overview and wait for ECS to complete its deployment.



Let's verify that our load balancer was created correctly. This process can take several minutes. Navigate to EC2 and go to Load Balancing > Load Balancer Groups. Copy the load balancer's DNS name so we can use it later to submit a request to our application.

Next, go to Load Balancing > Target Groups to view our target groups. You should see the newly created group attached to our ALB. Since we have only configured one task to run at a time, there should be one healthy target.

A screenshot of the AWS EC2 Target Groups console. The top navigation bar shows 'EC2 &gt; Target groups &gt; awsfundamentals'. The main section displays 'awsfundamentals' target group details: Target type (IP), Protocol (HTTP: 80), Protocol version (HTTP1), VPC (vpc-05e559f382af5deda), IP address type (IPv4), Load balancer (awsfundamentals), Total targets (1), Healthy (1), Unhealthy (0), Unused (0), Initial (0), and Draining (0). Below this, tabs for 'Targets', 'Monitoring', 'Health checks', 'Attributes', and 'Tags' are visible. The 'Targets' tab is selected. At the bottom, the 'Registered targets (1)' section shows a table with one entry: IP address 172.31.42.95, Port 3000, Zone eu-west-1b, and Health status healthy. There are buttons for 'Deregister' and 'Register targets'.

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

```
curl http://awsfundamentals-1961464914.eu-west-1.elb.amazonaws.com -vvvv
*   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 have successfully set up our first application with ECS and Fargate.

**Reminder:** As Load Balancers and ECS are charged by the hour, please do not forget to delete all of your resources after you are finished. This can be done by deleting our ECR repository and then 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. Do not be afraid to delete those; there is 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 for maintaining the security of their own data, operating systems, networks, identity, and 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, infrastructure and data.

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 would need for other external repositories is less likely to cause security incidents as there is no way to expose security credentials to unauthorized principals easily.
2. **Use IAM roles to control access to ECS resources and assign 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. **Use 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 provides different deployment strategies, including rolling updates, Blue/Green deployments with CodeDeploy, and external deployments.

The most common strategy is the rolling update, which uses the ECS scheduler to replace currently running tasks with new tasks. This strategy depends on the `minimumHealthyPercent` and `maximumPercent` values, which determine the upper and lower bounds of the number of healthy tasks.

Consider the following service definition example:

- Desired count of 3 containers
- Minimum percentage: 100%
- Maximum percentage: 150%

This means that ECS can only deploy a single new task definition, wait until it's available, and then stop one of the old tasks.

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

Deploying new tasks to your cluster can take several minutes, even though the containers may start quickly. However, you can speed up the process by fine-tuning different settings.

### **ECS Deployment Settings**

In your service definition, you can specify the minimum and maximum number of healthy task percentages that ECS should maintain during task replacement. By reducing the minimum and increasing the parallelism, you could improve deployment times. This is because ECS will be 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 reuse them for subsequent requests, reducing latency. During 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. If you're not relying on real-time connections, 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 for the transition to the active state in the target group and how often these health checks are executed.

By reducing the interval and required number of successful health checks, the target group will mark containers as healthy much faster.

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

The key metrics and events that should be monitored include:

1. **Task and cluster status & events** - Monitor the statuses and events related to your tasks and clusters, such as task failures or cluster scaling events. 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. Use these utilizations 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** - 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 previously, we can determine the CPU and memory utilization of our containers through ECS metrics that are forwarded to CloudWatch. This allows us to determine the load on our cluster and create CloudWatch alerts that trigger scaling actions for our ECS cluster via step scaling policies.

With this policy type, you define how ECS should react to different utilizations (either CPU or memory). For example, we can 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 can stop one task if the CPU utilization is below 40% for 2 minutes.

It is recommended to define a cool-down period that limits the frequency of starting or

stopping tasks. Additionally, you can 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.

For example, we can 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**

Different times of the day will result in different traffic load patterns. 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.

#### **Leveraging AWS Cloud Map to Tackle One of the Major Challenges of a Growing Microservice Ecosystem**

This goes way beyond fundamentals, but due to its importance, we'll quickly cover it here: Service discovery.

With a growing ecosystem, the number of services will also increase. Instead of having a single monolith that includes all features in a single deployment package that can be executed in a single service, we'll have many dedicated microservices.

Microservices architectures, which often utilize lightweight serverless technologies like Docker containers executed on Fargate or Lambda functions, introduce complexity due to their flexibility. A single microservice can be divided into multiple execution units, such as running in Docker containers across multiple tasks and cluster instances in ECS, each with its own unique IP or DNS address.

How can a service's clients locate the appropriate endpoint for the specific version? And that's where the need for service discovery comes in:

1. **Client-based discovery** - Clients connect to a service registry that contains up-to-date information about the service. By using a logical naming scheme, the client can look up the service using a known identifier, and the registry provides one or more DNS names or IP address endpoints where the service is hosted.
2. **Server-based discovery** - Clients connect to a known server-side endpoint, like a load balancer (e.g. ELB or ALB, as we've learned in the hands-on part), which resolves the request to a healthy, running instance of the service.

Both approaches have their own advantages and disadvantages.

AWS Cloud Map provides a managed, highly available solution out of the box, so you don't need to build your own client-based discovery tool. Simply register your application and its running instances, and then use either the AWS Cloud Map API or DNS lookup to resolve the service's name to a current, active endpoint.

### The Endless Use Cases of Container Services

AWS ECS is a core building block in nearly any enterprise that works with AWS. Its strength lies in its ability to simplify running containers, which are ubiquitous in today's software landscape. As a result, any use case you can think of has likely already been implemented with ECS.

Instead of providing a lengthy list of examples for various applications that use ECS, we will explore something that software engineers in the cloud area are almost certain to encounter.

#### 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 sphere. As this book aims to provide as many examples as possible to help you get started, this scenario must be included. Many companies are currently in the process of migrating their on-premises applications to the cloud, with ECS often being the target core service. This is due to several reasons, including:

- **Software that is already in production is likely to stay** - if software has been shipped to production and is being used by the company, customers, or other third parties, it will likely remain there for a long time. Replacing existing software with new versions or a completely new product is challenging, time-consuming, and risky for companies. Therefore, it is often only tackled if there is no other option. With the benefits of the cloud, most companies choose to migrate their core to mature and battle-tested cloud services

like ECS.

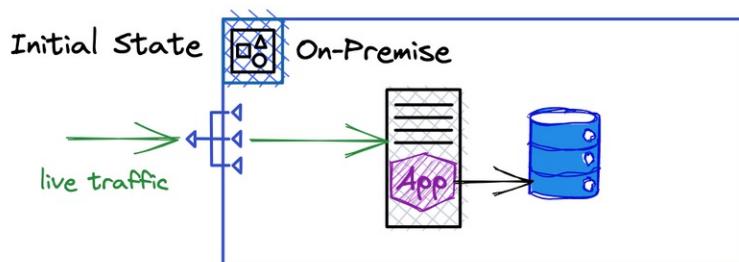
- **On-premises applications are often monolithic** - software development processes from previous decades often followed the classic approach of requirements analysis, planning, development, testing, and shipping the software as a whole product, resulting in large software products in just one or a few services or applications. Containerizing these applications to run in ECS is relatively simple compared to restructuring their entire 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 because there is 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.

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

But how do you do this? There is no single answer, but many. We will 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

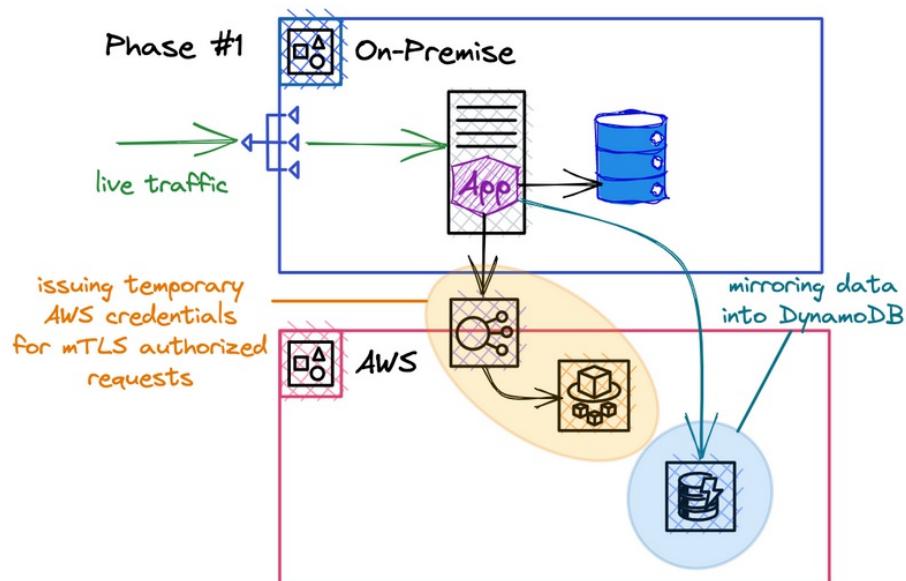
Initially, the application was a simple Java-based monolithic application running on on-premise servers with a directory serving as the main data storage. The application was already running in production, and the requirement was to migrate it with as little downtime as possible.



Our migration plan to AWS involved lifting the application in multiple steps without causing any downtime:

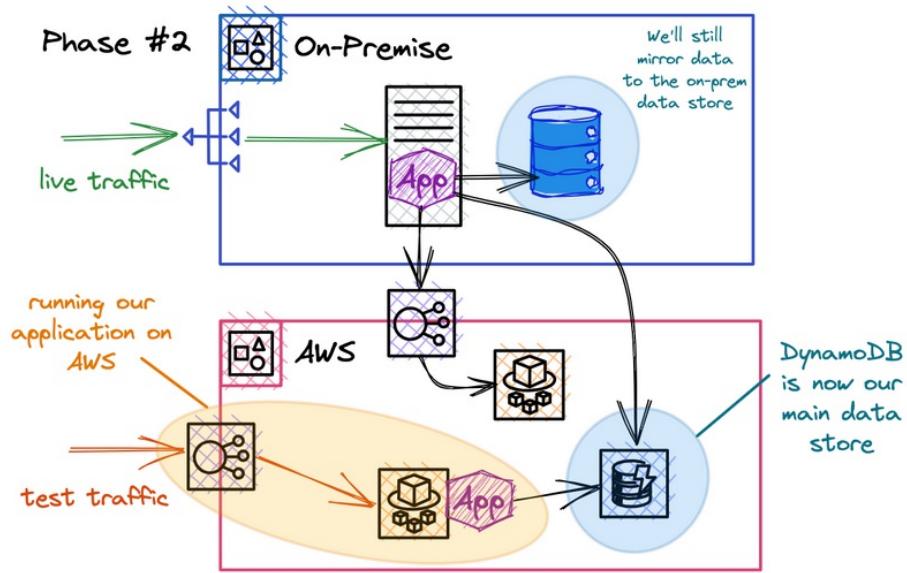
1. Creating an on-the-fly copy of all on-premises data in DynamoDB via the running application.
2. Containerizing the application and running it on a new ECS cluster. Additionally, we now use DynamoDB as the primary data storage and only mirror data to the on-premises directory.
3. Completely removing the on-premise directory and performing integration testing on the application that now runs on ECS and can access live data.
4. Switching live traffic to the ECS cluster and removing the on-premises application.

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



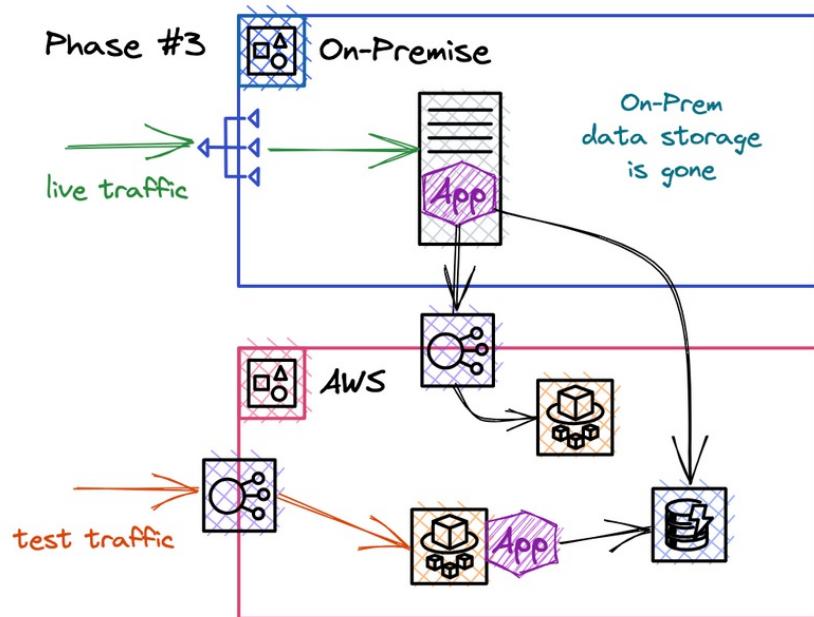
During the first phase, we needed the ability to retrieve temporary AWS credentials for secure writing to DynamoDB. To achieve this, we exposed a Fargate task that supports mutual authentication using a certificate. Upon receiving a valid request, it returns temporary AWS credentials for a role that only has write access to our DynamoDB tables.

We then abstracted our data into layers within the application, allowing us to use as many data layers as we need. The root data layer always returns the source of truth, while secondary data layers receive a copy that is forcefully overwritten. To ensure data is in sync when we read and write, we check for any observed issues and log them. This process involved multiple iterations, as we not only moved from NoSQL to NoSQL, but also switched database solutions from a directory to DynamoDB.



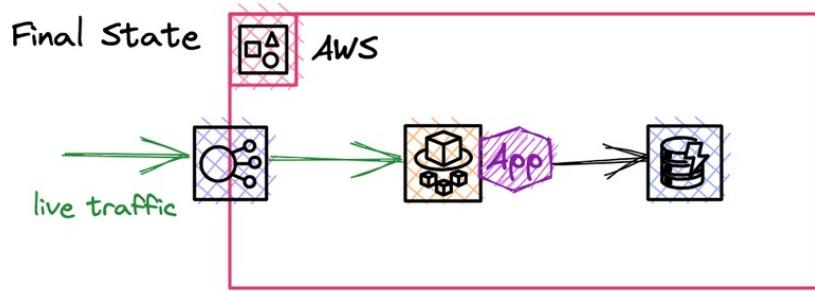
The on-premise application now writes data to both the on-premise storage and DynamoDB. The application has been containerized and an ECS cluster has been built for our service, which accesses DynamoDB. With this setup, we have a copy of our application that can run on AWS and be tested with real data without affecting customers.

The order of the data layers has been switched, with DynamoDB now being the source of truth for the data. The data is only copied to the on-premise storage as a backup.



At the next stage, we have finally eliminated on-premise data storage. Now, we can concentrate

on testing the cloud application by conducting integration tests, load tests, and disaster recovery tests. This will ensure that everything runs as smoothly as possible.



Finally, all that was left to do was switch the DNS records to our cluster on AWS. A few days prior, we had reduced the Time-To-Live for the main record to prevent clients from caching the record for more than 60 seconds.

The switch was executed without any outage, not even for a second, and no one even noticed the change. In case anything went wrong, we always had the option to switch the DNS records back to the on-premise cluster and resolve any issues.

### Tips & Tricks for the Real World

ESC is fantastic, and we enjoy working with it every day. When used with Fargate, a well-structured setup often results in worry-free 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. For example, you can keep only 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 hardcode environment-specific strings into your application, but pass them as environment variables. By doing that, you can use the exact same Docker image for every stage. The only thing that changes is 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. Otherwise, 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 will too.



**AWS Lambda**

# Using Lambda to Run Code without Worrying about Infrastructure

## Introduction

Amazon EC2 enables you to stop managing physical servers and focus on virtual machines. With Lambda, launched in 2014, AWS took this one step further by completely removing customers' liabilities for the underlying infrastructure.

The only thing you need to 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 have never worked with Lambda before, this may be the **most important chapter** as the included information may not seem very intuitive at first.

Let's take a look at how Lambda works under the hood and what trade-offs we have to face due to its on-demand provisioning of infrastructure. Also, let's see what measures we can use to slightly mitigate the limitations we face.

#### **Micro-Containers Running Your Code in the Background**

One thing that is often missed or misunderstood is that serverless does not mean that there are no servers. They are simply abstracted away and opaque to the application developer.

When an incoming request is made to your Lambda function, AWS will either:

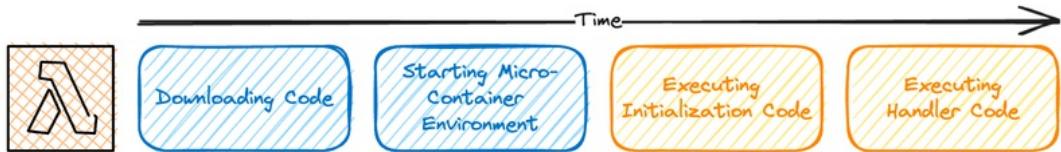
1. **Internally provision a micro-container** and deploy your code into it, or
2. **Reuse an existing container** that has not been de-provisioned yet and is not already busy processing another request.

As you have 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 closer look at the first scenario.

It is not surprising that there is a certain amount of bootstrapping time required before our code can be executed. This process of preparing Lambda's environment for code execution is known as a **cold start**.



Lambda needs to download your function's code and start a new micro-container environment which will receive the downloaded code. Afterward, the global code of your function will run. This includes everything **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. Think of it as a (very) volatile and unpredictable cache.

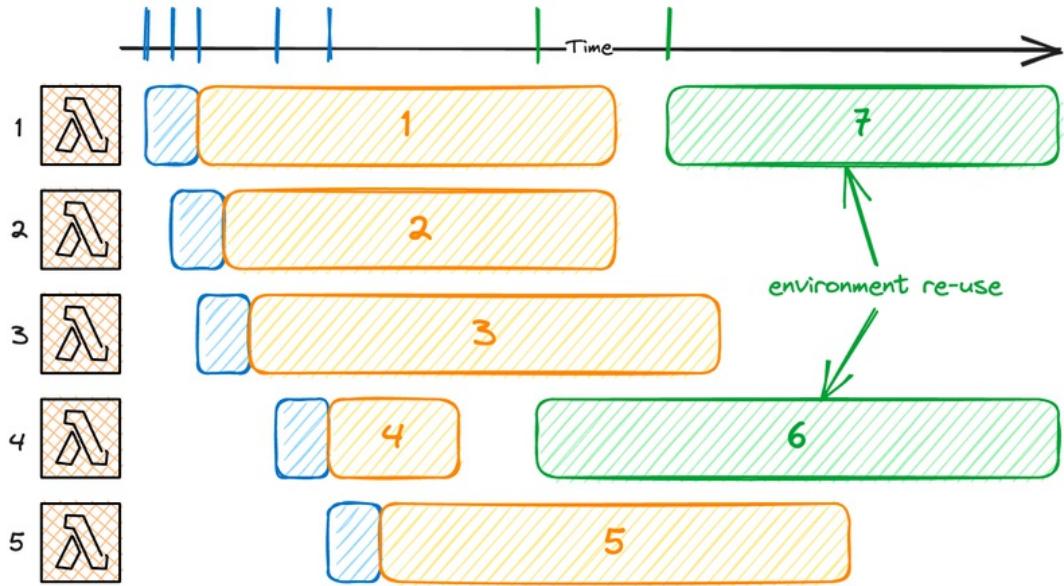
Lastly, the main code inside your handler function is executed.

**Worth noting:** Although 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, such as 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 much slower in Lambda than on Fargate, as the cold starts will significantly contribute to those.

#### A Micro-Container Can Only Serve One Request at a Time

Each provisioned Lambda micro-container can process only one request at a time. This means that even if there are multiple execution environments already provisioned for a single Lambda, a **cold start may occur** if all of them are currently busy handling requests.



In the invocation scenario described above, it can be observed that five Lambda micro-containers were initially started in the first phase because each consecutive request came in before another container had finished.

The first re-use of a container occurred only at request number six when micro-container number four finished its previous request.

This makes it difficult to reduce cold starts, especially if your application landscape is composed of many different Lambda functions, which may even require mutual synchronous invocations.

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

If we look 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 executing this code can create global variables that remain in memory, persisting

across multiple executions of the same micro-container. They are only lost after the Lambda environment is torn down.

In our example, the results of our core framework's bootstrap and the start time are kept in memory. They will only disappear when AWS de-provisions our function's container.

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

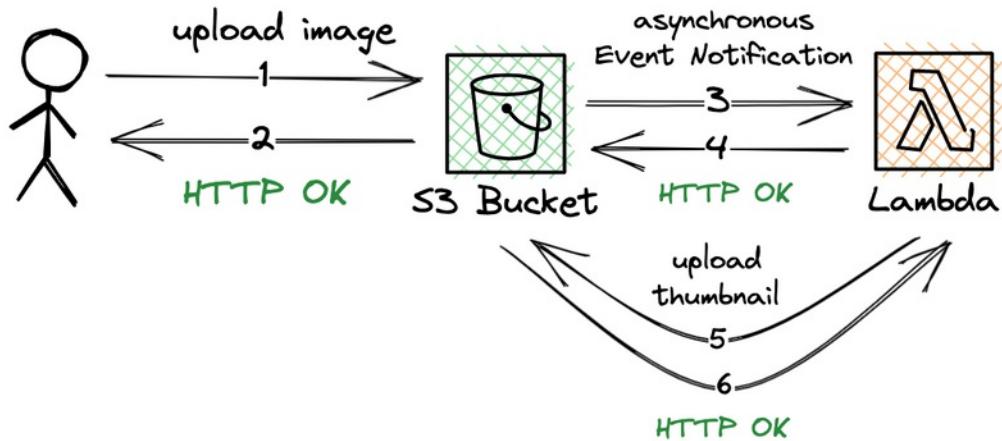
Take advantage of this and bootstrap as much as possible outside the handler function, keeping a global context while your function runs.

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

### **Your Functions Can Be Invoked Synchronously and Asynchronously**

There are two methods to invoke your function's 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, such as 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.

```

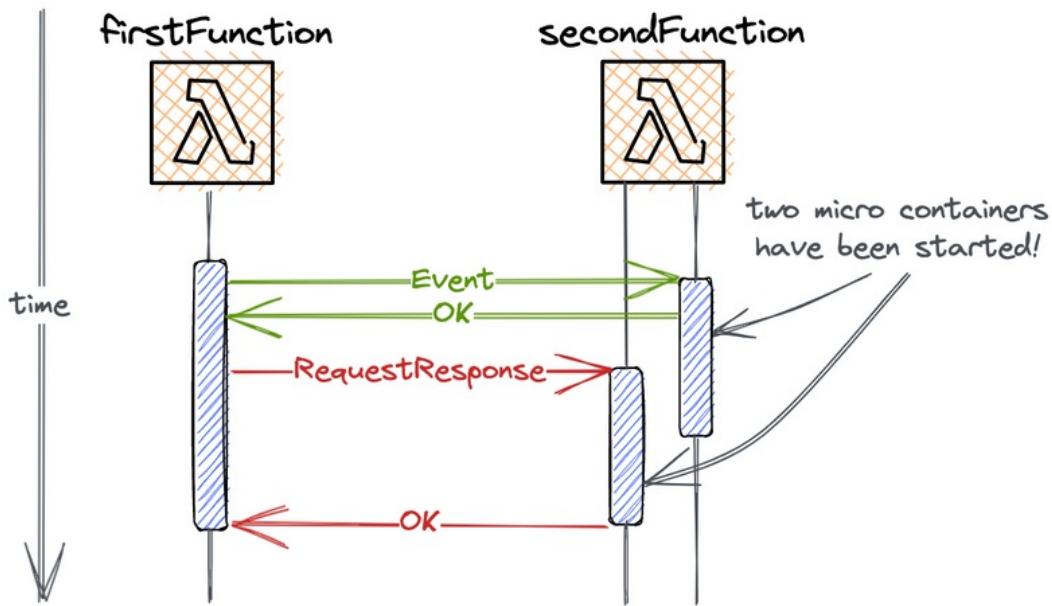
const AWS = require('aws-sdk');
const lambda = new AWS.Lambda();

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 the function `firstFunction`, we can see how both functions 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 occurs and therefore starts another micro-container while 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. Some can be changed afterward, 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 many runtimes at Lambda, including Node.js, Python, Java, Ruby, and Go. Besides deploying your function as a ZIP file, you have the option to provide a Docker container image. It's also possible to bring your own runtime to execute any language by setting the function's 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 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 not only determines the available memory but also the assigned vCPUs. This means that higher settings result in higher computation speeds. You will be billed for GB seconds, so more memory will result in paying more per executed millisecond. However, this does not necessarily mean that your bill will increase linearly with the assigned memory, as more memory and therefore vCPUs will decrease the function's execution time.

#### **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 that 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

The screenshot shows the 'Edit environment variables' page. At the top, there's a section titled 'Environment variables' with a descriptive text: '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.' Below this, there's a table showing two environment variables:

Key	Value	Action
ENVIRONMENT	development	Remove
APP_PREFIX	awsfundamentals	Remove

Below the table is a button labeled 'Add environment variable'. Further down, there's a link to 'Encryption configuration'. At the bottom right, there are 'Cancel' and 'Save' buttons.

Each variable is stored within the function's environment and can be accessed from your code.

- In Node.js, you can use the `process.env` object.
- in Java, you can use the `System.getenv()` method.

- in Python, you can use `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 include enhanced security requirements, such as restricting outbound traffic from your functions.

Moreover, 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.

However, there are considerations when using VPCs. Even though AWS has improved this drastically with the integration of AWS Hyperplane, VPC integration will increase your function's cold start times. Additionally, VPC integration can increase 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 many other services via triggers, meaning you can launch Lambda functions based on events that are fired from other services.

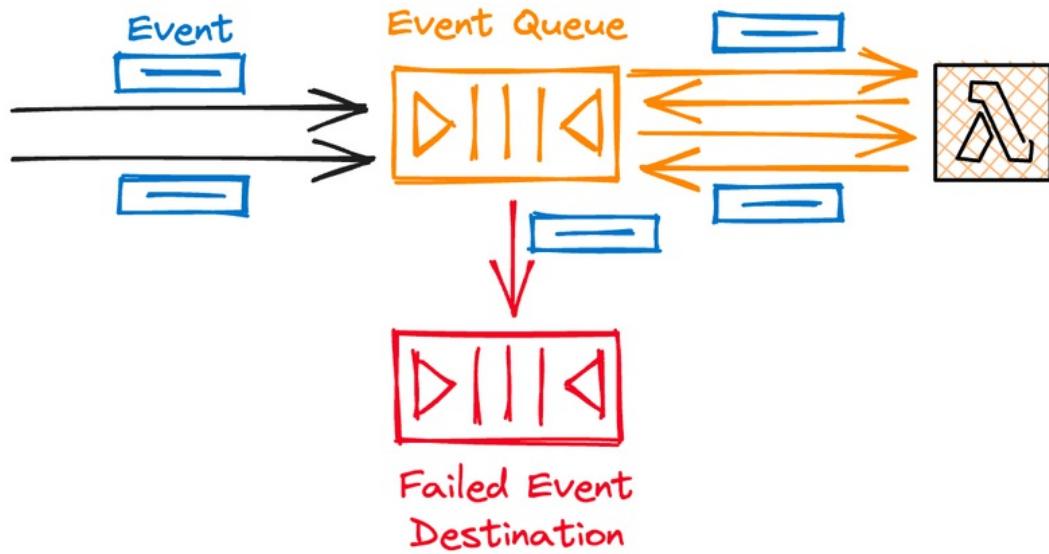
Prominent examples include:

- 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 significant feature for building reliable event-driven architectures that can also recover in case of outages and errors.

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

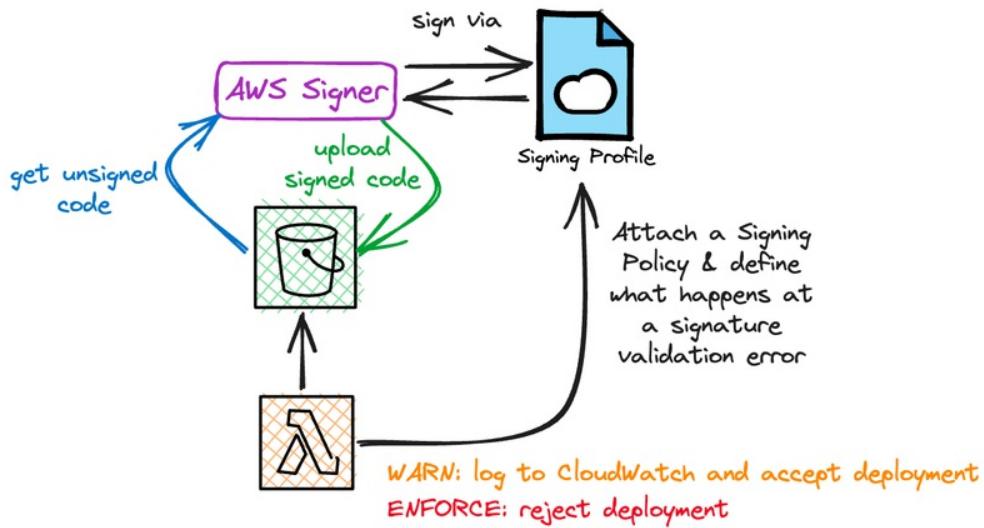
The benefit of not having to wait for responses on asynchronous invocation is also the drawback: you can't immediately determine if the execution resulted in any errors. That's why Lambda offers destinations, so you can react to successful or faulty executions.



In our example, failed invocations or invocations that cannot 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 that your code was never tampered with? With AWS Signer and code signing, you can create signing profiles to enforce that only code from 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 running in parallel. This allows you to test code changes without affecting the currently stable version on your staging environment.

When you publish a new version, you will get a version number that can be used to invoke your function via the qualified ARN:

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

In addition, you can create an alias for each version of your function. An alias acts as a pointer to your function.

The benefit of using aliases instead of qualified ARNs is that you can use them with event source mappings without the need to update each mapping after publishing a new version. You only need to update a single resource: the alias itself.

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

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

Both can improve the performance and scalability of your functions, but they serve different purposes. Reserved concurrency ensures that a certain number of instances of your function are always available to handle requests, while provisioned concurrency keeps instances pre-

warmed in anticipation of traffic.

### **Reserved Concurrency for Guaranteeing a Function's Concurrency Capacity**

The default concurrent execution limit for a Lambda function in an account is 1000. For new accounts, this limit may be even lower. This means that more than 1000 Lambda functions cannot be executed in parallel. It also implies that a large number of functions can run in parallel, especially if you use fan-out mechanisms (using a function to trigger other functions which will then trigger more functions) to trigger your Lambda functions.

To restrict the maximum number of parallel executions, use reserved concurrency for your function. Reserved concurrency is 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 doesn't declare a value for reserved concurrency, it uses the unreserved concurrency capacity left in your account, which could be completely consumed under certain circumstances.

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

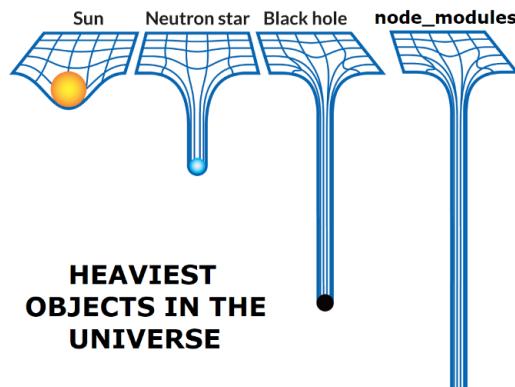
Regardless of the strategies you're using to keep Lambda functions warm via strategic health checks or your level of permanent requests per second, your micro-containers **will be de-provisioned** at some point.

To overcome this, use provisioned capacity. AWS keeps a certain number of Lambda environments provisioned so they're 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**

When building extensive business logic, using existing libraries can often be more efficient than reinventing the wheel. This sometimes results in having more code in dependencies than in actual self-implemented business logic, which can slow down packaging and deployments.



In order to include dependencies in your Lambda function, you must package them individually for each function, even if multiple functions rely on the same packages.

Lambda Layers provide a solution to this problem. You can create a versioned Layer that includes the dependencies needed for your Lambda function and attach one or several functions to the same layer. This way, all functions will have access to the included dependencies.

When deploying new functions, you only need to package your own code, which drastically reduces packaging and deployment times, as your code likely only takes up a few kilobytes.



There is a size limitation for deployment packages, which includes the size of referenced layers. The limit is 50 MB for zipped files and direct upload, and 250 MB for the unzipped archive.

Make sure to package your dependencies in the correct folder. For instance, Lambda expects your `node_modules` to be located inside the top-level folder `nodejs`.

### **Monitoring Your Functions with CloudWatch to Detect Issues**

Like with other services, Lambda integrates with CloudWatch by default and sends a lot of useful metrics without needing 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 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 errors.
- **Throttles** - The number of times 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 sent 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 no longer in use.

### **Going into Practice - Creating Our First Serverless Project**

We have covered the most important fundamentals of Lambda. Now, let's create our first small Lambda project.

We will divide this into four major steps:

1. **Creating a simple Node.js function.** We will create a small function, adapt and deploy code changes within the AWS management console, and test our function via our own test events.
2. **Adding external dependencies.** We will 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

To create a simple Node.js function, go to the AWS Lambda console and click on `Create function`. Define a function name, select your target architecture, and choose which runtime you want to use. For this example, we will use Node.js.

The screenshot shows the 'Create function' wizard in the AWS Lambda console. The first section, 'Basic information', is visible. It includes fields for 'Function name' (set to 'awsfundamentals'), 'Runtime' (set to 'Node.js 18.x'), and 'Architecture' (set to 'arm64'). Other options like 'Author from scratch', 'Use a blueprint', and 'Container image' are also shown. A note at the top says 'AWS Serverless Application Repository applications have moved to [Create application](#)'.

Option	Description
<input checked="" type="radio"/> Author from scratch	Start with a simple Hello World example.
<input type="radio"/> Use a blueprint	Build a Lambda application from sample code and configuration presets for common use cases.
<input type="radio"/> 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`

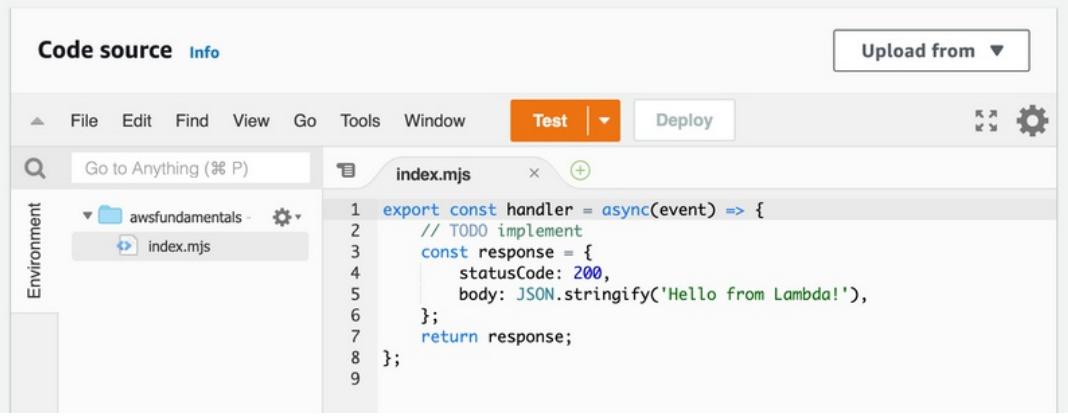
**Runtime** [Info](#)  
Choose the language to use to write your function. Note that the console code editor supports only Node.js, Python, and Ruby.  
`Node.js 18.x`

**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, as well as to save and deploy updates to our function.



The screenshot shows the AWS Lambda Code source editor interface. At the top, there's a navigation bar with File, Edit, Find, View, Go, Tools, Window, a Test button (which is orange), Deploy, and a gear icon. Below the navigation bar is a search bar labeled "Go to Anything (% P)". To the left, there's a sidebar titled "Environment" with a folder named "awsfundamentals" containing an "index.mjs" file. The main area displays the contents of "index.mjs":

```
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.

**Configure test event**

A test event is a JSON object that mocks the structure of requests emitted by AWS services to invoke a Lambda function. Use it to see the function's invocation result.

To invoke your function without saving an event, configure the JSON event, then choose Test.

**Test event action**

Create new event     Edit saved event

**Event name**

test-event

Maximum of 25 characters consisting of letters, numbers, dots, hyphens and underscores.

**Event sharing settings**

Private  
This event is only available in the Lambda console and to the event creator. You can configure a total of 10. [Learn more](#)

Shareable  
This event is available to IAM users within the same account who have permissions to access and use shareable events. [Learn more](#)

**Template - optional**

hello-world

**Event JSON**

Format JSON

```

1 {
2   "message": "Hello World!"
3 }
4

```

Let's modify our function to return the message passed in the function's response. After clicking `Deploy`, the changes will be deployed to our function. Then, we can invoke our function with the test event.

**Code source** [Info](#)

[Upload from](#)

**index.mjs**

```

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 Test interface. At the top, there's a toolbar with File, Edit, Find, View, Go, Tools, Window, Test, Deploy, and a gear icon. Below the toolbar, there's a search bar labeled "Go to Anything (% P)" and a sidebar titled "Environment" which lists "awsfundamentals" and "index.mjs". The main area displays the "Execution results" for a test event named "test-event". The status is "Succeeded", memory usage is 63 MB, and time is 8.20 ms. The response body is shown as a JSON object: { "statusCode": 200, "body": "\"Our message: Hello World!\""} . Function logs show the START, END, and REPORT events with the same RequestId. The Request ID is highlighted as 68edb45f-6d7c-4e6b-be32-5bf0694b8de7.

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

### Adding External Dependencies

Now, let's continue and add some external dependencies. To do this, we will initialize a new project using `npm` and install Axios (a powerful HTTP client library). We will also create 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 by adding a new HTTP call via Axios to <https://ipinfo.io/ip> in order to obtain 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 the `node_modules` folder into a ZIP archive. Afterward, we can upload the archive to our Lambda function via `Upload from > .zip file.`

```
zip dist.zip .
```

After uploading, you can execute it again by clicking on the test button. The response should resemble the following:

```
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
```

Although our function is currently small because it only includes Axios, its size can quickly increase. Once it reaches a certain threshold, the code becomes impossible to view or edit in the Lambda console. Additionally, we upload many kilobytes for every function upload, even though there may only be changes to our code.

To solve this issue, we will extract our dependencies into a Lambda Layer in the next part.

## Externalizing Dependencies into a Layer

Creating a Lambda Layer has two significant benefits:

- The size of the deployment unit required for function updates is reduced.
- A layer can be shared with multiple Lambda functions that share the same dependencies.

The process is quick and simple. We only need to include our dependencies in our layer's zip file in an expected directory format. In the case of Node.js, the `node_modules` folder must 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](#)

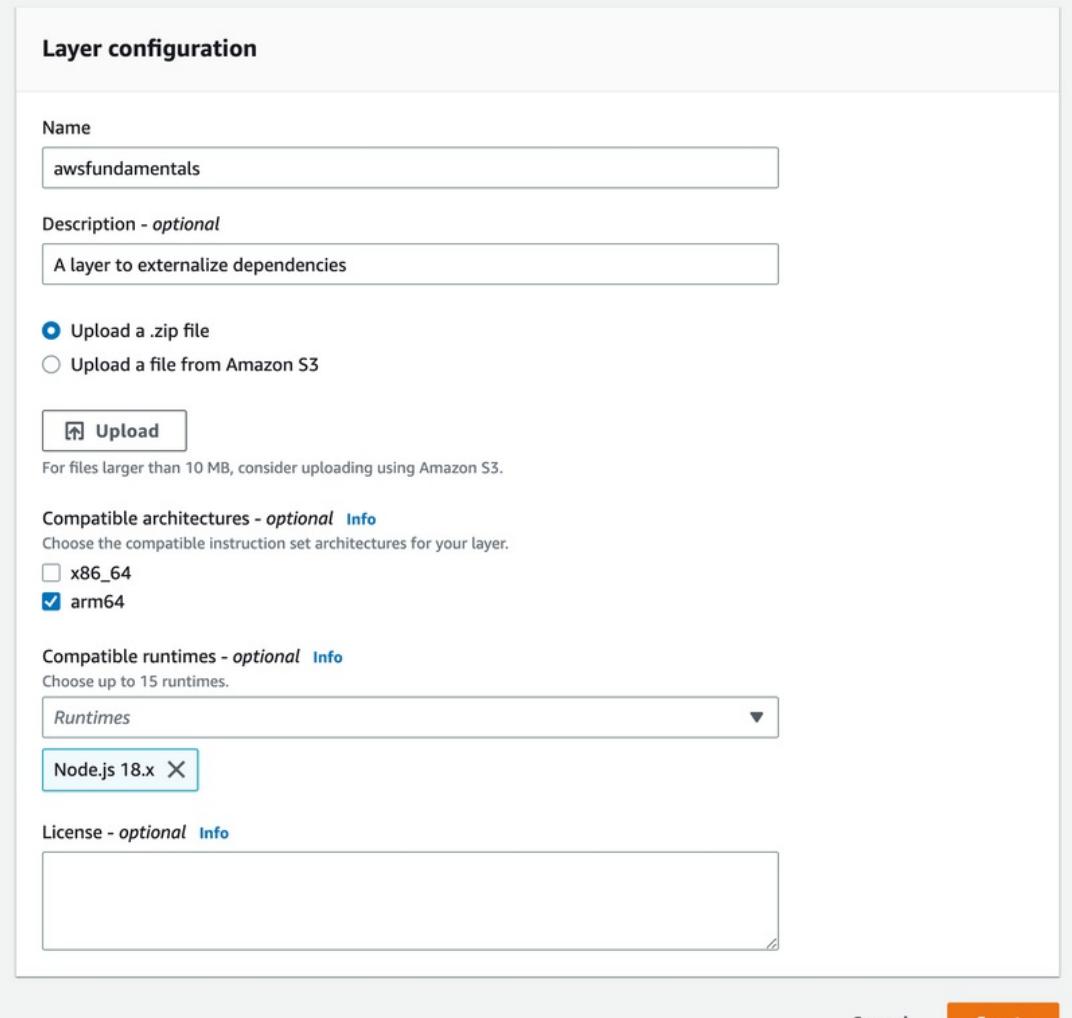
For files larger than 10 MB, consider uploading using Amazon S3.

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](#)

[Cancel](#) [Create](#)



After clicking the `Upload` button, select the `layers.zip` file that we created, and finally create the layer by clicking `Create`. Our Lambda Layer will then be ready to use.

Let's go back to our function and scroll down to the layers overview to connect our new layer. Click `Add a layer` to do so.

## 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.

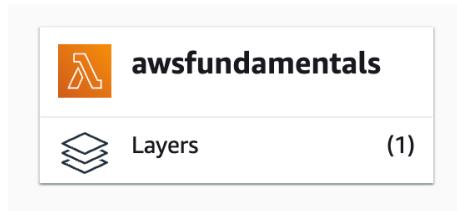
**Custom layers**  
Layers created by your AWS account or organization that are compatible with your function's runtime.

awsfundamentals ▾

Version  
1 ▾

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

Select custom layers and choose our previously created layer. After clicking `Add`, it will take a few seconds to update the function. Afterwards, 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 by right-clicking and selecting `delete`, as we don't need it anymore. It will be provided via our Lambda Layer.

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



The third achievement has been unlocked. Let's take on the 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, head back to the functions overview and click on [Create function](#).

To invoke our function, we need two things:

1. Our first function must have the `lambda:InvokeFunction` permission.
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. By clicking on the link, you'll be taken to IAM where we can edit the policy.

A screenshot of the AWS Lambda Configuration tab. The tab navigation includes 'Code', 'Test', 'Monitor', 'Configuration' (which is active and highlighted in blue), 'Aliases', and 'Versions'. On the left, a sidebar lists 'General configuration', 'Triggers', 'Permissions' (which is also highlighted in blue), and 'Destinations'. On the right, under the 'Execution role' section, it shows 'Role name: awsfundamentals-role-ifd5zmc' with a blue link.

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 Lambda policy editor. At the top, there's a header with 'Lambda (1 action)' and 'Clone | Remove'. Below it, under 'Service', is 'Lambda'. Under 'Actions', 'Write' is selected, and 'InvokeFunction' is listed. Under 'Resources', 'Specific' is selected, and a single resource ARN is shown: 'arn:aws:lambda:eu-west-1:157088858309:function:awsfundamentals-lambda'. There's also an 'EDIT' button and 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
```

To upload a new version of the layer, go to your layer and click the `Create version` button. Then, return to the initial function and switch to the new version (2).

The screenshot shows the 'Edit layers' interface. At the top, there's a 'Function runtime settings' section with 'Runtime' set to 'Node.js 18.x' and 'Architecture' set to 'arm64'. Below this is a 'Layers' table with one row. The row has a 'Merge order' column with '1', a 'Name' column with 'awsfundamentals', and a 'Layer version' column with '2'. There are buttons for 'Merge earlier', 'Merge later', and 'Remove'. 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 have included some waiting periods before the function returns, so that we can clearly observe the impact of different invocation types. By default, the waiting period is set to 10 seconds, but you can also pass a different value via the incoming event.

Now, let's go back to our initial function and add the invocation part. For the time being, we will 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`,
};
}

```

Deploy the update and invoke our function.

```

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

Function Logs
START RequestId: 24b0dca3-52b0-4026-afa1-73a143df8e09 Version: $LATEST
2022-12-23T07:57:26.010Z 24b0dca3-52b0-4026-afa1-73a143df8e09
Task timed out after 3.01 seconds

```

Well, that's not what we expected. However, upon examining our function's configuration, it appears that the default timeout for Lambda is 3 seconds. Since we wait for 5 seconds until our second function finishes its execution, both functions will time out. To resolve this, we need to adjust the timeout setting. Go to [Configuration > General Configuration](#) and set the timeout 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 Triggers, Permissions, Destinations, and Function URL. The main content area is titled 'General configuration' and contains fields for Description (empty), Memory (1024 MB), Ephemeral storage (512 MB), Timeout (0 min 10 sec), and SnapStart (None). An 'Edit' button is located in the top right corner of this section.

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 redeploying the function, we will see in the next invocation that the execution time has significantly decreased.

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

Now, our initial function doesn't wait for the second one to finish executing. This results in faster execution, but we can't be sure if the second function finished executing without errors.

We've covered a lot in this small project, which is a great starting point for further experimentation 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 can configure the authentication type to `NONE` to allow public, unauthenticated invocations.

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

### **Attaching Shared Network Storage with EFS**

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

For a durable storage solution, you can either choose 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 and 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 are 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 will get into detail about this in the upcoming chapter about CloudFront - we just wanted to include this here for the sake 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 are only paying for the actual time at which your code is executed.

What you will 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 us 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 GB → 800,000 GB-seconds → **9.2 days** of execution
- 1.0 GB Memory → 400,000 / 1.0 GB → 400,000 GB-seconds → **~4.6 days** of execution
- 10 GB Memory → 400,000 / 10 GB → 40,000 GB-seconds → **~0.5 days** of execution

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

### **Recursion Protection to Avoid Exploding Costs on Accidents**

A Lambda function has the ability to invoke itself, which can lead to endless recursion if there is no or insufficient cancellation conditions. This can result in unexpected charges and concurrency issues.

To prevent this, Lambda recently introduced a feature to detect and stop recursive loops shortly after they occur. This feature uses AWS X-Ray tracing headers to track the number of times an event has invoked the function. If a function is invoked more than 16 times in the

same chain of requests, Lambda stops the next invocation and notifies the user via the health dashboard and email.

Recursive loop detection is enabled by default for all AWS customers and supports certain AWS services such as Amazon SQS and Amazon SNS. The article also provides guidance on how to respond to recursive loop detection notifications and prevent further loops.

### **Speeding Up Cold Starts with SnapStart**

SnapStart for Java on Lambda can enhance the startup performance of latency-sensitive applications by up to ten times at no additional cost, usually without requiring any modifications to your function code. As we've learned, the primary factor contributing to startup latency is the time Lambda takes to initialize the function. This includes loading the function's code, launching the runtime, and initializing the function code.

SnapStart allows Lambda to initialize your function at the time you **publish a function version**. Lambda captures a snapshot of the memory and disk state of the initialized execution environment, encrypts the snapshot, and stores it for rapid access.

When you first invoke the function version, and as the invocations scale, Lambda launches new execution environments from the stored snapshot instead of starting them from scratch, thereby reducing startup latency.

Currently, this feature is only available for the Java runtime. This makes sense, as Java has one of the slowest cold start times. Lightweight scripting languages like JavaScript do not suffer from this kind of latency during cold starts.

### **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 cannot be changed unless AWS updates its policies. Let us have a look at the most important ones as it is likely that you will 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 is worth checking back with the current state at AWS Quotas.

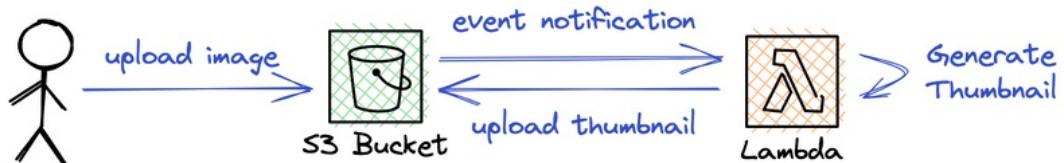
## A Deep Dive into Great Lambda Use Cases

Lambda is the most flexible service of all AWS offerings due to its on-demand pricing, low entry barrier, and ease of use. There are almost no limitations to what you can use Lambda for. Additionally, Lambda natively integrates with many other services, often without requiring much or any glue code.

The following list provides just a few examples of the many use cases for which Lambda is known. It is possible to write about simple or extraordinary use cases for weeks or months!

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

Traditional approaches to video or image processing involve uploading files to storage and having a server regularly pull for newly created files. This approach can result in idle server times and unnecessary costs during periods of low traffic.

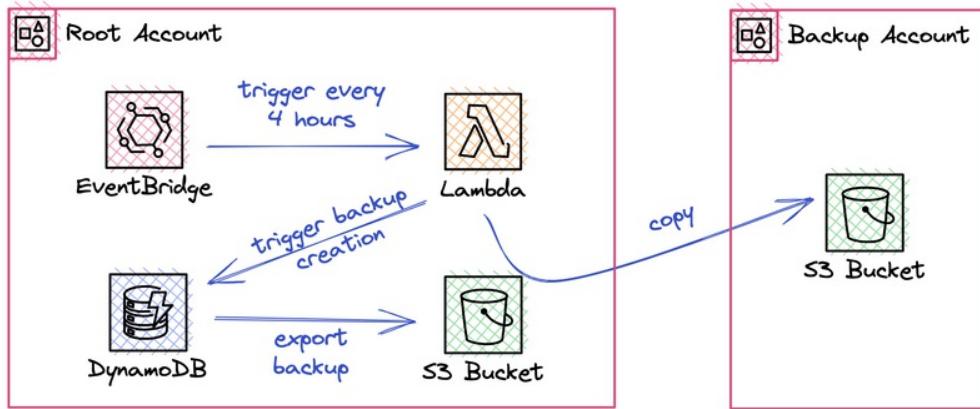


Lambda can provide a completely different approach by allowing you to put video conversion or thumbnail generation in its hands, while uploading files to S3. You can attach Lambda invocations to S3 object lifecycle events such as *ObjectCreated*. This means that new files will automatically trigger your function with an event that contains all necessary information about the lifecycle event and file.

With Lambda, you will only be billed for computations, so you won't pay for any idle time.

## 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 ensure redundancy and high security.

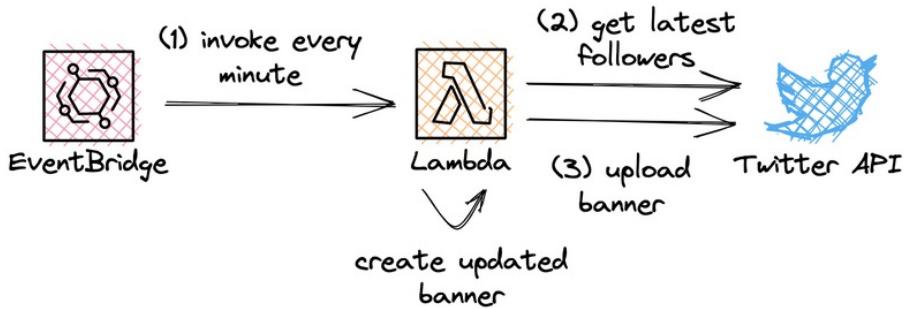


When looking at the previous use case for image and video processing, you can also synchronize files directly based on lifecycle events and event notifications.

## 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, with many blog posts written about it, is to automatically update your Twitter banner based on recent followers, recently published blog posts, or any other information you want to display 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 rely 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 do 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 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 heavily influence performance and perceived satisfaction with applications. That is why we want to deep-dive into strategies to mitigate or reduce them.

There are many tricks and best practices to reduce cold start times and 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 requires 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 on 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 Are the Right Fit

As mentioned in the starting chapters, we believe cloud-native is the future. This future heavily revolves 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.



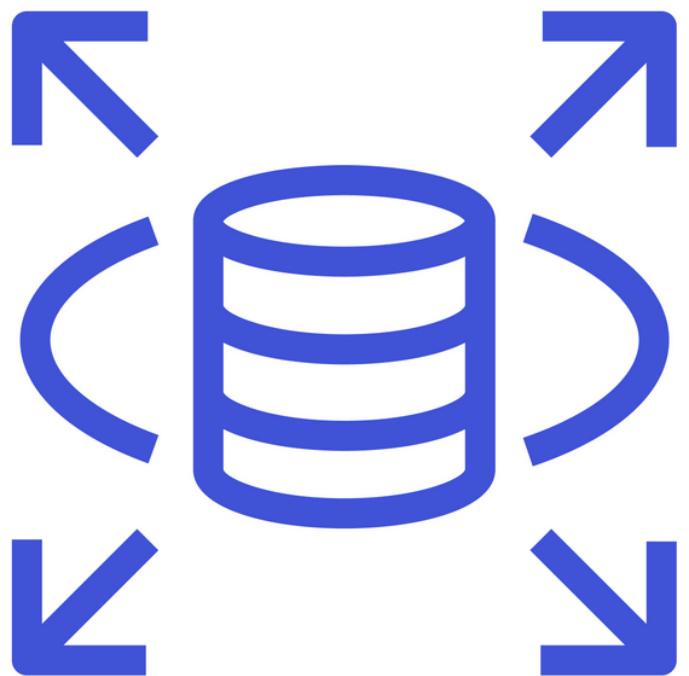
# Database & Storage

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.

In previous chapters, we discussed handing over as much responsibility as possible for infrastructure management. Amazon **RDS**, **Simple Storage Service (S3)**, and **DynamoDB** are great options for building web applications because they are managed, mature, and offer high resiliency and availability. They are also beginner-friendly and require no previous knowledge.

**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. **S3** is designed for storing large amounts of unstructured data, such as images, videos, and documents. It is highly durable and scalable, with data stored across multiple devices in multiple locations to ensure availability. 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 highly available, highly scalable, and fully managed, making it easy to build highly available web applications. They also support 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 just a few clicks. You don't need to manage the infrastructure or operating system.

It is a very popular service because it reduces the effort 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:

- Operating System configuration
- System updates
- Security
- Network configuration

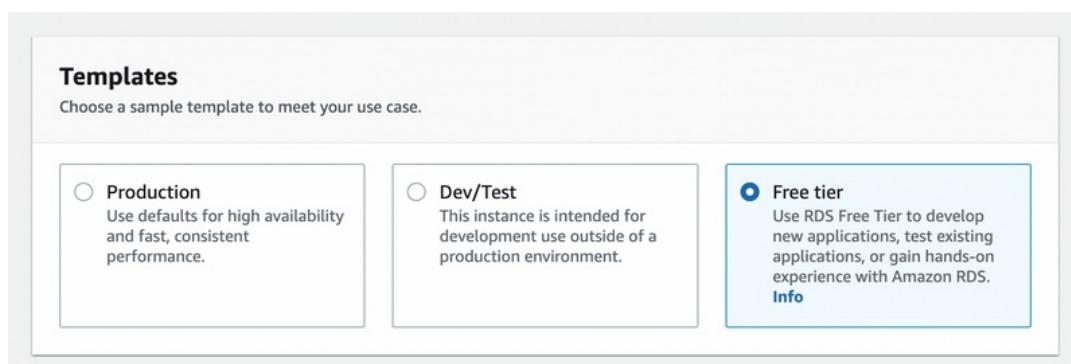
...and much more. With RDS, this is much easier.

You set up your database, receive a connection string, and you're 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 that RDS is charged by the minute. You can choose the free tier template, and AWS will give you the option to 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 offers different engines to meet various requirements.

## Database Engines



Postgres



MySQL



Aurora



Maria DB



Oracle



MS SQL



These include:

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

One thing to keep in mind about these engines is the exact version that AWS supports. The engine itself is an official release, but AWS often supports only certain versions of the engine.

## 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 introduce AWS Data Centers in general.

### One Data Center is Available in One Region and One Region Has Multiple Availability Zones

AWS separates its data centers into regions, and inside each 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 find 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.

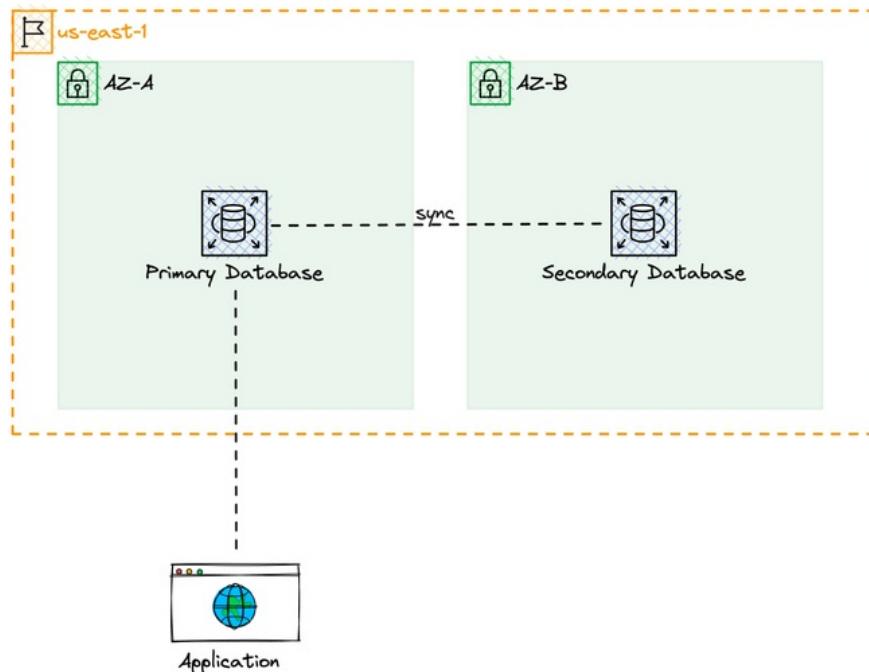
Single DB instance

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. Suppose 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 and writing access. This is without any data loss or manual intervention. All of that is managed by AWS.

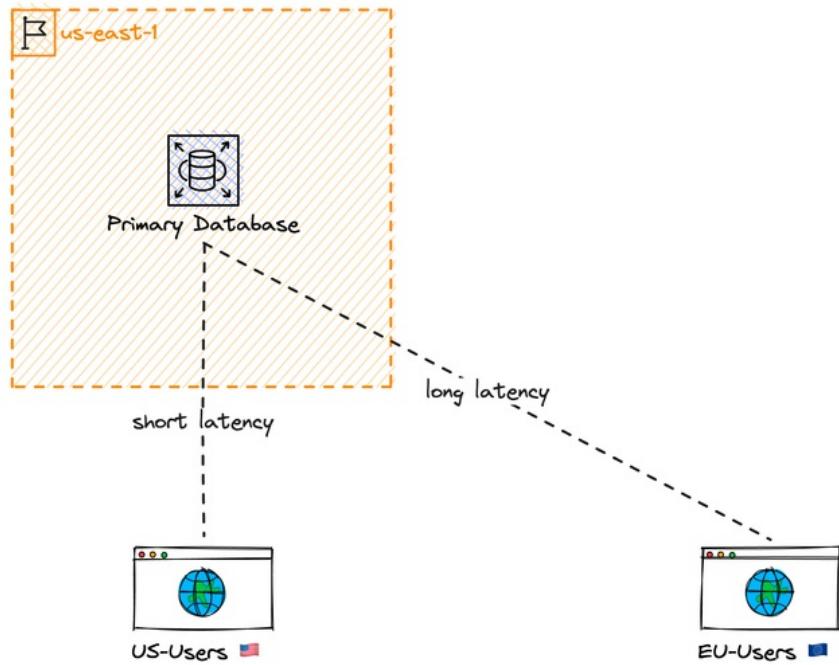
Achieving this behavior a few years back was a huge effort. In AWS, it is simply clicking a button.

The behavior and startup times depend on the database engine you use. The best thing to do is, as **always, test things!** Don't rely on things to work out. Instead, test them out. AWS has a good comparison table to see how different engines behave and how to choose the right one.

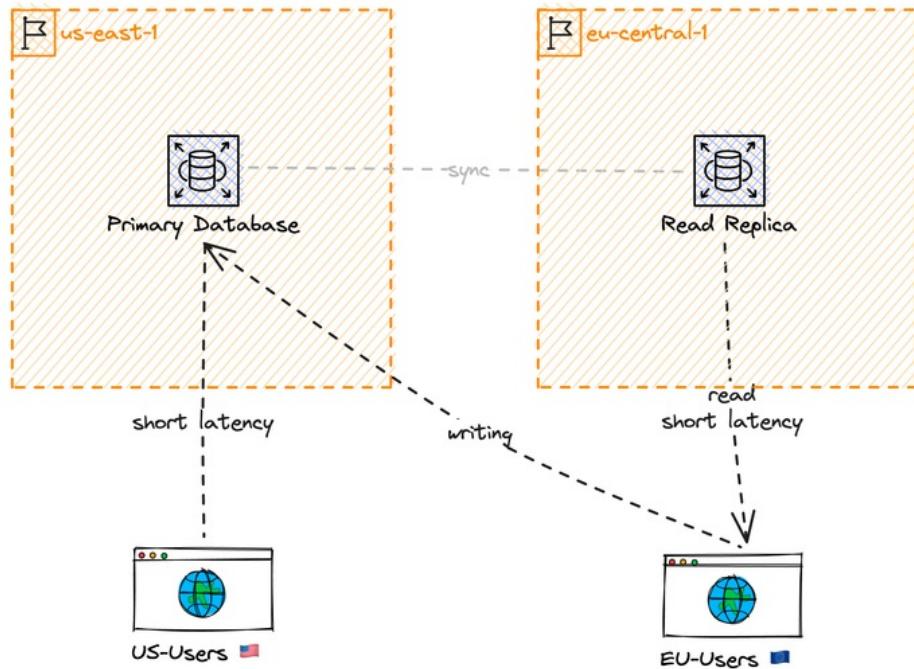
### **Read Replicas Allow You to Increase Performance for Read-Heavy Workloads Across Regions**

Another great feature of RDS is the use of Read Replicas. A read replica is a second database that only allows read access. Read replicas are often used for read-heavy applications to improve performance. Read Replicas can be in the same region or you can create one in another region.

For example, let's say you have an application where most users are in the US, and your application is located in `us-east-1`. All users from other countries, such as Europe, will experience longer latency, which is not ideal.



To improve performance, you can create a read replica in `eu-central-1`. This will enable much shorter latency to the database for reading workloads. Writing will still happen in `us-east-1`.



You can create a read replica in the console by selecting your database and clicking on [Create Read Replica](#).

RDS > Databases

Databases		<input checked="" type="radio"/> Group resources	<input type="button" value="C"/>	<input type="button" value="Modify"/>	<input type="button" value="Actions"/>	<input type="button" value="Restore from S3"/>	<input type="button" value="Create database"/>
			<input type="button" value="Delete"/>				
			<input type="button" value="Set up EC2 connection"/>		<input type="button" value="&lt; 1 &gt;"/>		
			<input type="button" value="Create read replica"/>				
			<input type="button" value="Create Aurora read replica"/>				
			<input type="button" value="Promote"/>				
<input type="checkbox"/> DB identifier				<input type="button" value="Region &amp; AZ"/>		<input type="button" value="Size"/>	
<input checked="" type="checkbox"/> database-1				<input type="button" value="us-east-1d"/>		<input type="button" value="db.t3.micro"/>	

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 to a certain point in time in case you introduced a bug, accidentally deleted entries, or experienced any other incident that resulted in corrupted or lost data.

RDS supports automated backups, which 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 by setting a backup window, which is the time when a backup takes place.

The backup window is optional. If skipped, RDS will fall back to default times for each region.

It's important to keep in mind that the backup process will **increase the load** on your database while it is being created. It's a good practice to configure backup windows for times when your application is not at peak load, such as at night 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 created it from the console.

You can easily restore your backup either via API or via the console. Simply choose a backup and promote it to your current data.

## Encrypt Your Data Directly on the Server

You can encrypt your data within RDS. Activating encryption will **encrypt your data at rest**, meaning that your data will be encrypted on the actual hardware using the industry standard 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 automatically activates this feature for every database creation, and it is recommended that you keep it activated.

The default encryption key is stored in KMS. You can also provide your own keys or a customer key, which will allow only your customer to decrypt the data. All logs, backups, and snapshots are encrypted, and a read replica of your encrypted database must also 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.

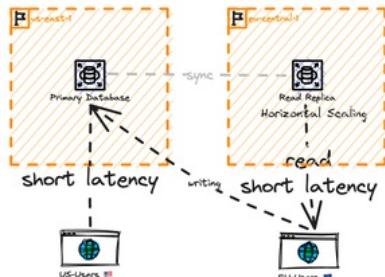
You can configure auto-scaling policies with RDS. With these policies, you are able to automatically scale up and down.

## Horizontal Scaling Adds More Instances While Vertical Scaling Enhances the Current Instance

## Horizontal vs. Vertical Scaling

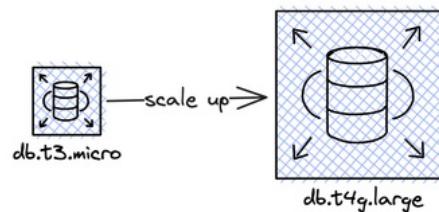
### Horizontal Scaling

- Increase performance
- Add read replica



### Vertical Scaling

- Enhance current instance
- Scale instance storage up



### Horizontal Scaling Adds More Instances

Horizontal scaling is a common way to scale compute resources, especially with the age of the cloud. It has become very popular to use consumer hardware and scale it up.

You can also do this with RDS. Instead of having just one database instance, you can add more instances. 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 a lot.

Another way to increase 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

The second way of scaling is vertical scaling. Vertical scaling refers to making the current database instance better. This results in more memory, more CPU, or more storage.

You can **auto-scale** your instances. In RDS, this is **only possible for storage**. Don't confuse this with other auto-scaling options in ECS or EC2. **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.

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.

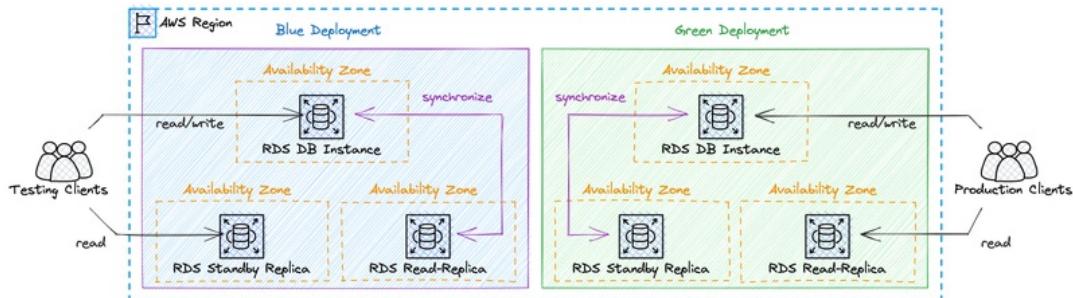
## Safely Deploy To Production With Blue/Green Deployments

Blue/Green deployment is a software release management strategy that reduces downtime and risk by running two identical production environments called Blue and Green.

Here's how it works with Amazon RDS:

1. **Blue Environment:** This is your live production environment, which is currently serving your users.
2. **Green Environment:** This is the clone of your production environment. Any changes or new features are deployed to this environment.
3. **Testing:** After deployment to the Green environment, thorough testing is performed. This includes integration testing, performance testing, and any other necessary testing.
4. **Switching:** If the testing is successful, the router or load balancer switches all user traffic from the Blue environment to the Green environment. This switch is done with zero downtime, as both environments are running simultaneously.
5. **Rollback:** If any issues are detected after the switch, you can immediately roll back to the Blue environment, again with zero downtime.

With Amazon RDS, you can create a snapshot of your live database (Blue) and restore it to a new database (Green). You can then point your application to the new database for testing. Once testing is completed, you can switch your application to use the new database. If any issues are found, you can switch back to the original database.



Remember, the key to Blue/Green deployment is having two identical environments and a reliable process to switch traffic between them.

This strategy allows you to test the new changes in a clone of your production environment without affecting the actual production environment.

### **RDS is Priced Based on the Instance Class and 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 have to pay for the database. You can check out the pricing calculator to get an accurate estimate of your database's cost.

The main pricing 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 required.
- Backup: How many backups have been taken.

### **Use Cases from the Real World**

There are numerous examples of use cases for SQL databases. Anywhere you need to store and maintain data is suitable for a SQL database. Let's take a look at three example use cases that are commonly seen in the industry.

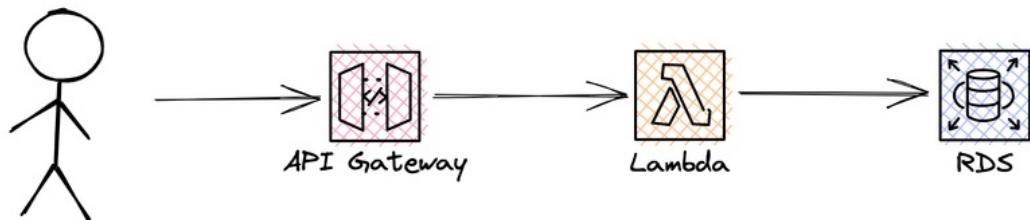
#### **Use Case 1: Building Web and Mobile Applications**

Building web and mobile applications is a common use case for RDS. Let's say you are developing customer management software. In your RDS database, you can save all your customer data, such as:

- Names
- Addresses
- Telephone Numbers

RDS makes it easy to set up a database and store this data in the cloud. A typical application could consist of the following services:

- **API Gateway** - exposes your business logic
- **Lambda** - executes business logic on API requests
- **RDS** - persists data



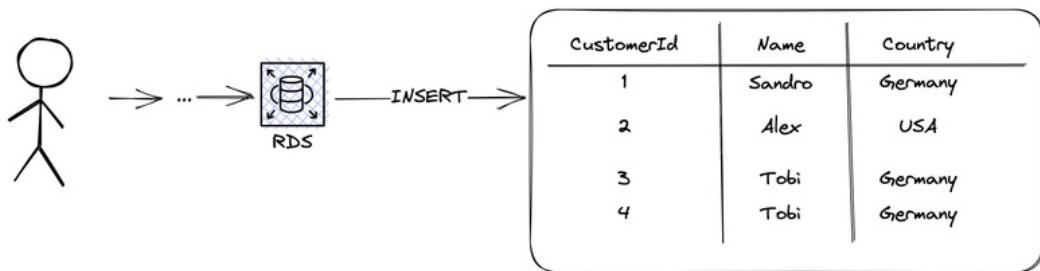
This architecture enables you to construct a redundant, resilient, and high-performance application without needing to concern yourself with the underlying infrastructure.

Furthermore, RDS enables you to scale up your database to handle peak loads and automatically generates backups to ensure the availability of your data. This makes it an excellent choice for building web and mobile applications that require data storage and persistence.

#### 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 its ability to perform Online Analytical Processing (OLAP) transactions, which involves grouping data together, such as with basic aggregates like sums or averages.

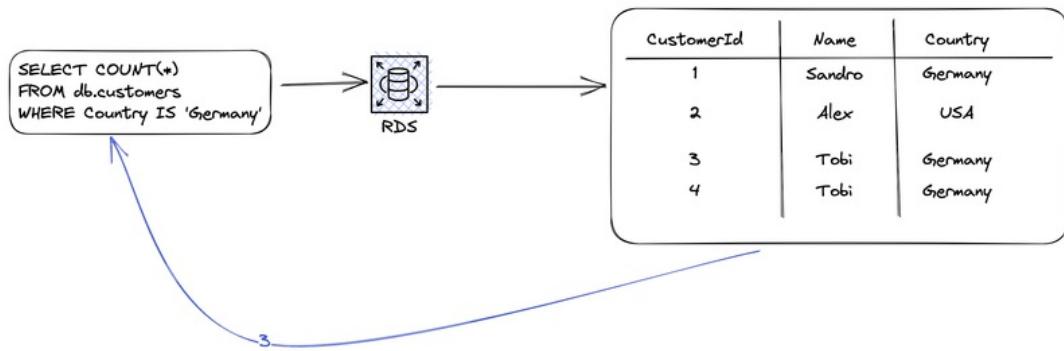
Let's take our customer management software as an example again. This is what your data should look like:



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-Premises Databases to Managed Databases

One common use case for RDS is to take advantage of its managed service. Many companies have their entire databases running on bare metal servers or virtual machines, which incurs a lot of overhead. Dedicated engineers are required to:

- Back up databases
- Update the underlying operating system
- Keep track of licensing issues
- Scale databases manually

Many of these tasks can be automated and transferred to AWS by using RDS. A common project is to move on-premises workloads (workloads that are not on the cloud) to the cloud.

Many on-premises databases also have expensive licenses. Another step that is often involved here is to eliminate the high licensing costs. Engines like Aurora or Postgres are often used for this purpose since they are free.

### Tips for the Real World

- **Use Multi-AZ** - your database or Availability Zone may 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 Replicas** - 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**

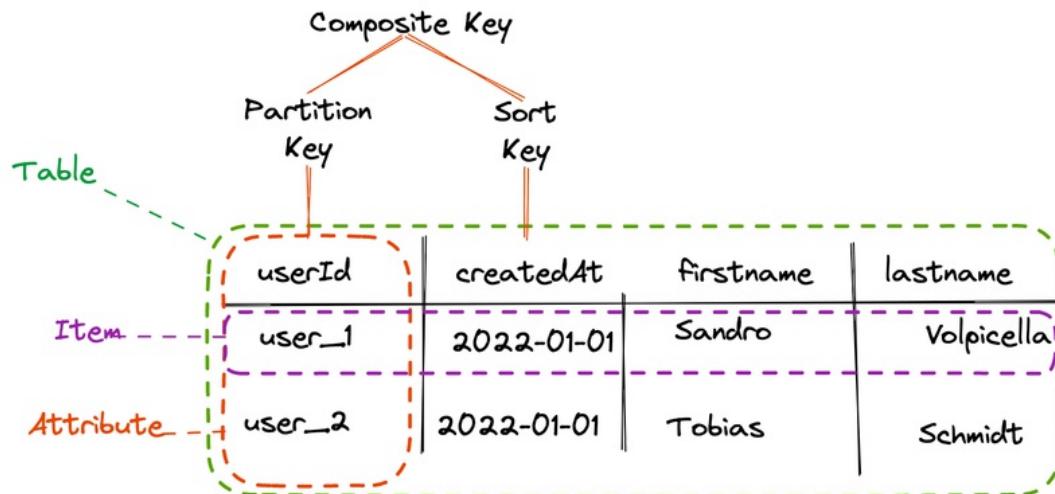
# Building Highly-Scalable Applications in a True Serverless Way With DynamoDB

## Introduction

DynamoDB is a fully-managed NoSQL database that can handle any scale. Additionally, it offers great features to integrate natively with other services. As it is not your typical NoSQL storage but comes with a long list of unique features, it is 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 them.



Term	Definition
<b>Table</b>	A table is a collection of items. A table in DynamoDB is akin to a table in a SQL database, storing numerous items. In our example, it is the entire user table.
<b>Item</b>	An item is one row of a table. For instance, one user could constitute one item. One item can possess several attributes such as last name, first name, address, etc.

Term	Definition
<b>Attributes</b>	An attribute is a field within an item, similar to a key in a JSON document. An attribute is defined as a type like a string, number, etc. Furthermore, one attribute can also represent an entire JSON document.
<b>Type</b>	DynamoDB supports various types of attributes including: - Number - String - Binary - Boolean - Null - List: ["Apple", "Banana"] - Map: Another document / JSON - Sets: List with one data type, e.g. only numbers. We'll delve into these in greater detail.
<b>Primary Key</b>	A primary key <b>uniquely identifies</b> an attribute within a table. There are two types 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 combination of these two keys <b>uniquely identifies your item</b> . The uniqueness isn't determined by the partition key alone but by the composition of both. A sort key automatically sorts your query, which can be very handy. For a user table, this could be, for instance, <code>userId</code> and <code>createdAt</code> . We will explore this in more detail later.

**DynamoDB is a NoSQL database. It saves your data in 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.

#### There Is No Explicit Schema in DynamoDB

One of the main differences between SQL and NoSQL databases is the schema.

SQL enforces a pre-defined schema, requiring certain attributes to be added and only allowing attributes present in the schema.

NoSQL works differently. You can write anything to your database, with the only required attribute being the primary key.

All other data can be written to the database, and it is common practice to have attributes available only for a certain set of your data. This may result in many empty fields, which is acceptable.

Developers with a strong SQL background may struggle with this, as 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.

Let's see another example from the RDS chapter again.

CustomerId	Name	Country
1	Sandro	Germany
2	Alex	USA
3	Tobi	Germany
4	Tobi	Germany

RDS

In the screenshot above, you can see a typical database table. The table has the following fields:

- CustomerId
- Name
- Country

Each customer in this database must have these fields. You cannot add any additional fields.

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 the second screenshot, you can see a typical example of a DynamoDB table. It has the following fields:

- CustomerId
- Name
- Country
- Address
- Email

The fields "Address" and "Email" have some `null` values. DynamoDB does not enforce a schema while writing to the database.

This lack of enforcement means that you can add other fields to the table that were not defined before.

However, there is still an implicit schema, as your application expects the data in a certain format. This format is not random and often cannot be changed easily. This is especially true for DynamoDB and its access pattern requirements, which we will explore later on.

#### DynamoDB is a key-value database and follows a JSON syntax

DynamoDB is a **key-value** database that stores data differently than SQL databases.

Items within DynamoDB are documents that follow the JSON syntax. Let's take a look at one of our users 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, such as a string (`S`) or number (`N`). 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 this case, a string. We will dive into this 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.

#### NoSQL Doesn't Mean Non-Relational. Your Data Always Has Relations

A word about non-relational: NoSQL often refers to a database that does not have relations. However, this term 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.

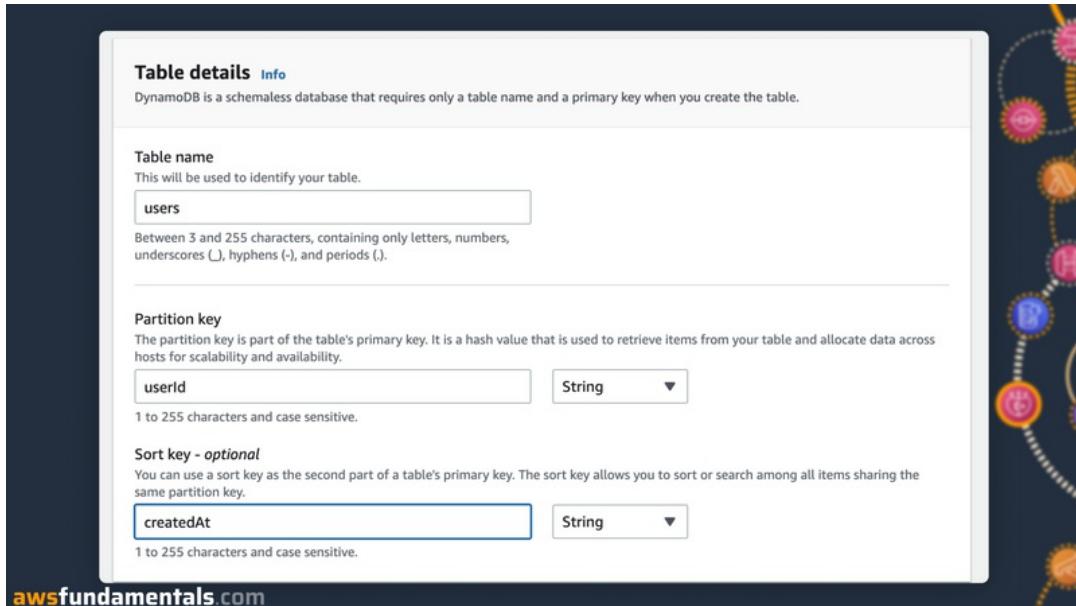
While the technology is nonrelational, your data still has relations. Unlike SQL, you cannot perform joins across tables, but your data still has relations.

For example, if you have multiple tables in your application and one of them is a user table, your users still have relations to other items in different tables, such as subscriptions, companies, etc.

Keep this in mind while setting up your database. We'll see different access patterns later, and 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 Items Uniquely. They Can Be a Simple or Composite Key

Primary Keys identify each item in your database uniquely. This concept is similar to a SQL database. DynamoDB offers different options for primary keys: **simple** and **composite**.

### Simple Primary Key - Relies Only on the Partition Key

A **simple** primary key is a single attribute that identifies the item in the table. For example, in a user table, this could be a `userId`.

Once you add a user with the `userId: user_1`, you cannot add another user with the same ID. The primary key refers to a partition key in this case.

A **Partition Key** defines how DynamoDB stores your data. DynamoDB uses partitions to store your data. If you want to learn more about this, check out this guide. However, this is not crucial for getting started 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**.

Let's use an example for the same user table:

- Partition Key: `userId`
- Sort Key: `createdAt`

As the name suspects, 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. This allows having users with the same `userId` but different `createdAt` times (which doesn't necessarily make sense).

	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 is 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 may not seem natural at first glance but it 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:

- Begins with
- Equal to (=)
- Less than or equal to (<= | <)
- Greater than or equal to (>= | >)
- Between

By having more data within your sort key, you can fulfill even more query requirements.

Let's add three more users:

<input type="checkbox"/>	userId	createdAt
<input type="checkbox"/>	user_2	companya#2020-08-21
<input type="checkbox"/>	user_2	companya#2020-08-20
<input type="checkbox"/>	user_2	companyb#2020-08-20

We've added some metadata to the `createdAt` field: `[COMPANY]#[DATE]`.

We can now do several things:

1. Query for `user_2`.
2. Query for `user_2` where `createdAt` begins with `company`.

The first query will show us all `user_2` records. The second query allows us to select only a subset of the data.

The screenshot shows the AWS DynamoDB Query interface. In the top section, under 'Scan/Query items', the 'Query' tab is selected for 'Scan/Query a table or index'. The table name 'Users' is chosen. The 'userId (Partition key)' is set to 'user\_2'. The 'createdAt (Sort key)' filter is set to 'Begins with companya' with the 'Sort descending' option unchecked. Below the filters, there are 'Run' and 'Reset' buttons. The results section shows a green 'Completed' status with 'Read capacity units consumed: 0.5'. It displays two items: 'user\_2' with 'createdAt' 'companya#2020-08-20' and another 'user\_2' with 'createdAt' 'companya#2020-08-21'. There are 'Actions' and 'Create item' buttons at the top of the results table.

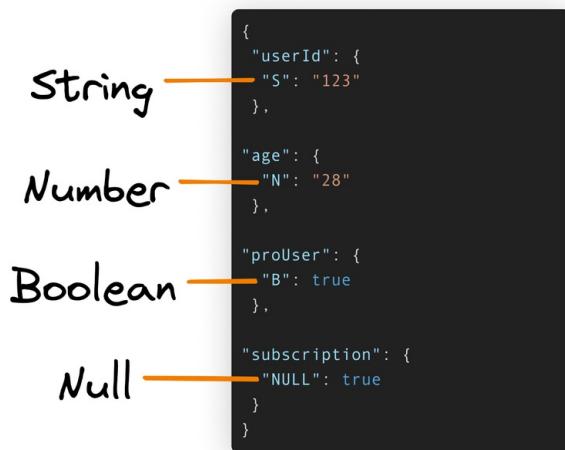
While this access pattern may not be useful for our example, thinking about access patterns when designing your keys is a huge benefit because you cannot change them afterward.

You can add indexes later, which will have their own keys. However, 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.**



**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



**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
List	L
Map	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

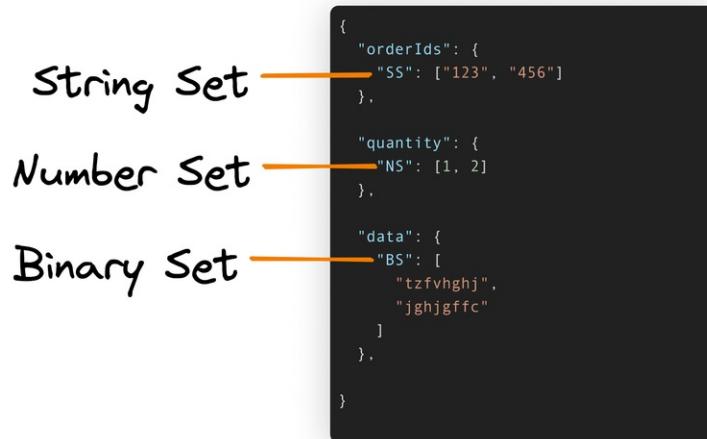
```
{
  "userId": {
    "S": "user_1"
  },
  "createdAt": {
    "S": "2022-08-20"
  },
  "firstname": {
    "S": "Sandro"
  },
  "lastname": {
    "S": "Volpicella"
  },
  "subscription": {
    "M": {
      "customer_id": {
        "S": "cus_123"
      },
      "plan_id": {
        "S": "plan_123"
      }
    }
  }
}
```

## 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 and writing data. For most use cases, this boils down to the following operations: **Query** and **Scan**.

userId	createdAt	firstname	lastname
user_1	2022-01-01	Sandro	Volpicella
user_2	2022-01-01	Tobias	Schmidt

**Scan**  
Scans the whole database

**Query**  
Queries a subset of the table

### Scan - A Slow Iteration Through the Whole Database

A scan goes through the **entire database** and retrieves **a list of items**. Depending on the number of items in your database, this list may not be complete. Your data is also structured into different pages. This is called pagination.

Users

Autopreview View table details

▼ Scan/Query items

Scan/Query a table or index

Scan Query Users

▶ Filters

Run Reset

Completed Read capacity units consumed: 2

Items returned (5)

C Actions ▾ Create item

< 1 > ⌂

userId	createdAt	age	orderIds	orders	subscription
user_2	companya#2020-08-20	28	[{"S": "1"}...]	{hijklkjkl, true}	
user_2	companya#2020-08-21				
user_2	companyb#2020-08-20				
user_1	2022-01-01				{"customer_id": {"S": "cus_1"}, "p...}
user_1	2022-01-02				

Since we only have five items in the table, we can see all of them. One scan can retrieve up to 1 MB of data.

Scans are expensive because you pay for all the iterated items. They are also slow because they iterate the entire table. **Always consider a query first**, or better yet, define your access patterns before creating your table. Only consider a scan as a last resort.

### 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. In the top section, under 'Scan/Query items', the 'Query' tab is selected. The 'Scan/Query a table or index' dropdown is set to 'Users'. A 'userId (Partition key)' input field contains 'user\_1'. An 'createdAt (Sort key)' input field has 'Equal to' selected and contains 'Enter sort key value'. Below these fields are 'Filters' and 'Run' and 'Reset' buttons. In the bottom section, a green 'Completed' status bar indicates 'Read capacity units consumed: 0.5'. The results table is titled 'Items returned (2)'. It lists two items: one with 'userId' 'user\_1', 'createdAt' '2022-01-01', and 'subscription' object; and another with 'userId' 'user\_1', 'createdAt' '2022-01-02', and an empty 'subscription' object. The table includes columns for checkbox, userId, createdAt, and subscription, with sorting arrows for userId and createdAt.

	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 perform operations in a batch. Instead of making a dedicated API call for every item, you can group many items into one batch call, such as submitting updates for 10 different items.

DynamoDB offers the following batch operations:

- `BatchGetItem`
- `BatchWriteItem`

- `BatchExecuteStatement`

We won't go into detail about these. If you need to retrieve or write a high volume of data, always check if a batch operation is available.

#### **Scans as a Last Resort**

As previously mentioned, scans should only be used as a last resort. Thoughtful access patterns, good primary key design, and queries are the preferred methods.

#### **Extending Your Query Capabilities with Indexes**

You are not limited 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 called `Orders` to explore this feature in detail.

*Partition  
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

This table has the following attributes:

- `orderId` - the partition key
- `productId` - ids of other products
- `quantity` - a number of how many were ordered

- `user` - the user who ordered

We can only **query** data based on the `orderId`.

### **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. This is where indexes come in.

#### **Understanding Global and Local Secondary Indexes**

Let's dive quickly into the different types of secondary indexes in DynamoDB: **global (GSI)** and **local (LSI)**. Each has different features and requirements for creation. Both can also help us solve the issue of querying additional data.

Global Secondary Index (GSI)				Local Secondary Index (LSI)			
				LSI Partition Key & Table Partition Key		LSI Sort Key	
orderId	productId	quantity	user	orderId	productId	quantity	user
order_1	product_2	2	user_1	order_1	product_2	2	user_1
order_2	product_2	40	user_1	order_2	product_2	40	user_1
order_3	product_1	1	user_2	order_3	product_1	1	user_2

- 1. 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 the data** from the table. This index lives in its own partition space.
- 2. 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 details about partitions, check out this amazing article by Alex Debrie.**

### Let's Create a GSI for Our Use-Case

Our challenge of querying **orders for a specific user can be solved by using a global secondary index**. But why use a global index instead of a local one? There are two main reasons:

1. We want to define a **new primary key** (`userId`). With **local** secondary indexes, you need to reuse the existing partition key.
2. You can only generate LSIs **during table creation**. Once the table is created, you cannot generate new LSIs.

This is why we will use a **Global Secondary Index**. Let's create it now.

Go to your table → Indexes and click on **Create Index**.

The screenshot shows the 'Indexes' section of the AWS DynamoDB console for the 'Orders' table. The 'Indexes' tab is active. At the top right, there is a 'Create index' button. Below it, a table lists 'Global secondary indexes (0)'. The table has columns: Name, Status, Partition key, Sort key, Read capacity, Write capacity, Projected attributes, Size, and Item count. A note below the table reads: 'No global secondary indexes. Global secondary indexes allow you to perform queries on attributes that are not part of the table's primary key.' At the bottom of the section, there is another 'Create index' button.

Doing this will open a small guide for creating an index:

**Index details** [Info](#)

Partition key	Data type
userId	String ▾
1 to 255 characters.	
Sort key - optional	Data type
Enter the sort key name	String ▾
1 to 255 characters.	
Index name	
<b>byUser</b>	
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 a common practice to name indexes like that:

`by<ENTITY_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 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`.

The screenshot shows the AWS Lambda console interface for the Orders table. At the top, there is a dropdown menu labeled "Scan/Query items" with "Query" selected. Below it, the "byUser" index is chosen from a dropdown under "Index". The "userId (Partition key)" dropdown contains "user\_1". The "Filters" dropdown also contains "byUser". There are "Run" and "Reset" buttons at the bottom of this section. Below this, a message indicates "Completed" with "Read capacity units consumed: 0.5". The results section shows a table titled "Items returned (3)" with three rows of data: "order\_2" with productId "product\_2" and quantity "2" for user "user\_1"; "order\_28" with productId "product\_2" and quantity "2" for user "user\_1"; and "order\_1" with productId "product\_2" and quantity "2" for user "user\_1". There are "Actions" and "Create item" buttons at the top of the results table, along with navigation controls for pages 1 and 2.

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

This index represents one of our access patterns: **Getting orders by a user**. Your application consists of multiple access patterns like this.

By thinking about your access patterns upfront, you can choose a primary key that supports the access patterns of your application and helps 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 be using the CLI for the first time. We felt the need to explain mutating items in more detail since some concepts can be 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 applies to `#quantity`.

We map the values in the same way only in the block `--expression-attribute-values`.

Remember that **expression attributes** are like variables.

- Expression Names start with `#`
- Expression Values start with `:`

### Inserting Items

First, let's insert items into DynamoDB. There are many ways to insert data into DynamoDB, including using the web console, the AWS API, or the CLI. In this case, we will use the CLI.

To insert 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 `item.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**. With **On-Demand**, there's no need to know how many reads and writes you'll need per second, as it will scale immediately. **Provisioned capacity** requires you to know your traffic patterns and maintain a steady level of available read and write capacity, which can also be scaled via CloudWatch, but at a much slower rate.

In general, these capacity modes define two things:

1. Your bill: On-Demand capacity is **much more** expensive.
2. The possibility that your request may 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 (WCU)**.

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 - for example, 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>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 purpose.

### On-Demand Capacity for Unknown and Inconsistent Traffic Patterns

The On-Demand capacity mode does not 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 do not want to monitor and manage write and read access.
- It is 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 for Predictable Traffic Patterns

Provisioned capacity is up to seven times cheaper than on-demand, but requires you to define RCUs and WCUs. These will be billed regardless of whether 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 Which Mode?

- **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, choose on-demand.

## Global Tables: Creating Synchronized Tables Across Regions

With DynamoDB's global table feature, you can easily synchronize tables across regions, increasing resiliency and following the patterns of the AWS Well-Architected Framework.



Data is not only backed up to another region but also has a **bi-directional** synchronization. Regardless of the write region, each region within the global table definition will receive all updates.

Although 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>	This is also called <b>Point in Time recovery</b> . 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

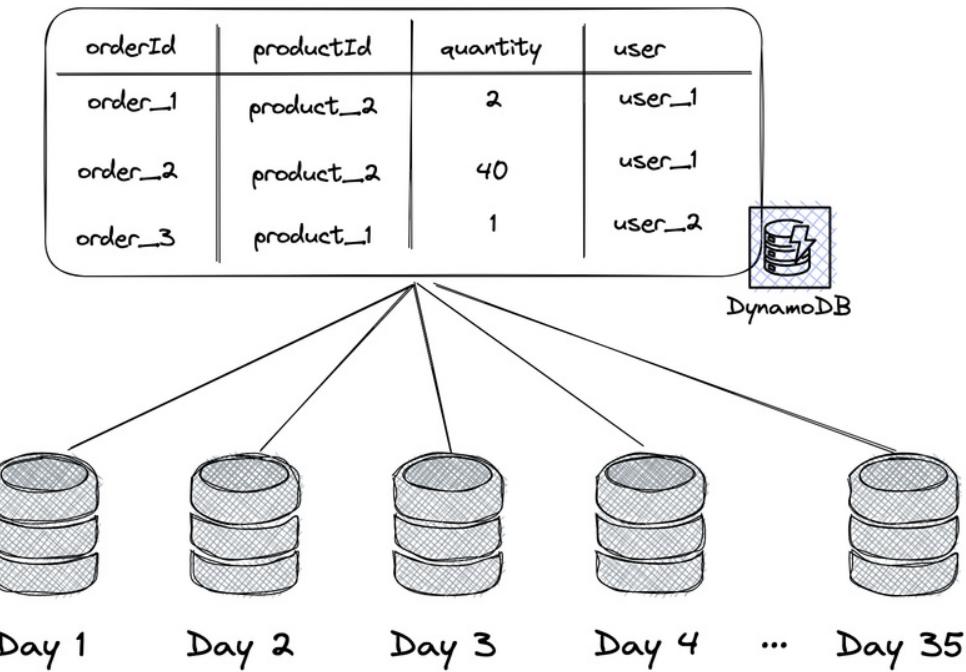
AWS Backup is AWS's **global** backup service, which does not only cover DynamoDB but also other services such as S3. Its feature set includes amongst others:

- 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 supports both **continuous** and **on-demand** backups. Tables with continuous backup enabled use point-in-time recovery to reset the table to any point in the past within 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

Restoration is a quick and easy task. Go to your backup console, click on **Restore**, and specify the time to which you want to restore, as well as the 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.

OrdersRestore

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)

2022/08/18   23:01:11

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

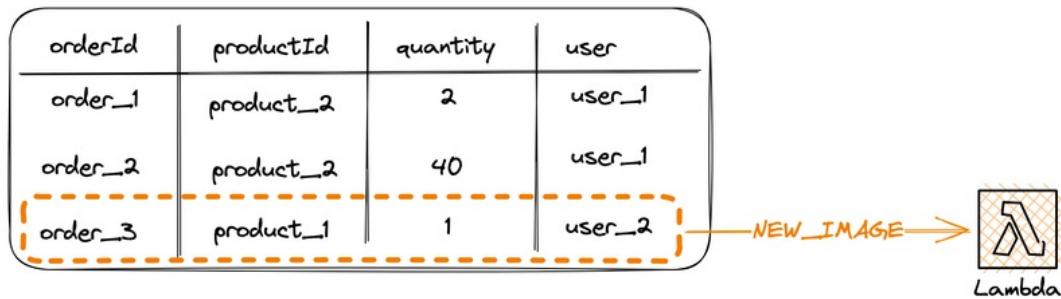
On-demand backups can be done manually in the console tab (**Backup > Create Backup**) or you can trigger them, for example, with an EventBridge rule and a Lambda function. Each backup will also be saved in DynamoDB.

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 a status of 'Enabled'. Below it, a table provides restore points: 'Earliest restore point' on August 20, 2022, at 17:11:32 (UTC+02:00), and 'Latest restore point' on August 21, 2022, at 12:35:33 (UTC+02:00). There are 'Edit' and 'Restore' buttons.

Below this is a 'Backups (0)' section. It includes a search bar ('Find backups by ARN or name'), buttons for 'View details', 'Restore', 'Copy', 'Delete', and 'Create backup' (which is highlighted with a dropdown menu showing 'Create on-demand backup' and 'Schedule backups'). A table below shows columns for Name, Status, Creation Time, ARN, and Size, with a note 'No backups'.

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. For example, we can 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 **Exports and Streams** tab.

The screenshot shows the AWS DynamoDB console interface. The top navigation bar includes tabs for Overview, Indexes, Monitor, Global tables, Backups, and Exports and streams, with the latter being the active tab. Below the navigation is a section titled "Exports to S3 (0) Info" which says "Showing all export jobs from the last 90 days." It features a search bar labeled "Find exports" and a table header with columns: Export ARN, Destination S3 bucket, Status, and Start time (UTC+02:00). A message "No exports" is displayed. There is also an "Export to S3" button. The next section, "Amazon Kinesis data stream details", contains a description of Amazon Kinesis Data Streams for DynamoDB and an "Enable" button. The status is listed as "Disabled". The final section, "DynamoDB stream details", describes capturing item-level changes through the DynamoDB Streams API and also has an "Enable" button. Its status is also "Disabled".

Once you enable streams, you have the option to choose 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.

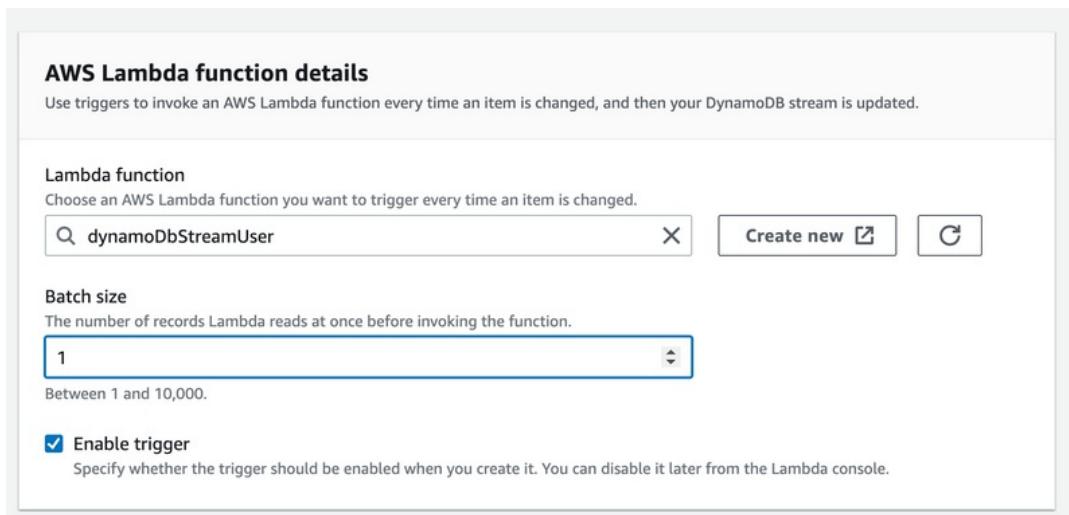
- Key attributes only**  
Only the key attributes of the changed item.
- New image**  
The entire item as it appears after it was changed.
- Old image**  
The entire item as it appears before it was changed.
- New and old images**  
Both the new and old images of the changed item.

[Cancel](#) [Enable stream](#)

There are different options that define what will be passed to your function.

Type	Description	Example when inserting a new user
<b>Key attributes only</b>	Only the key of the changed item	userId
<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 activating 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. This can be useful if you want to expire items in your database. It is often used to remove idempotency items, for example.

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.

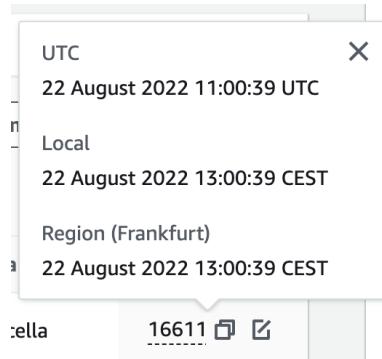
August 22, 2022, 12:58:47 (UTC+02:00)

ⓘ 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.



If you want to check which data will be removed and at what time, you can go back to your TTL settings and select **Run Preview**.

**Preview Time to Live**

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)**

userId	createdAt	firstname	lastname	ttl (TTL)
123	2022-08-20	Sandro	Volpicella	1661166039

TTL is a real life-saver for getting rid of unused and temporary data.

### DynamoDB Encrypts Data at Rest on the Server

DynamoDB encrypts all of your data at rest by default, meaning that 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.

## Getting the Most Out of DynamoDB by Aiming for a Single Table Design

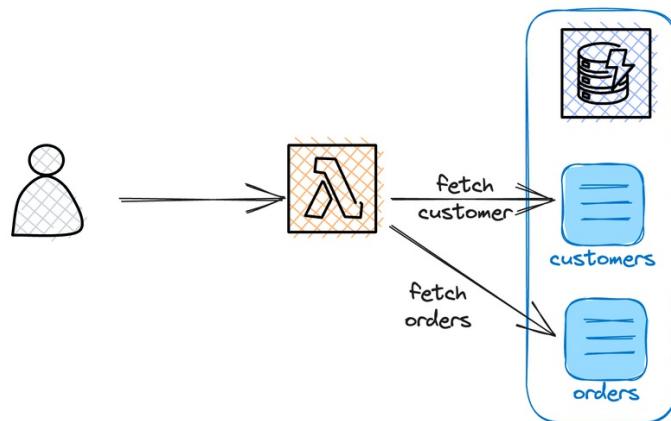
We know what DynamoDB does greatly: offering high availability and stable performance almost completely regardless of the demand. But as we've learned it also comes with many restrictions.

In the realm of SQL modeling and database design, the concept of joins plays a crucial role in effectively retrieving and combining information from multiple tables. When working with relational databases, it is common practice to normalize data by creating separate tables for each type of entity in an application.

Consider the very common scenario of an e-commerce application. To represent the entities involved, we would typically have a `customers` table and an `orders` table. Each order placed by a customer is associated with a specific customer. This relationship is established using foreign keys, which serve as pointers from one table to another. By utilizing these foreign keys, we can easily access additional information about a customer who placed a particular order.

The power of joins comes into play when we need to retrieve information that spans across multiple tables. By employing the SQL language's join functionality, we can combine records from the customers and orders tables (or even more tables if needed) during the read-time process. This allows us to access a comprehensive view of the data by leveraging the relationships defined through foreign keys.

In essence, joins enable us to bridge the gap between related tables, facilitating the retrieval of interconnected information. They provide a means to seamlessly merge data from different tables, ultimately enhancing the flexibility and efficiency of querying relational databases. Even though joins are very powerful, they come with a major downside: they are very expensive.



In DynamoDB, we don't have joins. Our workaround would be to still work with two tables and

do multiple operations: one to retrieve the customer, and one to retrieve their orders. This means, that even though we don't have enforced relations via foreign keys, we still have some implicit relations.

Sequential network requests can become a significant issue for your application. Network I/O is often the slowest part, and as your application scales, this pattern becomes slower.

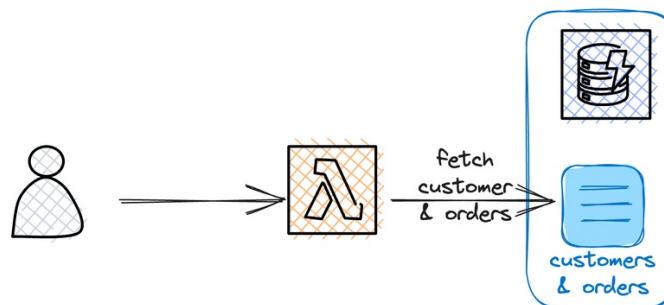
### Pre-Joining Data into Multiple Items with the Same Partition Key

To achieve fast, consistent performance in DynamoDB, pre-join data using item collections. An item collection refers to all items in a table or index sharing a partition key.

For instance, when we look at our previous example with orders, we can build a composite primary key so we can combine the customers and the orders into a single table.

Primary Key							
Partition Key	Range Key	Username	Name	Email	Status	OrderId	
sandro	CUSTOMER#sandro	sandro	Alessandro Volpicella	sandro@awsfundamentals.com	PENDING	bb536283	
	ORDER#bb536283	sandro			SHIPPED	ef70945f	
	ORDER#ef70945f	sandro					
tobi	CUSTOMER#tobi	tobi	Tobias Schmidt	tobi@awsfundamentals.com	SHIPPED	79452665	
	ORDER#79452665	tobi			SHIPPED	b7ad7278	
	ORDER#b7ad7278	tobi			SHIPPED	e24deab4	
	ORDER#e24deab4	tobi					

With a single query operation, we can now read the customer entry itself and all of their orders if we want to.

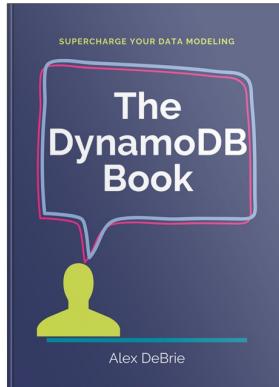


We can also easily read only the customer entry or the orders when we make use of the range key and the `starts_with` operator.

### Single Table Design Doesn't Come without Drawbacks

Although the single-table pattern in DynamoDB is highly scalable, it does have some drawbacks. These include a steep learning curve, limited flexibility for adding new access patterns, and challenges with exporting tables for analytics.

Keep this in mind when thinking about the possibility of making use of this pattern before starting your implementation. As we've learned, this approach is always important when you want to use DynamoDB. Read more about the fundamentals of the Single Table Design in Alex DeBrie's blog.



If you are looking to work with DynamoDB in the future, we highly recommend buying and reading "The DynamoDB Book". This comprehensive resource provides valuable insights and guidance on effectively utilizing DynamoDB. It covers various topics, including best practices, advanced techniques, and practical examples, making it an excellent reference for mastering DynamoDB.

Whether you are a beginner or an experienced developer, it will greatly enhance your understanding and proficiency in working with DynamoDB.

## Use Cases from the Real World

DynamoDB can be used for any application that needs to store data. Let's look at three common examples.

### Use Case 1: Building a High-Performance and Scalable Application

A very common use case for DynamoDB is building high-performance, scalable applications. NoSQL databases are really good at handling high loads. With DynamoDB, you don't need to worry about any scaling issues if your indexes are set up correctly.

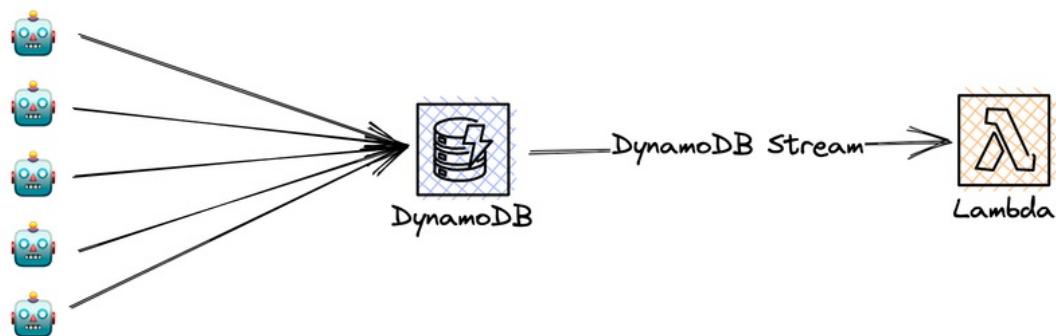
Let's take a social media platform as an example. DynamoDB allows you to store user data and activity logs, enabling millions of users to interact with the database without any downtime.

Peak traffic is handled automatically, and you don't have to worry about 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 large amounts 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 quickly.



This makes it a great choice for a database for Internet of Things (IoT) applications. IoT applications include a large number of sensors that constantly send data to a database. The number of sensors can reach millions. DynamoDB's streaming capability makes it an excellent 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 write 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** - When designing your database, don't start by thinking about your keys and attributes first. Instead, start by thinking about your

access patterns. For example, if you're building a project management tool, your access patterns could include:

- Getting users by user ID
  - Getting users by project
  - Getting 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** - This will allow you to restore your database to points in the last 30 days.
  - **Monitor usage 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-performing 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

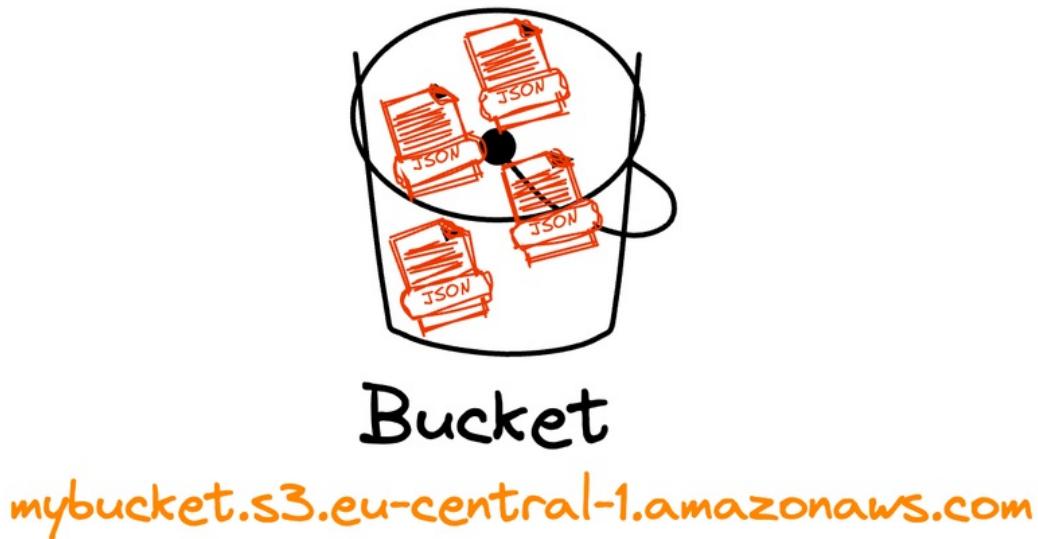
## Introduction

Amazon Simple Storage Service is one of the most widely used and well-known services on AWS. With S3, you can store and retrieve any amount of data at any time. S3 is not a database, but a **BLOB** (Binary Large Object Files) storage for saving large amounts of unstructured data. S3 is also one of the safest places in the world to store 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 store files. S3 offers unlimited storage and can store an unlimited number of files.

A bucket name needs to be **globally unique** because 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

Let's dive into some more details.

### **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. However, it is always a trade-off with other factors, such as:

- How **long** S3 needs to return objects
- How **durable** your data is
- How long it takes to **retrieve** your data

Let's take 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 that you don't have to memorize them all unless you're taking a certification exam. The main thing you need to understand is that there is no such thing as a free lunch.

The "no free lunch theorem" describes that there will always be a trade-off. You won't save costs without making 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 think about optimizing early and 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 yourself. AWS analyzes your access patterns and puts the objects into different storage classes. But all of that depends on your detailed access patterns and your stored data.

#### **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:

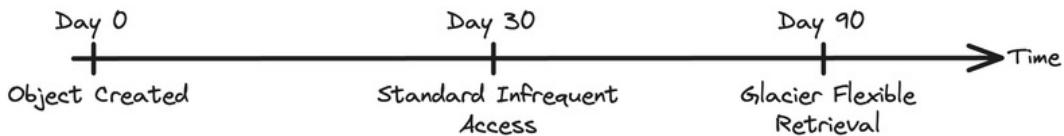
1. **Instant** Retrieval
2. **Flexible** Retrieval
3. **Deep** Archive

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 crashes, 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 the trade-offs out there.

### Lifecycle Policies Send Objects Automatically to a Different Storage Class

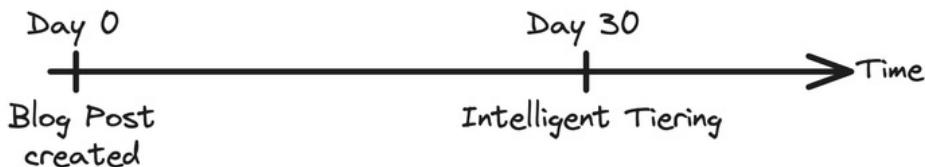
Lifecycle policies let you send data to a **different storage class after a defined time**. You can also remove objects after a certain time with policies.



Let's take the example of audio generation for our blog posts. The main access to a blog post often happens in the first few days or weeks after publishing, so 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 be moved to a cheaper storage class. This is where a lifecycle policy can be useful. It would be even better to use **Intelligent Tiering** in this case. AWS can figure out the access pattern and move it across storage classes. If traffic spikes again, for example due to search engine indexing, Intelligent Tiering can also move 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 navigating 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 with a certain prefix 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](#)

<input checked="" type="checkbox"/> Move current versions of objects between storage classes
<input type="checkbox"/> Move noncurrent versions of objects between storage classes
<input type="checkbox"/> Expire current versions of objects
<input type="checkbox"/> Permanently delete noncurrent versions of objects
<input type="checkbox"/> 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
Intelligent-Tiering	30
<a href="#">Add transition</a>	<a href="#">Remove</a>

**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 - Trigger Events on S3 Changes

S3 allows you to trigger events to other AWS services when changes occur in S3. These changes can include:

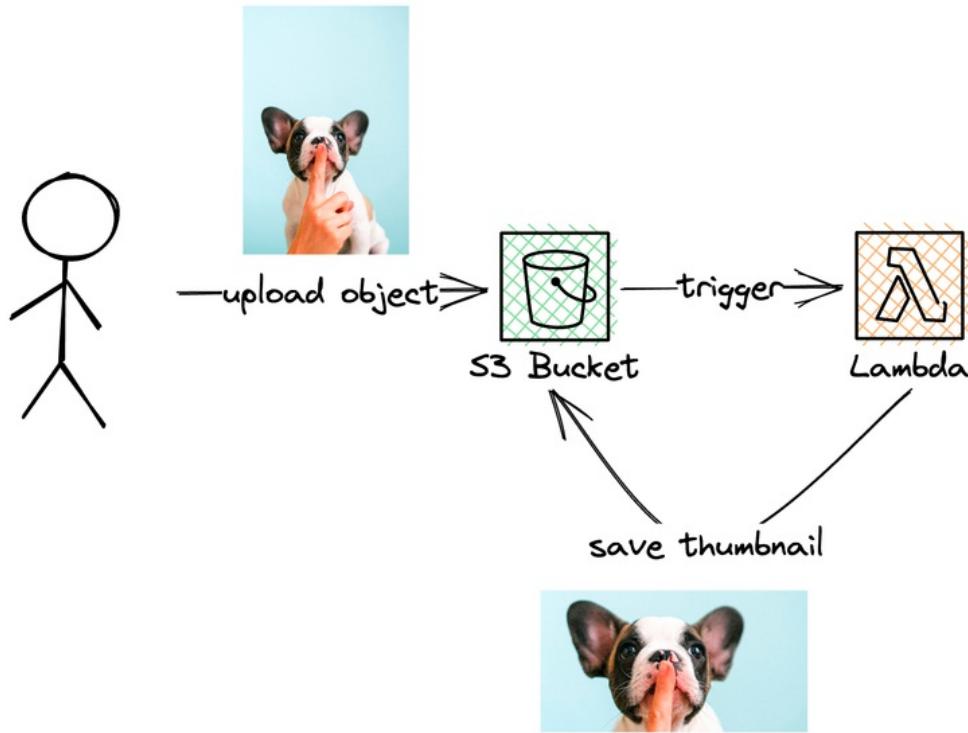
- Object creation
- Object removal
- Lifecycle expirations

- ... and many more

Building event-driven systems is a significant part of AWS. There are entire applications that are built on top of S3's notification 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 touchpoints.

And that's where S3 event notifications can step in. After an object is 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.
4. **EventBridge** → Use an event bus handling your events

For our example, we will choose Lambda. We will create a Node.js function that only logs the event for demonstrating purposes. You can follow along if you want.

This is the handler code:

```
exports.handler = async (event) => {
  console.info('EVENT\n' + JSON.stringify(event, null, 2));
};
```

After creating a Lambda function with this code we choose this Lambda function in the S3 console as the consumer of the S3 event notifications.

## Destination

**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**

**Cancel** **Save changes**

After saving the new event notification, 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!

### **Server Access Logging - Log All Access to Your Bucket**

Understanding what is happening in your S3 bucket can be quite challenging, especially with larger 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 logging requests for access to the bucket, with a 'Learn more' link. Below that, the 'Server access logging' section has two options: 'Disable' (radio button) and 'Enable' (radio button, which is selected). 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.' Under the 'Target bucket' section, the URL 's3://audioblogloggingbucket' is entered into a text input field, with a 'Browse S3' button next to it. Below this, a note says 'Format: s3://bucket/prefix'. At the bottom right are 'Cancel' and 'Save changes' buttons.

S3 saves the logs in another S3 bucket that you need to provide. The logging itself is **free**, but the **storage is charged**. Since S3 storage is relatively cheap, it's recommended to always activate the logging feature unless you expect an enormous amount of operations.

### CloudTrail Data Events Give You a Detailed View of API Calls in Your Bucket

Another way to log your S3 access is through CloudTrail data events. This service provides a detailed view of your S3 data usage.

AWS CloudTrail is Amazon's auditing service that captures internal API calls in your AWS Account. S3 offers the capability of capturing data events, which is different from normal CloudTrail API access.

API events in CloudTrail include the creation (`CreateBucket`) or deletion (`DeleteBucket`) of a bucket. Data events are resource operations such as retrieving (`GetObject`) or deleting (`DeleteObject`) an object.

For applications that use S3 frequently, many logs can occur. Make sure to calculate this with your pricing costs, as 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 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/*"
        }
    ]
}
```

These resource-based policies are set on the resource itself. In our example, we are granting permission to list objects as well as retrieve objects from the bucket. We are allowing this for a specific IAM user via the `Principal` field.

As we learned previously in the IAM chapter, there are many other things you can do, such as adding conditions.

### **Batch Operations Let You Save Money by Uploading Objects in Batches**

Like other services, S3 also offers batch operations that allow you to cluster operations into a single API request. This results in lower costs, as each S3 operation (above the Free Tier limit) incurs a charge.

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 itself makes use of batching many times in its internal services, such as EventBridge. If you use Schema Discovery, AWS batches many requests. You can tell this 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 in compliance with regulations. The acronym for object locks is **WORM**, which stands for **Write-Once-Read-Many**. Object locks are used to ensure that your uploaded data has not been tampered with by any systems or developers. This is especially useful 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 overwrite your data, but you want to generate a new version of your S3 objects. S3 allows you to version your objects, so you can keep older versions and roll back to a previous state.

Let's revisit 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 accidentally upload the wrong data.

Versioning in S3 is **fully managed**. You don't need to do anything other than activate the feature.

### Activating Versioning in the Management Console

Versioning can be activated in the Management tab of your S3 bucket.

The screenshot shows the 'Bucket Versioning' section of the AWS S3 console. It includes a descriptive text about versioning, an 'Edit' button, a status section for 'Bucket Versioning' (set to 'Disabled'), and a detailed section for 'Multi-factor authentication (MFA) delete' (also set to 'Disabled').

**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 view 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	Actions
<input type="checkbox"/>	FileA.png	png	3l_me3UrzNzTmALnlje3ArAtgoReOeh6	August 25, 2022, 16:31:52 (UTC+02:00)	76.8 KB	Standard	
<input type="checkbox"/>	FileA.png	png	7QfMCVrNohH7a0om0DK6kXAdhoYfn_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_aJXnfBAjDYzo0A	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 items to a previous version, download the object you want to restore and upload it again. This process is not very intuitive, but it is the official explanation by AWS. Alternatively, you can delete the newer version of the file to make the previous version available again.

### Making Your Bucket Public

Your S3 bucket is **not public by default**. Therefore, the option for blocking public access is **activated**. You can modify the option to 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 is **hosting static pages**. Your website should be public so that everyone can access it.

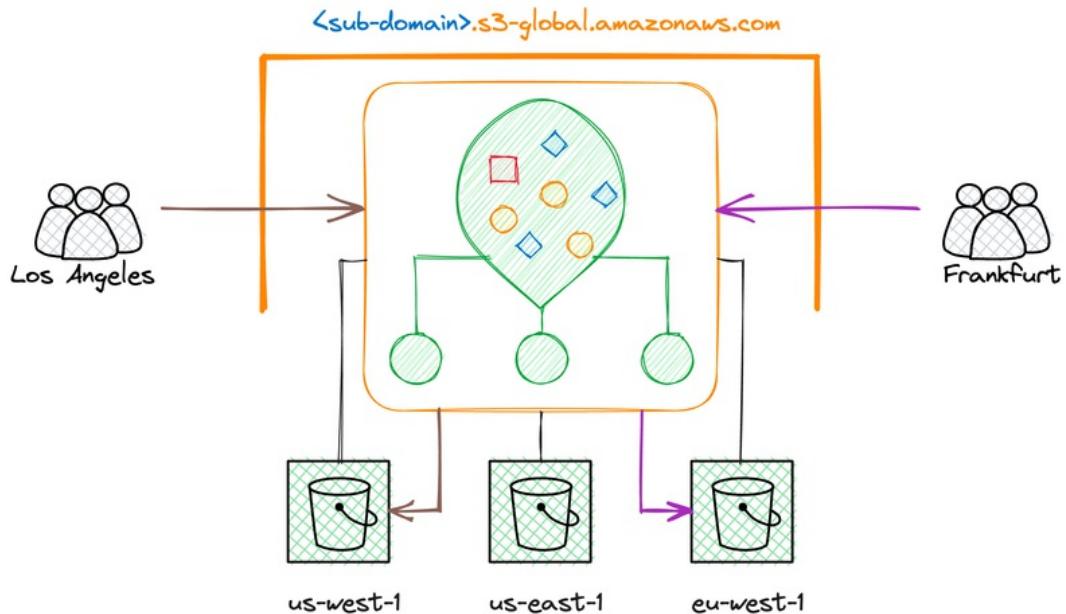
Another example is the audio blogs we've used throughout this chapter. Users should be able to listen to the podcast versions of blog posts. If users are not authenticating, they need to have public access. Thus, the bucket needs to be public. You can adjust this by deactivating **Block all public access (double negation yes → deactivate the block)**.

You need to be careful with this. You don't want to open up the wrong bucket (like all your CloudTrail logs) to the public. People **will** find it.

## **Multi-Region Access Points in Amazon S3 for Reduced Latency and Even Better Resiliency**

Amazon S3 Multi-Region Access Points provide a global endpoint for accessing S3 buckets in multiple AWS Regions. You can synchronize data amongst your multiple regions using S3 Cross-Region Replication.

- This enables applications to showcase their capability to function seamlessly across multiple AWS Regions.
- It enhances operational resiliency by operating across multiple AWS regions.
- By leveraging AWS Global Accelerator Edge locations, applications can achieve improved performance to S3 across multiple AWS Regions.



This means requests are automatically served from the closest available Region. Multi-Region Access Points are compatible with applications in Amazon VPCs and support AWS PrivateLink for Amazon S3. See the image below for a visual representation.

How to configure it and how it works internally:

1. Begin by establishing an S3 Multi-Region Access Point, which serves as a gateway to one or more S3 buckets (only a single bucket per region).
2. S3 Multi-Region Access Points are equipped with a global hostname, which is located within a distinct DNS subdomain: s3-global.amazonaws.com.

3. When making requests to the global hostname of an S3 Multi-Region Access Point, the requests are efficiently routed and accelerated across the AWS Global Accelerator network. This means requests are then served by the bucket that offers the lowest latency.

### Pricing in S3 is Based on Storage and API Calls

Pricing in S3 is mainly based on the amount of storage you need and the number of API calls you make.

Let's take a look at everything you'll be charged for:

1. **Storage:** Prices vary depending on the storage class you're using. For example, for a standard storage class, 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 much.
2. **Requests against your buckets:** Requests are charged extra and also depend 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 cost of data transfer. For data transfer, it always depends on whether the traffic is coming from the same AZ, region, CloudFront, or the internet. For example, traffic coming into S3 from the internet is free, while traffic going 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 cloud architecture.
4. **Advanced Features:** There are a couple of advanced features that can cost money. One example of this is Intelligent Tiering, which is the storage class that analyzes your data access patterns and automatically moves your data to the correct storage class based on the pattern. The scanning can cost around \$0.002 per GB. If you have an application with a lot of data and you're thinking about switching storage classes, you need to take this into account. You can calculate whether it's cheaper to get the data scanned or to leave it in the same storage class. There are many more nuances to consider when optimizing costs with S3.

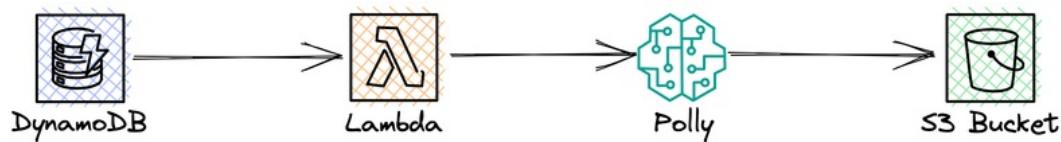
## The Wide Range of Use Cases for S3

Amazon's flagship storage service has a wide range of applications. Let's take a look at three use cases where S3 plays a major role: automatically generating audio descriptions from text, hosting static websites, and saving audio files.

### 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.

The following diagram shows an example of the architecture:



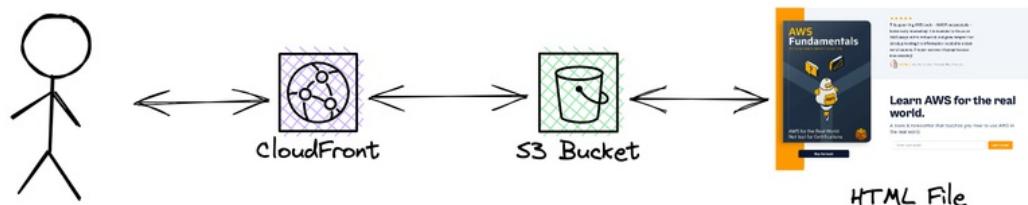
After you insert your data into DynamoDB, Lambda retrieves the event from a DynamoDB stream. Polly then generates the text into audio, which is stored in an S3 bucket.

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 hosting static websites. You can combine S3 and CloudFront to serve your content over AWS's edge network, which consists of many small servers around the world. This brings the content closer to your users.

S3 acts as the storage, while CloudFront serves the file worldwide.



If you want to host a static website on S3, you can create a bucket, upload your HTML files, and enable static web hosting.

You can host websites on S3 in five steps:

1. Create a 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 is designed to provide 99.99999999% durability.



With its various storage classes, you can save backups inexpensively (e.g. with S3 Glacier). You can automate the process of backing up data to S3 by using AWS SDKs and the AWS API.

#### Tips for the Real World

- Enable versioning for your S3 buckets. With versioning, you are protected 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 they are really needed. They can be very expensive.

## **Final Words**

S3 is one of the most widely used AWS services in the world. Even non-cloud developers know what S3 is for. However, most 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 and architectures for a very low price.

Digging deep into S3 is fun but is often not required. AWS gives us the opportunity to use its full storage class ability without the need to know all the details by activating Intelligent Tiering.



# Messaging

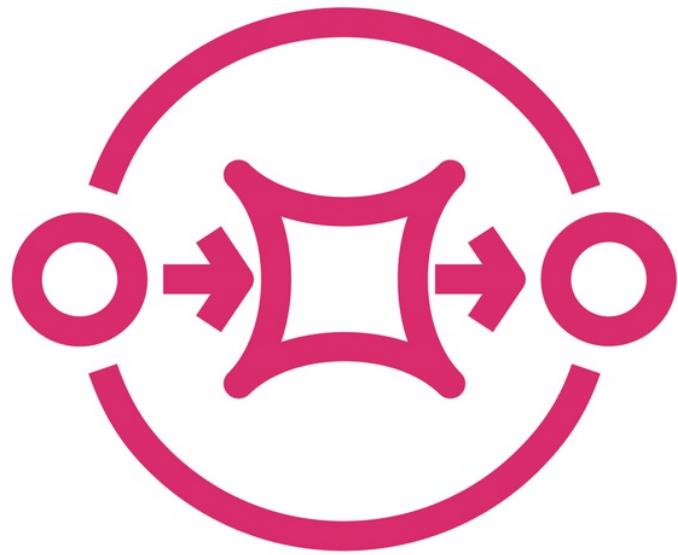
Messaging services are an important cluster of services for building web applications. They enable sending and receiving messages between different components of the application, allowing them to communicate and work together.

They can be particularly useful in situations where components need to be decoupled. This includes handling tasks that may take a long time to complete, or sending notifications to different parts of the application. They also help improve system resilience by enabling reprocessing of events in response to incidents.

In this document, we will introduce three fundamental messaging services: Amazon **Simple Queue Service (SQS)**, Amazon **Simple Notification Service (SNS)**, and **EventBridge**.

**SQS** is a fully managed message queuing service that simplifies decoupling and scaling of microservices, distributed systems, and serverless applications. **SNS** is a fully managed messaging service that allows sending messages to one or more destinations, such as email, SMS, SQS queues, and HTTP/S endpoints. Amazon **EventBridge** is a fully managed event bus service that enables connecting applications without tightly coupling them.

All three services are highly available, scalable, and fully managed, making it easy to build highly available web applications. They also support advanced features such as encryption, dead-letter queues, and filtering to improve messaging reliability and scalability in web applications. They enable routing and triggering events to the right resources, making it simple to build an event-driven architecture.



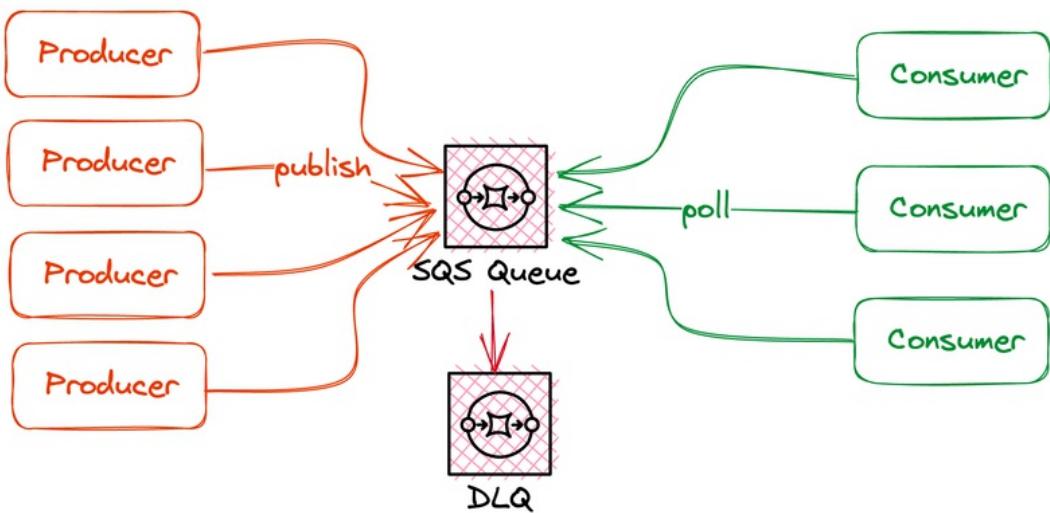
Amazon **SQS**

# Using Message Queues with SQS

## Introduction

Amazon Simple Queue Service (SQS) is a fully-managed message queue service provided by AWS. It enables you to build scalable and high-performance 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 are almost no cloud applications without SQS out there.

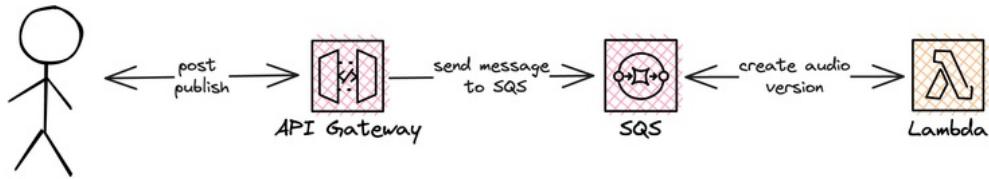


One amazing fact about SQS: it was the first service launched by AWS, even before S3 and EC2. To quote the legend, Jeff Barr: *"We launched the Simple Queue Service in late 2004, Amazon S3 in early 2006, and Amazon EC2 later that summer."*

And one stunning statistic: during 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. This misconception often results from the tight integration between Lambda and SQS.



Let's take an example of our blogging platform. Every time a user creates a blog post, we want to automatically convert it to an audio version.

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 we will discuss this further in the cost chapter.

In AWS, this is called the **Event Source mapping** between SQS and Lambda. It runs in the background and looks for new SQS messages. If a new message arrives, the mapping sends it to Lambda.

So the main point to take away is that **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 two types of queues in SQS: Standard and FIFO queues.

The main difference is in the ordering of messages. Standard queues do not order messages, while FIFO queues do. This has several implications.

First, we need to understand the delivery methods in SQS and AWS in general. This is necessary to understand the differences between 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 that they will deliver a message at least once. It also means that consumers of an AWS SQS Standard queue can receive a 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 if 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.

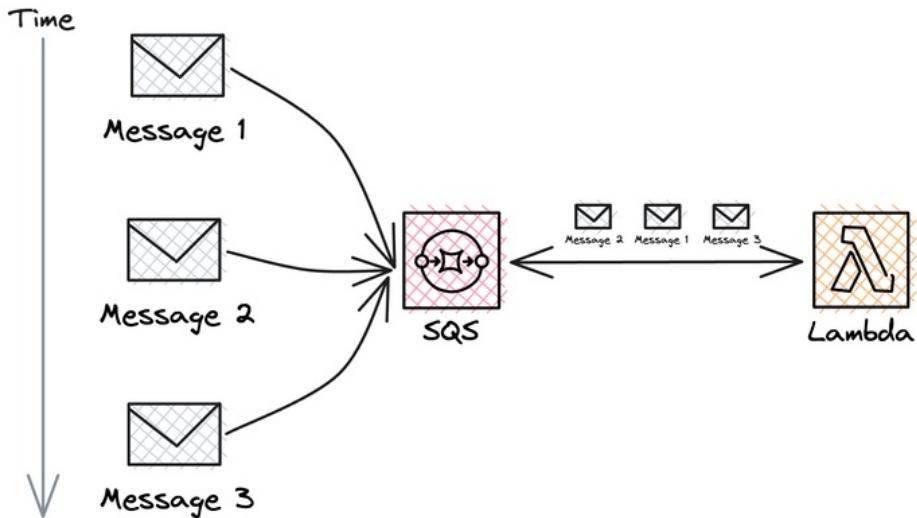
To overcome this issue and make your calls idempotent, you can do two things:

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**

## Standard Queue



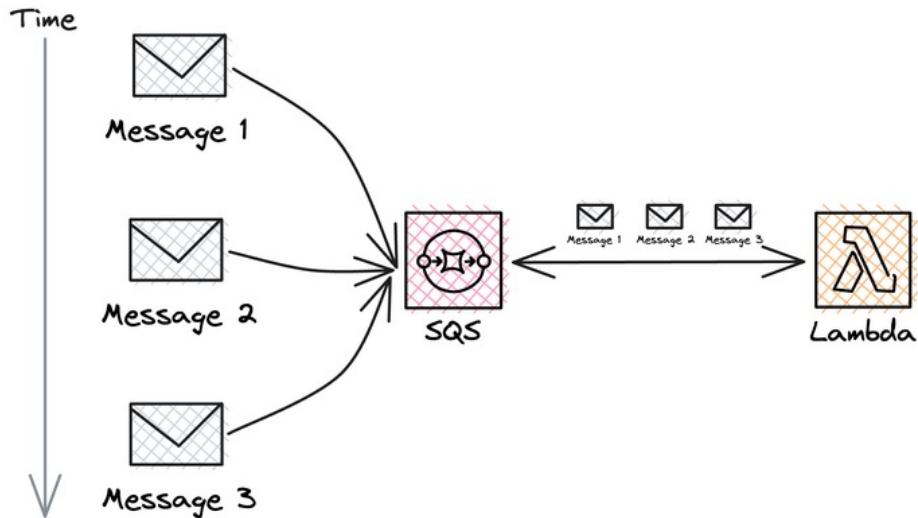
The **Standard Queue** in SQS can process a large number of messages. It follows the **at-least-once** delivery protocol, meaning that calls can occur two or more times.

Standard Queues have much higher quotas compared to FIFO queues. For example, in standard queues, you can have up to 120,000 messages in flight, whereas in FIFO queues, you can only have up to 20,000.

In Standard Queues, the order of your messages is not guaranteed. This means that a message that is 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 Queue



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.

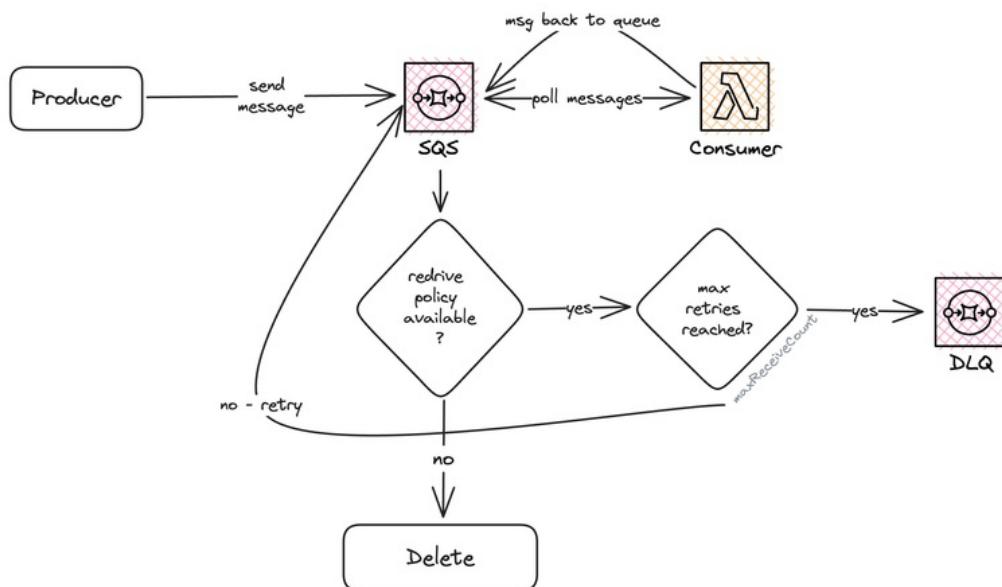
Understanding the difference between the queue types is crucial when working with SQS. You can't change the queue type after creating a queue. A major change is also within its quotas. We will focus on that in the chapter's limitations.

## Why Shouldn't We Use Only FIFO Queues?

One of the main reasons here is cost and limitations. FIFO queues are not only more expensive per request, but they also have much tighter quotas.

Batches can 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 a limit of 3,000 messages per second: 300 requests per second via 10 batched messages. This limit alone can make a significant difference in terms of costs. If you're 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-Queues for Handling Failures & Retries



A very important part of using message queues is handling failures and retries. You can attach Dead Letter Queues (DLQ) to your source message queue. DLQs allow you to save failed messages in another queue. You can execute 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 has 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 redriving. Once you understand and fix 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

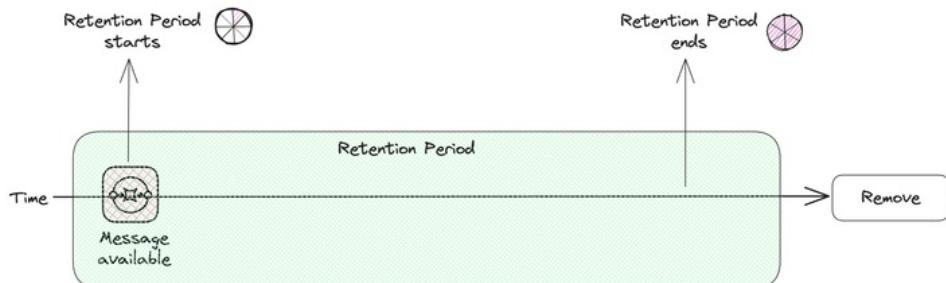
It is best practice to always add a DLQ to every SQS queue you create. Your consumer can always fail. Having a DLQ gives you insights into the reasons for failure. Typically, you attach a CloudWatch alarm to your DLQ to get notified once there are messages in your DLQ. You can then go ahead and check 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 will go through the most important ones here and try to simplify them. We want to provide guidance on how to best configure your queues.

### 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 depends 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 has ended and no consumer has picked up the message. This is often the case if you have busy consumers with a long polling interval.

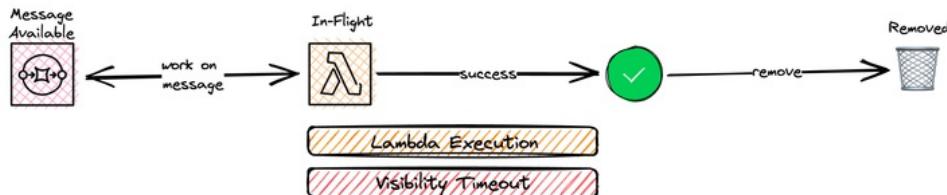
### Hiding Messages from Other Consumers by Defining the Visibility Timeout

The visibility timeout defines how long your message remains 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 begins**.

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 `in-flight` state.

Once the visibility timeout is over and the message has not been deleted from the queue, it changes its state back to `available`.

Setting the correct visibility timeout is not always easy. It is important for your application to work correctly.



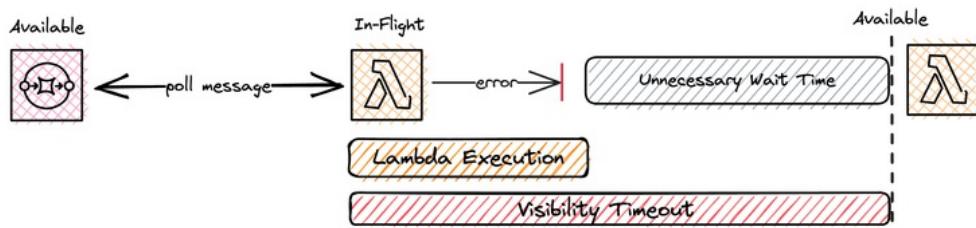
In the best scenario, the visibility timeout is about the same length as the Lambda execution time.

There are two other 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.

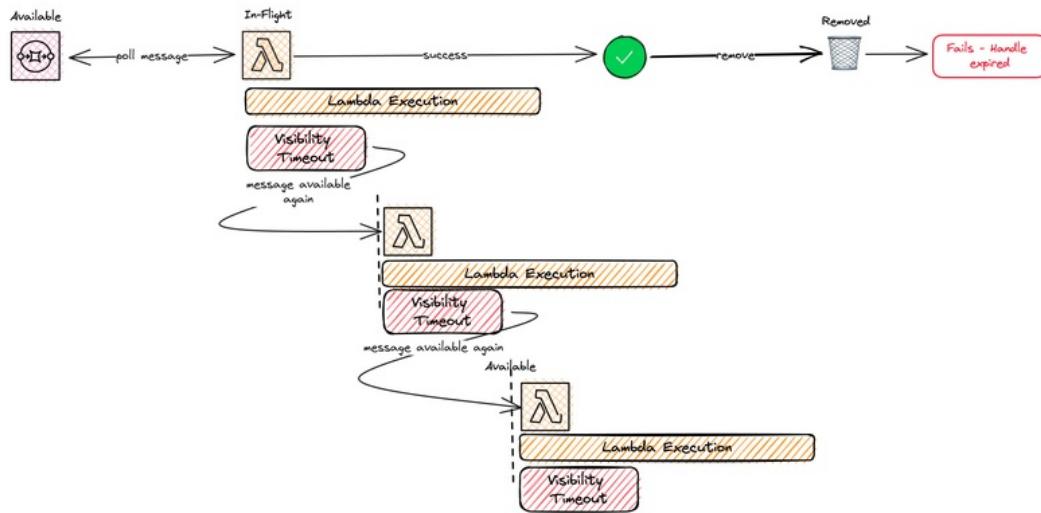
### Visibility Timeout Too Long



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.

### 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.

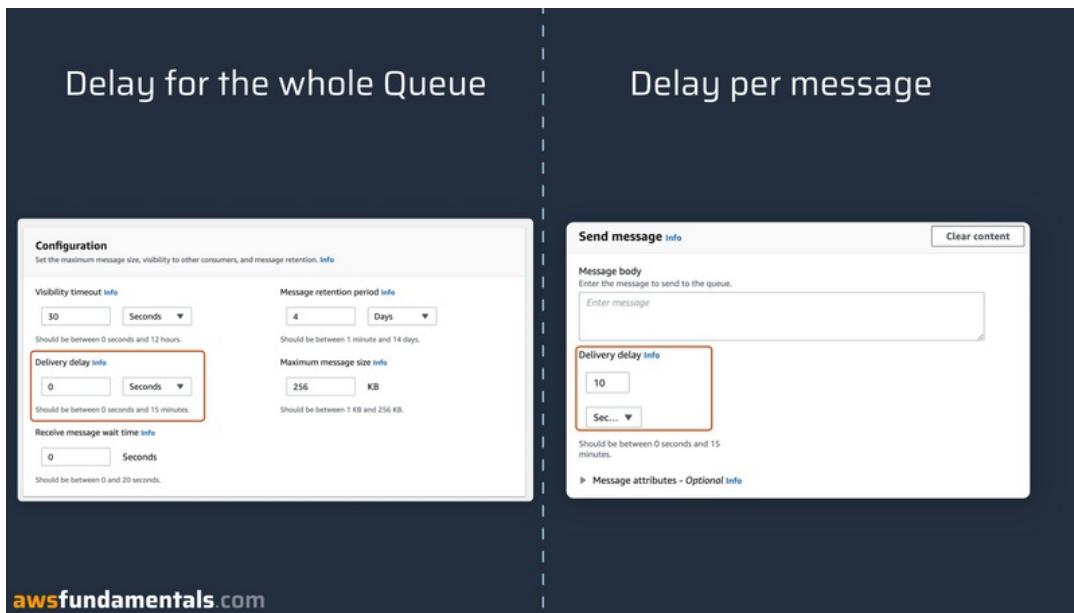
## Set the Visibility Timeout Rather Too High Than Too Low

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 it is if your retry only happens after a couple of minutes.

### Delay Messages With Delivery Delay



You can set up delays for your queue and messages. Consumers cannot pick up messages for the defined time. Delays can have a maximum value of 15 minutes. By default, this value is zero.

You can set this value either per queue or per message. Setting this value per queue means that every message in that queue will be delayed. Alternatively, you can set this value per message. When sending a message, you can define a delay for that specific message. In that case, only the single message will be delayed.

## Long Polling and Short Polling Defines How Messages Are Polled



SQS is a polling-based system, which means that your consumer needs to poll the queue for new messages. SQS charges you based on the number of requests.

A common scenario is to use Lambda as your consumer. The event mapping polls for new messages. If there are no new messages, the SQS API is still called, and 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. 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 queries all servers and waits until a message is available. It will not query a subset of the servers but all servers. This makes long polling much more efficient. SQS only sends a response if at least one message is available or the wait time has expired.

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.

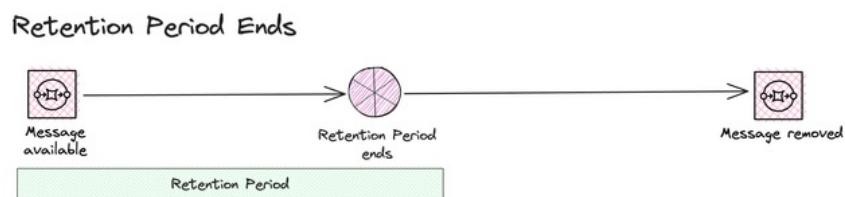
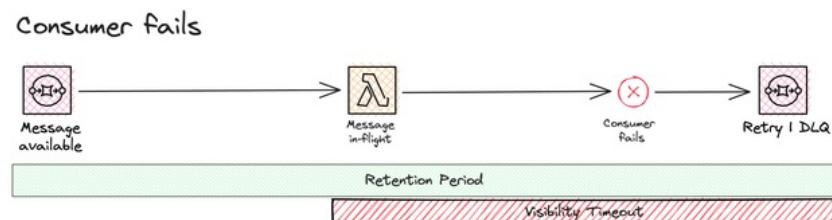
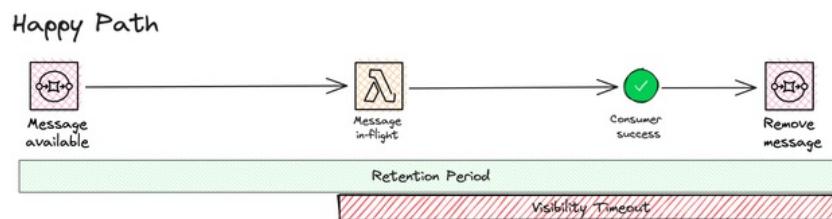
Short polling is activated by default because the receive message wait time is set to 0. With

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 Lifecycles Within SQS: The Happy Path, Consumer Failures, and Too-Short Retention Periods

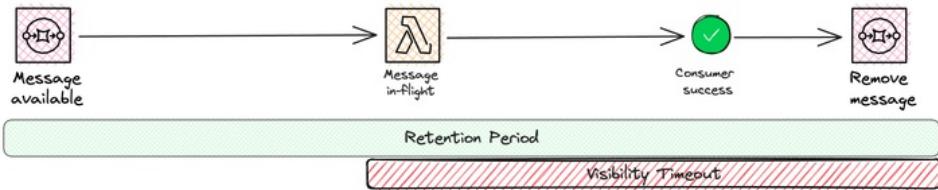
Let's take a look at the big picture. How does all of this come together? There are many different scenarios with SQS messages. We will look at three really common scenarios.

1. The Happy Path: Everything works.
2. Consumer fails.
3. Retention period ends.



## **Happy Path**

Let's see the happy path first.

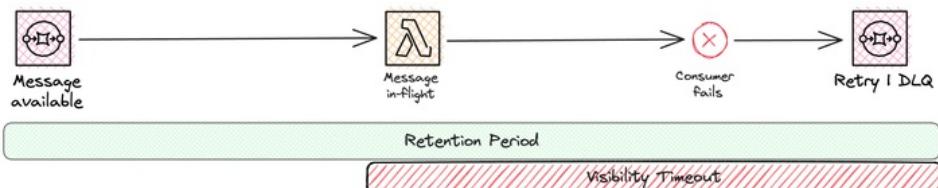


1. The publisher sends a message to the queue.
2. The message is now in the state **Message Available**.
3. The **retention period** time starts.
4. A Lambda function polls for new messages and receives the message.
5. The message changes the state to **In-Flight**.
6. The **visibility timeout starts**.
7. The message is processed successfully.
8. The consumer removes the message from the queue.

Everything worked, and the message is removed from the queue.

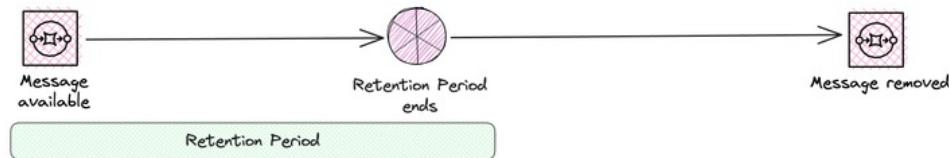
## **Consumer Fails**

The second scenario shows what happens when the consumer fails to execute 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 the 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. The 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 on your queue to ensure that nobody else can access your messages. You can either use SQS-owned keys or provide your own with KMS (Key Management Service). By using KMS, you can also allow your users to 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 significant difference in quotas for the type of queue you use, whether it's a standard or FIFO queue.

Before designing applications in AWS, make sure to understand quotas. Quotas often shape the architecture of your cloud application.

<b>Limit</b>	<b>Standard</b>	<b>FIFO</b>
<b>Message Size</b>	256 KB	256 KB
<b>Messages per queue</b>	∞	∞
<b>Messages In-Flight</b>	120,000	20,000
<b>Operations per second</b>	∞	300 / s Batching: 3,000 / s
<b>Batch Size</b>	10,000	10

### **Message Size**

The default maximum message size is 256 KB, which applies to both standard and FIFO queues. If you have larger items, consider saving data in a database or in S3.

### **Messages per Queue**

Messages per queue are infinite, referring to available messages, not messages in flight.

### **Message In-Flight**

Messages In-Flight are messages already picked up by a consumer, and 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 ensure 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 Based on the Number of Requests**

SQS is a serverless service, which means you only pay for your usage. However, this is only partly true because in SQS, you also pay if your consumer polls for empty messages. This can become quite expensive if you have many queues, such as when 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 generally more expensive. The pricing is tiered, which means the higher your usage, the cheaper the price per request gets. The pricing also varies slightly per region.

Tier - Requests per month	Standard	FIFO
1 Million	Free	Free
1 Million - 100 Billion	\$0.40	\$0.50
100 Billion - 200 Billion	\$0.30	\$0.40
> 200 Billion	\$0.24	\$0.35

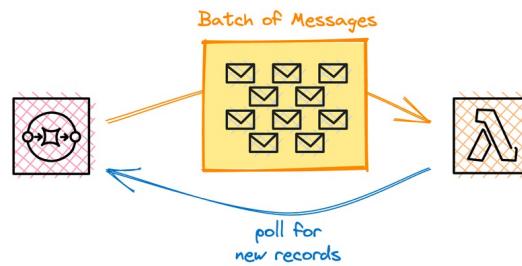
It is free to get started with SQS. Always consider your consumers as well when calculating the

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.

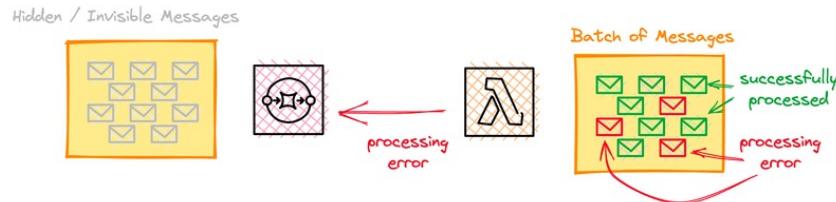
Make sure to think about whether that makes sense.

### Using Partial Batch Responses to Avoid Reprocessing the Same Message Multiple Times

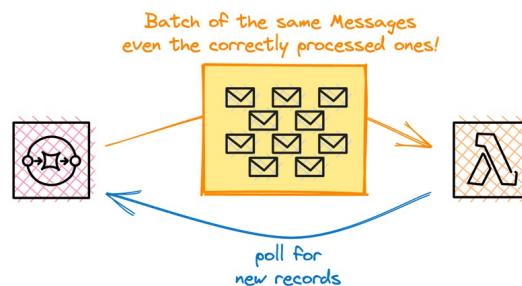
In the event that your Lambda function encounters an error while processing a batch, it is important to note that all messages within that batch will become visible in the queue again by default.



This includes messages that were successfully processed by Lambda.



Consequently, there is a possibility that your function may end up processing the same message multiple times.



To mitigate the issue of reprocessing successfully processed messages within a failed batch, you have the option to configure your event source mapping to only make the failed messages visible again. This is referred to as a partial batch response.

To enable partial batch responses, you can specify `ReportBatchItemFailures` for the `FunctionResponseTypes` action when configuring your event source mapping. By doing so, your function will be able to return a partial success, which can effectively reduce the number of unnecessary retries on records.

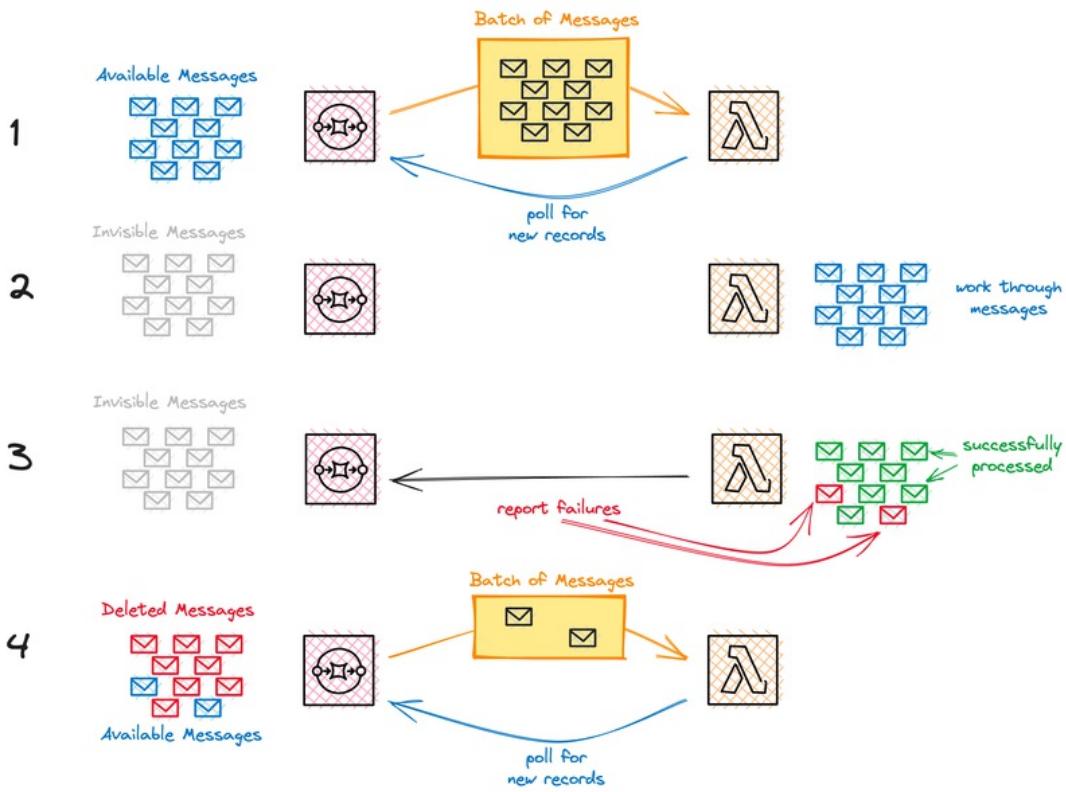
So instead of erroring out in your function, you should catch every processing error and after your function has worked through the batch, return a specific response (the `batchItemFailures` response) as JSON.

Let's consider a scenario where you have a set of five messages, each with a unique technical identifier: `id1`, `id2`, `id3`, `id4`, and `id5`.

Your function has successfully processed messages `id1`, `id3`, `id4`, and `id5`. Now, in order to make messages `id2` visible in your queue once again, your function should return the following response:

```
{
  "batchItemFailures": [
    {
      "itemIdentifier": "id2"
    }
  ]
}
```

This technique is simple and avoids double processing of messages, which can increase your throughput, especially if your processing failure rate is rather high.

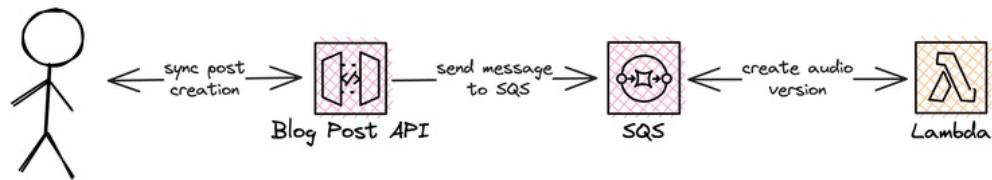


### Use Cases - Building Asynchronous Tasks with SQS

Let's look at three different use cases for SQS. Use cases in SQS always revolve around building event-driven systems and 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 already had this example in the data chapter.



Generating audio can take a while, especially for high-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 generate the audio in this function. Instead, you can send the post to SQS and queue the task. By doing that, you have decoupled your post generation from your audio generation.

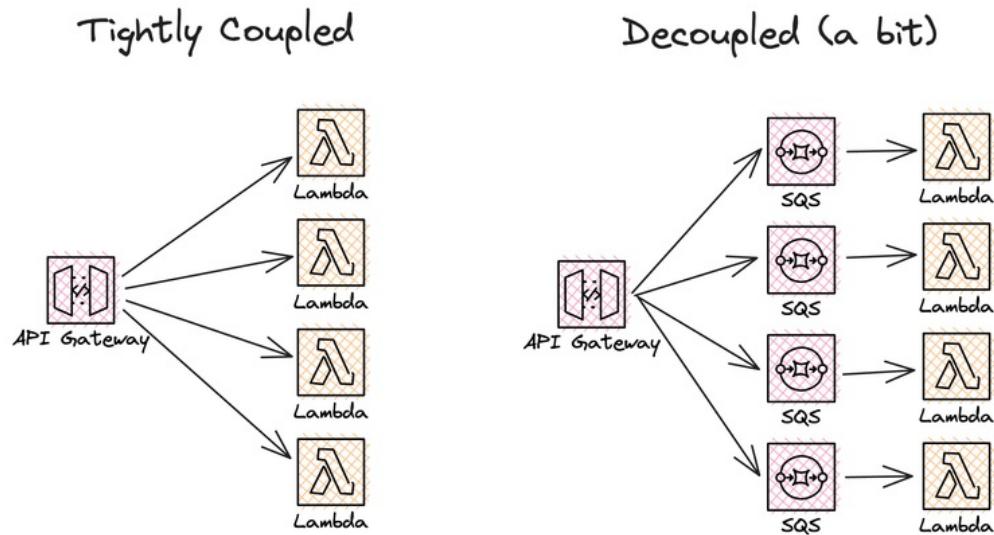
If we did everything in the Create Post API, the end user would need to wait a long time until the blog post is generated. We don't want that. Instead, the performance should be fast.

### 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 an 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 did 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 consumer failure. It can become quite complex if you need to handle failures when running 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 it. 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 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 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 use **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. In 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.

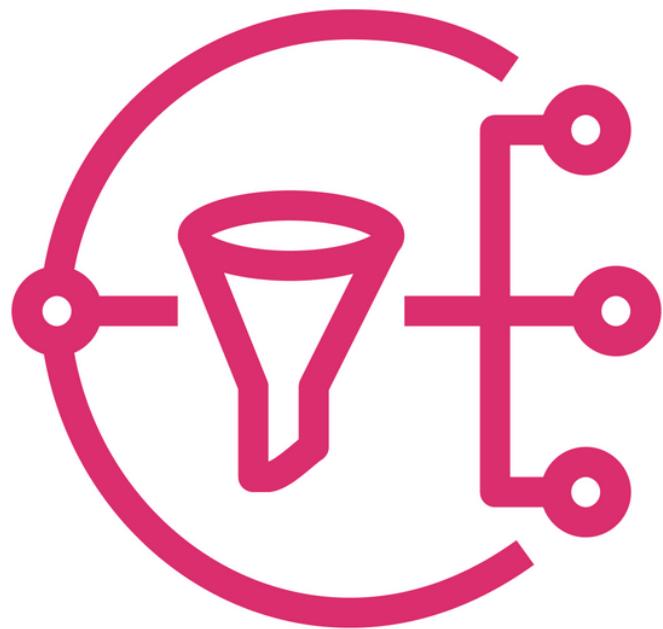
## Final Words

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 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 publishing and subscription service. A publisher sends messages to topics, and subscribers subscribe to those topics. The topic is the distributor of messages in SNS, forwarding all messages to the subscribers.

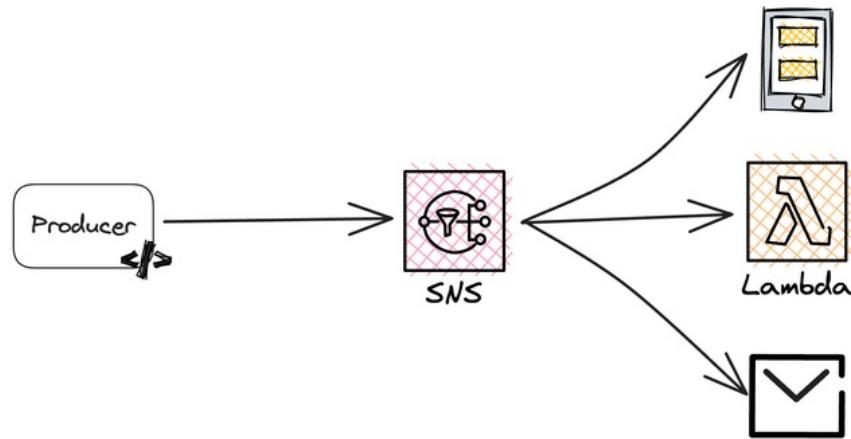
Subscribers can be personal applications, such as smartphone notifications, emails, or SMS, or another AWS Service, such as a Lambda function.

SNS is ideal for high-throughput applications, as it can accommodate up to 12.5 million subscribers per topic.

## **SNS is a Pub/Sub Service - Consumers Subscribe to Topics and Producers Publish Messages via Topics**

SNS publishes messages to subscribers with a push-based model. Once a message comes in, SNS immediately pushes the message out to its subscribers.

On the other hand, SQS uses a poll-based model. This means that a message remains in a queue and consumers poll for this message.



Consumers subscribe to topics, similar to 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

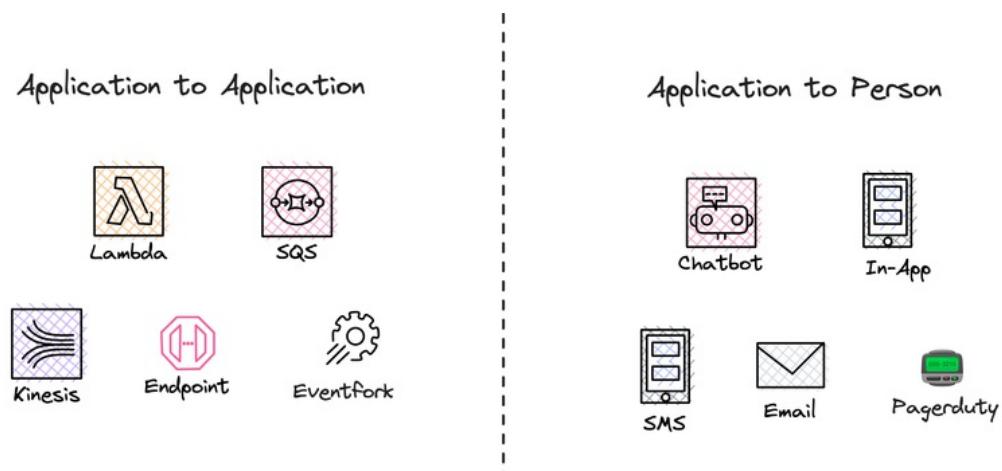
know which topic to send their message to.

This decouples the systems a lot. In an SNS context, you have:

1. **Publisher** - for example, your REST API.
2. **Topic** - a topic to which 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 have understood that SNS can send messages to another AWS service or to a personal device. Let's take a closer look at which services are exactly supported.



SNS distinguishes endpoints as **Application to Application** and **Application to Person**.

The picture above shows all available services. Let's go through each category.

#### Application-To-Application

Application to Application sends messages to other applications, not to people. This is often used to process messages further, either before sending them to a final customer or 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 send 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. **PagerDuty** - there is an official integration with PagerDuty. You can get informed about CloudWatch alarms, for example.

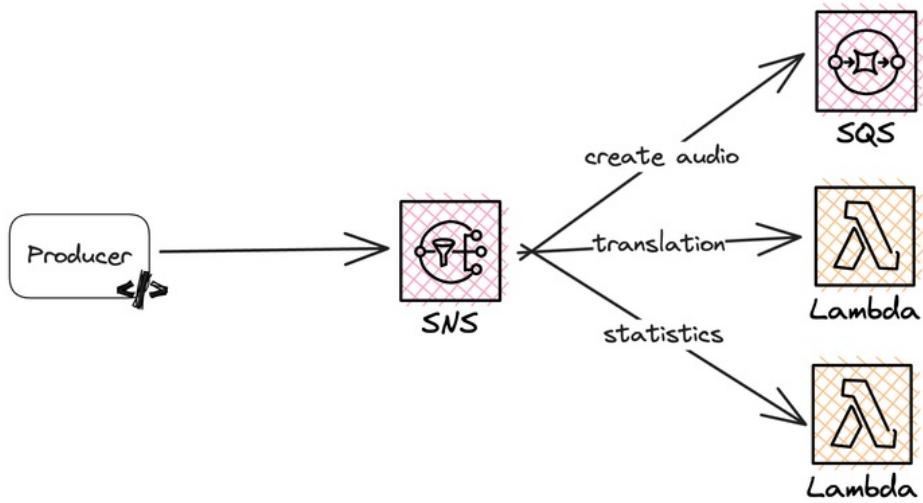
#### **The Fanout Pattern Allows Distributing One Message to a Variety of Services**

The fanout pattern is a very popular pattern in SNS, especially useful for building event-driven systems.

The fanout pattern is about event handling. SNS takes an incoming event and distributes it to many consumers. This is also a major difference in how SQS handles events. SQS only has one consumer, whereas SNS has many consumers.

This is the default behavior of SNS, so there is no 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 where users upload posts. After an upload happens, three actions should start:

1. **Translate** the post into several languages.
2. Create an **audio** version of the 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, while two different Lambda functions handle the translation 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 only be 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, both 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 entire encryption process on the server side. You can either use a key from 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 such as names and attributes
- Message metadata such as 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 between 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 preserve the order of your messages. That means SNS can send your messages in a different order than they were received.

Remember that SNS is a push-based service, not a poll-based service. SNS pushes out the messages as soon as 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 maintain the correct order. For example, if you want to send out notifications to your users in a specific order like:

1. Order came in
2. Order is processed
3. Order is sent out

If the time between each message is low, we 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 **at-least-once** processing, like how SQS behaves. This is due to the nature of a distributed cloud architecture. In our experience, this happens far less with SNS. It is still

important to keep that in mind. The deduplication ID ensures that SNS only sends your message once.

### **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 period.

This can be very useful in the following scenarios:

- You have 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 does not have a built-in archive (unlike EventBridge), but you can use **Kinesis Data Firehose** for that purpose.

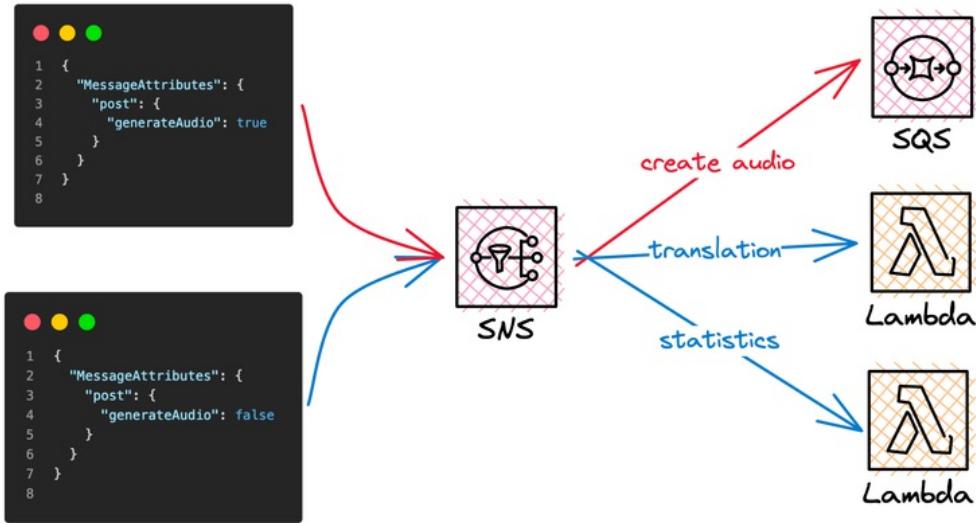
Kinesis allows you to save all incoming messages in S3 or Redshift. You can also build Lambda functions or Athena queries for the above use cases.

### **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, allowing for a flexible routing mechanism.

The filter policy can be applied to both **Message Attributes** and the **Message Body**.

Let's look at 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 Work for Server-Side Errors

For each delivery protocol, SNS defines a **delivery protocol** with which you can define how retries work. 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 sent out, it is considered 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 this:

```
{  
  "http": {  
    "defaultHealthyRetryPolicy": {  
      "numRetries": 3,  
      "numNoDelayRetries": 0,  
      "minDelayTarget": 20,  
      "maxDelayTarget": 20,  
      "numMinDelayRetries": 0,  
      "numMaxDelayRetries": 0,  
      "backoffFunction": "linear"  
    },  
    "disableSubscriptionOverrides": false  
  }  
}
```

You will define the following parameters:

- 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 of multiple errors?

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 message is only considered failed if a **server-side error** occurs, as we discussed in the previous section.

It is always best practice to use DLQs. In a worst-case scenario (e.g. accidentally removing an IAM policy), your DLQ will save the failed message. Each failed message is available in your

DLQ. After fixing the bug, you can resend the message to the topic or subscriber.

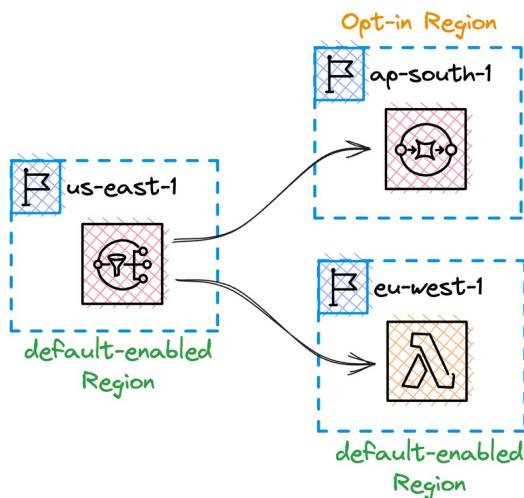
This is particularly important for notifications. Imagine you want to send notifications to all your users' smartphones, but the message fails. The message will be lost if you don't use a DLQ.

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 cover in the next chapter.

**Remember: use DLQs.**

### SNS Can Deliver Messages to SQS Queues and Lambda Functions in Other Regions

With Amazon SNS, you can deliver notifications to both Amazon SQS queues and AWS Lambda functions across different regions. However, when one of the regions is an opt-in region, it is important to specify a different Amazon SNS service principal in the policy of the subscribed resource.



Opt-in regions are regions that are launched after March 20, 2019, which means that most regions do support cross-region delivery by default. The opt-in regions are currently:

- Africa (Cape Town)
- Asia Pacific (Hong Kong)

- Asia Pacific (Jakarta)
- Europe (Milan)
- Middle East (Bahrain)

For SQS subscribers you can deliver messages only to default-enabled regions, meaning that you for example can do the following things:

1. us-east-1 (default-enabled) to eu-west-1 (default-enabled)
2. af-south-1 (**opt-in**) to us-east-1 (default-enabled)
3. us-east-1 (default-enabled) to af-south-1 (**opt-in**)

As mentioned before, both the second and third one needs an adjustment of the service principal policy. For the second one, it could look something like this:

```
{
  "Version": "2008-10-17",
  "Id": "PolicyID",
  "Statement": [
    {
      "Sid": "allow_sns_arn:aws:sns:af-south-1:123456789012:source_topic",
      "Effect": "Allow",
      "Principal": {
        "Service": "sns.af-south-1.amazonaws.com"
      },
      "Action": "SQS:SendMessage",
      "Resource": "arn:aws:sqs:us-east-1:123456789012:destination_queue",
      "Condition": {
        "ArnLike": {
          "aws:SourceArn": "arn:aws:sns:af-south-
1:123456789012:source_topic"
        }
      }
    },
    ...
  ]
}
```

For Lambda, this looks similar with the difference that you can't deliver any messages to opt-in regions.

If you're delivering messages to HTTPS endpoints, you don't have to worry about regions. The requirement for utilizing AWS is not necessary in this case. As long as your HTTPS endpoint is accessible from the Internet and has a valid SSL certificate, SNS will function properly. In this setup, the destination region and destination AWS account become irrelevant. There are no constraints or limitations in this scenario.

### **Pricing is Based on the Number of Requests**

SNS is a serverless service. You won'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**.

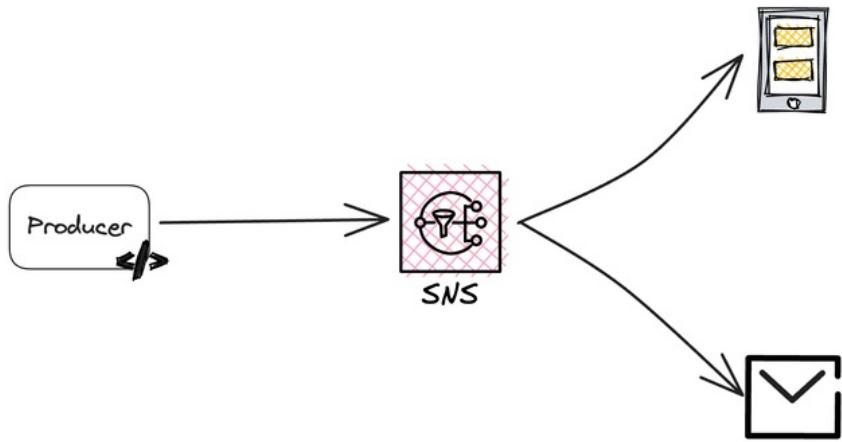
### **Typical Use Cases are CloudWatch Alarms and In-App Notifications**

Let's look at some example use cases to better understand this service.

#### **Use Case 1: SNS can be used to send In-App Notifications**

One common use case is using SNS to send In-App Notifications.

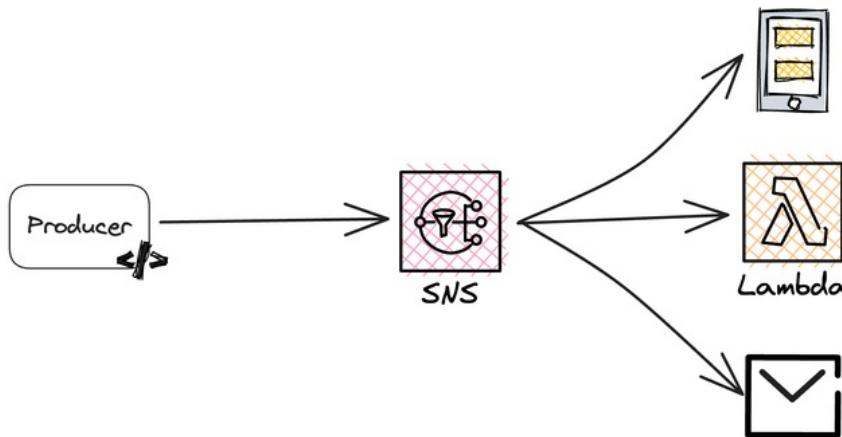
Most modern applications need to send In-App Notifications in different cases. A good example is a social media network. Once a user receives a comment or a like on their post, they should be informed in the app.



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.

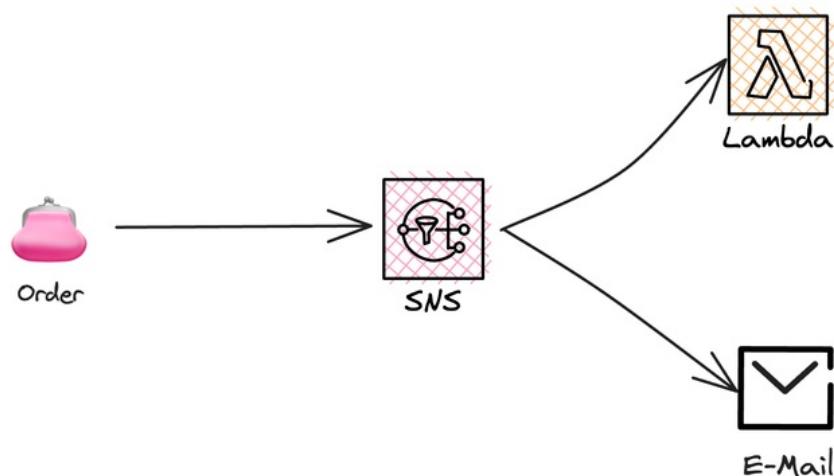
#### **Use Case 2: CloudWatch Alarms Trigger SNS Topics to Send Out Messages**

A common example is the use of CloudWatch alarms, which is the monitoring and alerting service within AWS. Each log and metric about your services lives in CloudWatch. With CloudWatch, you can create alarms, such as one for failures of your Lambda function. Once it fails, you will be informed.



Your CloudWatch alarm uses SNS to send an automated message to subscribers once the alarm goes off. Subscribers can be email, SMS, or a 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 real-time notifications, not just for mobile applications like phones or tablets, but also to trigger different systems or combine both.

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 distribute 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

## Event-Driven Architectures Decouple Producers and Consumers 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.

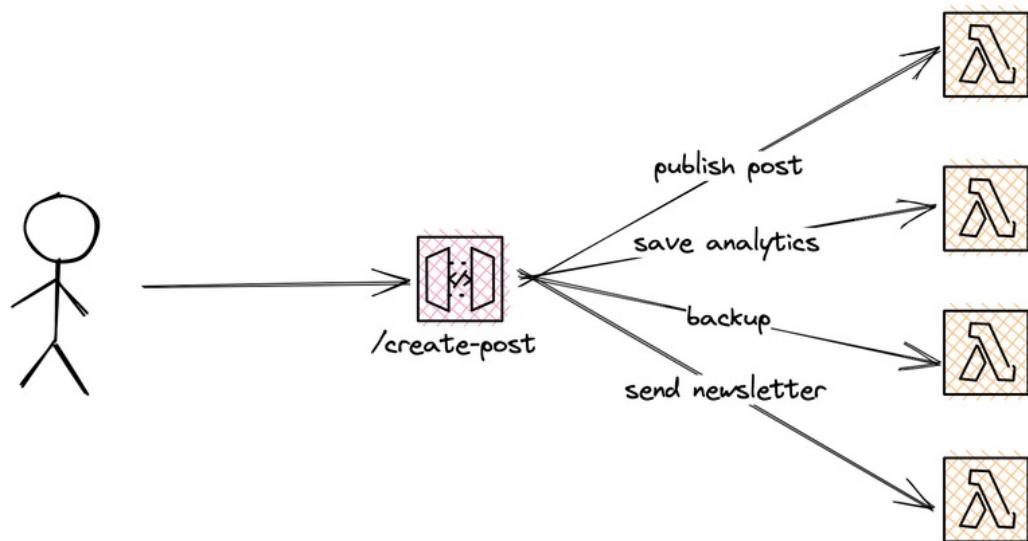
**Decoupling your producers and consumers** is one key factor in an EDA.

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 would normally start. By using this approach, we quickly see the pain points.

For example, let's imagine we are building a blogging platform. A user publishes a post and calls the `createPost` endpoint. This endpoint does many things:

1. Publishes the post
2. Saves analytics
3. Makes a backup of the post
4. Sends 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 and error handling**, it gets much more complex. What happens if one of these services becomes unavailable? Do we need to revert all changes? Should we retry it and let the user wait?

All these scenarios do not provide a good developer experience. This is often the point where developers start looking for alternatives. Alternatives often include:

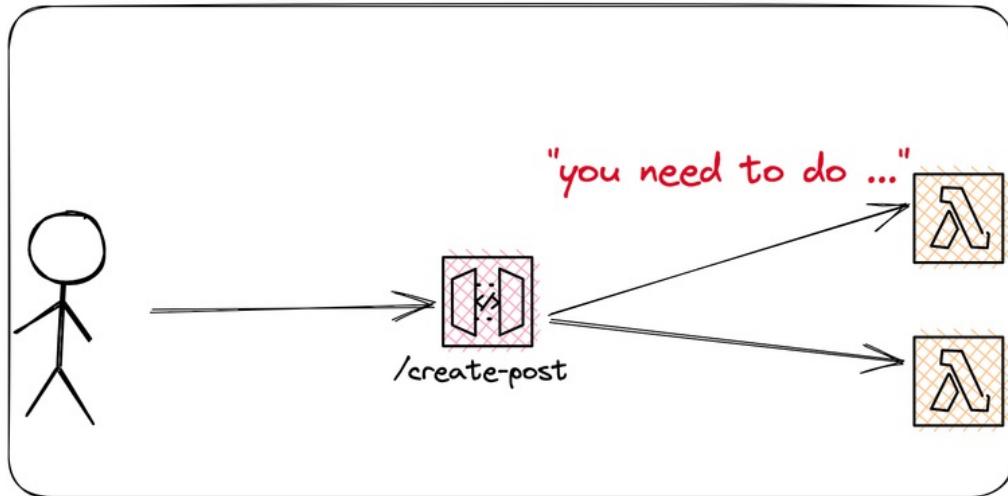
- Message queues like SQS
- Pub/Sub Services like SNS
- Event routes like EventBridge

With EventBridge, you'll get the most flexibility to decouple your architecture.

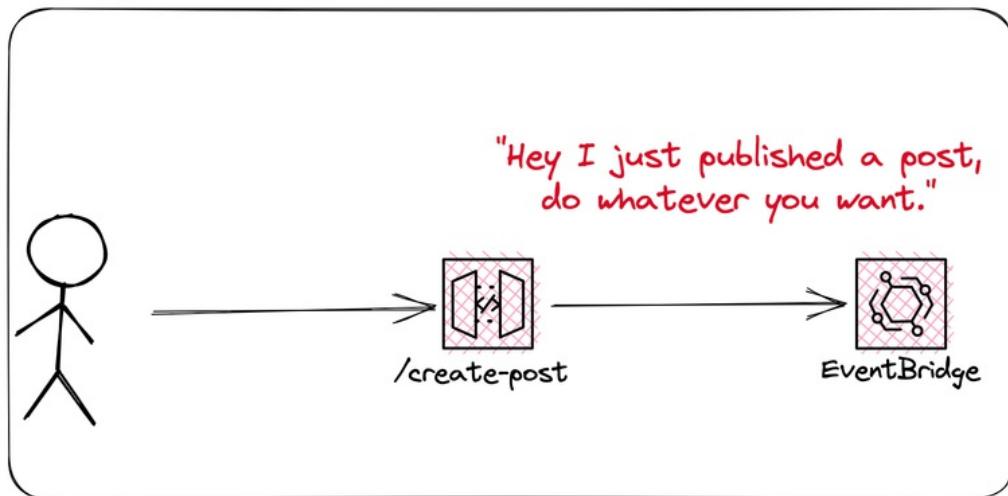
#### **EDA Decouples Your Architecture by Introducing Additional Components**

EDA helps you solve these issues. The main goal is to decouple the producer and consumer of events. EDA solves this issue by introducing an additional component, the event router.

## API-Driven



## Event-Driven



The `createPost` endpoint will only create the event and send it to the router. This is where the endpoint's responsibility ends.

The main difference between the API-driven and event-driven ways is how communication occurs. In the API-driven way, the API instructs 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 informs the router that something happened without

caring about what happens next: "***Hey, I just published a post.***"

We are not directing other consumers to perform tasks. We only inform the router about what happened and it decides what happens next.

We will discuss the above components in more detail soon. But to summarize, EDA has the following benefits:

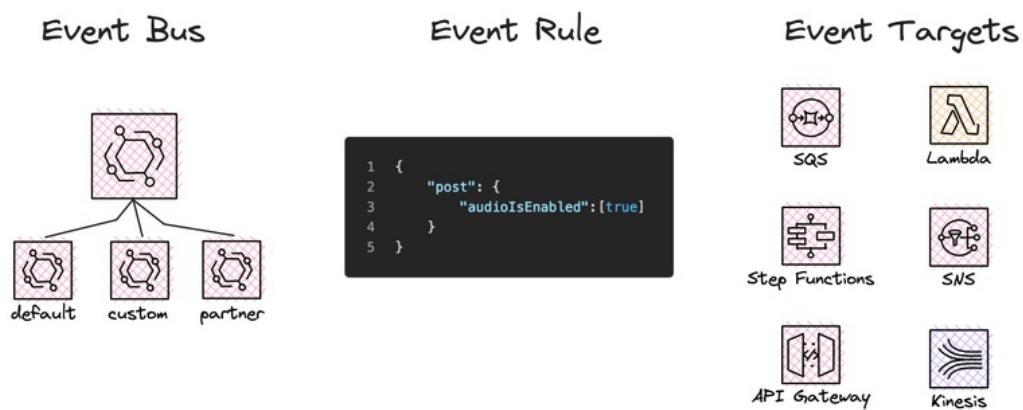
- The architecture is more **decoupled**.
- Tasks run in the **background**, and the most crucial task (publishing the post) executes quickly.
- **Adding** services to events **is much easier** because they can subscribe to events.

More of these benefits will become clearer after the entire 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 this:

```
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 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 include MongoDB, Auth0, or Zendesk. To receive and act on these events, you will create a partner event bus.

## Event Rules Define How to Route Events to Their Consumers

The second component is an **event rule**, which is one of the most powerful features of EventBridge.

The event rule describes how events are routed. Once an event arrives, the rule sends it to the consumer if the event pattern matches.

AWS describes this as **filter patterns**. This can be confusing for new starters on AWS. You are not actually filtering anything, but rather trying to **match** fields in the event.

Let's look at 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 this is a simple configuration. There is **no code** involved in forwarding the event. Less code is often better.

EventBridge has different event patterns. Event patterns define how events are matched. In the above example, our defined event pattern would look like this:

```
{  
  "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 for integration. 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 asynchronously. That means EventBridge does not wait until your consumer finishes. EventBridge only knows whether it delivered the message or not. The service has no information about the inner workings of your services. For example, it does not know if your Lambda function 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. Take a 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** to consider when dealing with 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, such as `version`, `account`, and `time`. However, the three mentioned above are the most important ones.

#### **Detail Is Your Actual Message**

In the `detail` section, you include your entire event message. We have seen 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 also match on the `detail-type`. Detail types often follow a convention of noun + verb. For example:

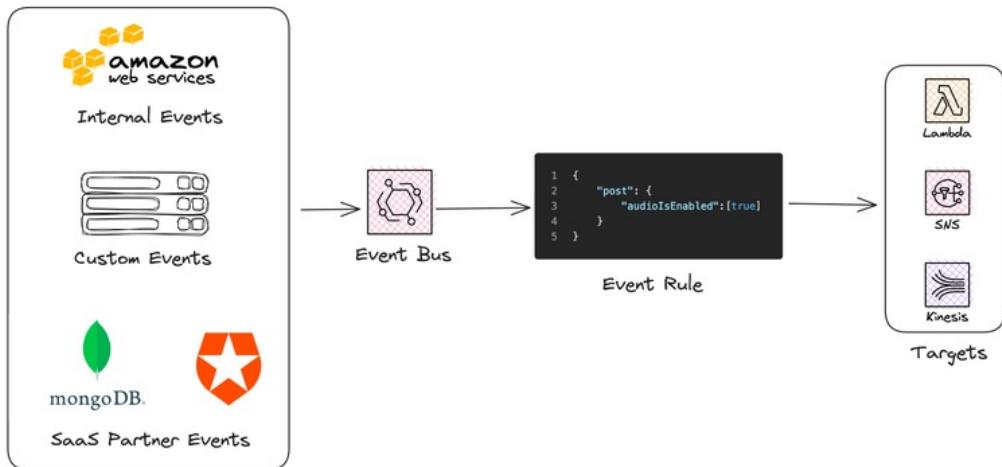
- Post published
- Order created
- Customer deleted

## 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 use `organization.api`.

## Event Sources Are Either Internal AWS Events, Custom Events, or Partner Events



## 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 come 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 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 match, 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, as mentioned before, including 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 homogeneous 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 using either 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 this: `0/10 * ? * MON-FRI *`

CRON has six different groups:

Number	Group
1	Minutes
2	Hours
3	Day of the month
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 use cases, this is sufficient. Please consider your fellow developers when creating complicated CRON jobs.

### Create a One-Time Schedule for Ad-Hoc Tasks

The latest innovation in the EventBridge Scheduler is the ability to create one-time tasks. Prior to 2022, this had to be built manually using services such as step functions, cron jobs, or Lambda functions.

Now, this functionality is built-in, making life much easier.

A common use case is to schedule emails for new user sign-ups after a specified timeframe, such as 7 days.

The screenshot shows the 'Schedule pattern' configuration page. At the top, there are two tabs: 'Occurrence' (selected) and 'Info'. A note below says 'You can define an one-time or recurrent schedule.' Below the tabs are two radio buttons: 'One-time schedule' (selected) and 'Recurring schedule'. Under 'Date and time', there are fields for date ('2022/12/08'), time ('12:24'), and timezone ('(UTC +01:00) Europe/Berlin'). There is also a note: 'The date and time to invoke the target.' Under 'Flexible time window', there is a dropdown menu set to 'Off'. A note below it says: '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.'

Define a date, time, and flexible time window. This task will be executed **at least once** during this time period. EventBridge has a one-minute buffer to execute these tasks, so keep that in mind if you need more precise execution.

The scheduler is still in active development. In 2023, the functionality was launched to auto-clean up your created schedules after they are finished. This is a huge help in maintaining a lean cloud architecture.

### **Automatically Delete Schedules Upon Completion**

Since August 2023, EventBridge Scheduler also offers automatic deletion of schedules upon completion.

Previously, completed schedules were counted towards the account quota limits and required manual removal. However, with the new automatic deletion feature, EventBridge Scheduler deletes schedules shortly after their last target invocation. This setting can be configured for both one-time and recurring schedules, ensuring that completed schedules are removed efficiently.

- **One-time schedules** are used when you only need to invoke the target once. After the schedule has successfully invoked its target, it is automatically deleted.
- **Recurring schedules**, on the other hand, are set with either a rate or cron expression. These schedules are used when you need to repeatedly invoke the target at specific intervals. Similar to one-time schedules, recurring schedules are also automatically deleted after the last invocation.

#### **Action after schedule completion**

##### **Action after schedule completion**

If you choose DELETE, EventBridge Scheduler will automatically delete the schedule after it has completed its last invocation and has no future target invocations planned.



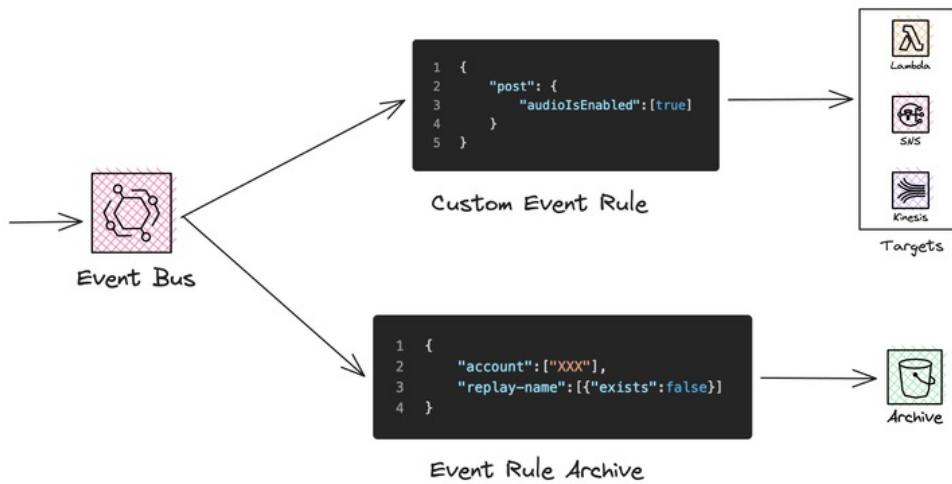
In the case of a schedule configured with automatic deletion, if all retries are exhausted due to failure, the schedule will be deleted shortly after the last unsuccessful attempt. This ensures that any unsuccessful attempts do not clutter the system.

If you're creating a schedule via the AWS CLI, you only need to pass the new parameter `--action-after-completion:`

```
aws scheduler create-schedule --name SingleExecution \
--schedule-expression 'at(2023-08-23T15:00:00)', \
--schedule-expression-timezone 'Europe/Berlin' \
--flexible-time-window '{"Mode": "OFF"}' \
--target '{"Arn": "arn:aws:lambda:eu-west-1:xxx:myfunc", "RoleArn": "arn:aws:iam::xxxx:role/myrole" }' \
--action-after-completion 'DELETE'
```

### Archive & Replay Lets You Save All Events and Replay Them Back to Your Event Bus When Needed

EventBridge allows you to create an archive that saves all your incoming events. This can be helpful if you encounter a bug and need to replay failed events. It is also a good opportunity to obtain production data for development.



### The Archive Stores All Events You Send to Your Bus

The archive stores all of your events for a defined period of time. You can also choose to 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. This means that you can't view the events in their JSON format. The archive is only there for its replay capability.

When creating the archive, you select the timeframe for how long you want to save the events. You can also choose to save the events forever. The archive itself doesn't cost anything. You only pay for the storage costs on S3, which 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 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 still be 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 a single event**. This functionality is really amazing and helpful.

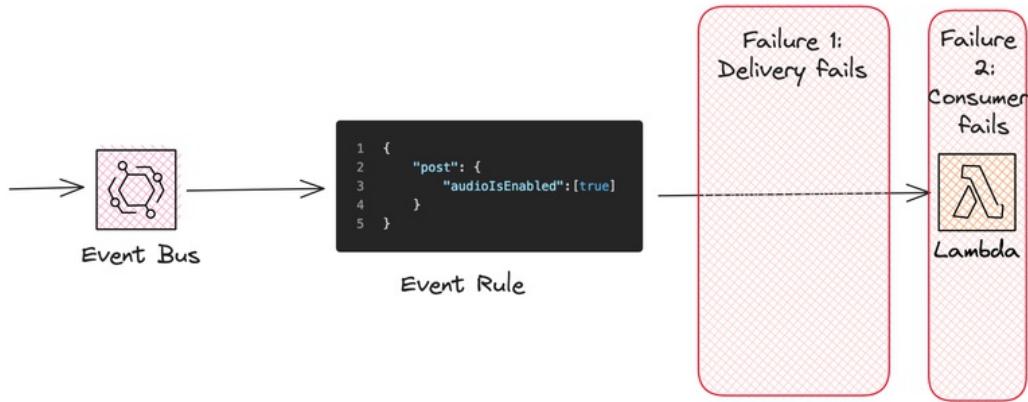
The main benefit here is that this function is completely managed by EventBridge. You don't have to implement anything. You simply tick a box and EventBridge saves all of your events in the archive. You can achieve a similar behavior on SNS, but in SNS, you need to build it yourself with Kinesis, S3, and Lambda.

## **Handle Failures by Using Dead Letter Queues (DLQs) for Target Delivery**

Quoting Werner Vogels, CTO of Amazon: "*Everything fails, all the time.*"

Your service will inevitably experience some kind of failure. It is important that you know how to handle these failures.

In EventBridge, there are **two ways** in which errors can occur:



### 1. EventBridge cannot deliver the event to the target.

The first scenario is when EventBridge fails to deliver the event. For example, if your target service is Lambda and the Lambda API is currently down, the event forwarding will fail.

Another, more common example is missing IAM permissions. If your event bus is not authorized to invoke a Lambda function, it will fail.

We have already learned about the concept of Dead-Letter-Queues (DLQs). 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.

**It is important to note here:** This is only about the **delivery** to the Lambda service, **not** about a failure **within** the Lambda function. This is similar to server errors we have seen in SNS and SQS.

#### 1. The consumer fails

The second scenario is more common. **Your target service fails.** If we go back to our example, our Lambda function would fail.

If the Lambda function fails, **EventBridge won't receive 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 see where errors can happen and what happens with the events afterward.

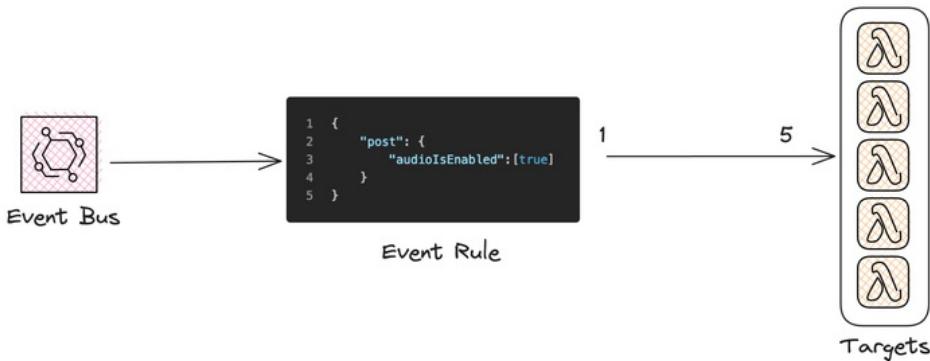
### The Subscription Pattern Defines That Rules Should Belong to One Consumer

We mentioned earlier that EventBridge has a fixed quota of **5 targets per rule**. Although this may seem like too few, we need to examine what it actually means. This chapter is about how to design your event rules, which is something to consider before implementing anything. This is a crucial topic, but it can also be a bit advanced. Please stay with us!

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 assume that the target belongs to the event rule, and the rule decides who should receive the event and who shouldn't, as shown in the following picture:



This has some drawbacks in the long run. First of all, the limitation of 5 targets would indeed be an issue. But, secondly, and more importantly, the rule should **belong to the target**, not the target to the rule. In this design, all targets belong to one rule. That means the rule is the "owner" of the target. However, the target or the consumer of the event has to decide which events and, therefore, which rules it wants to consume.

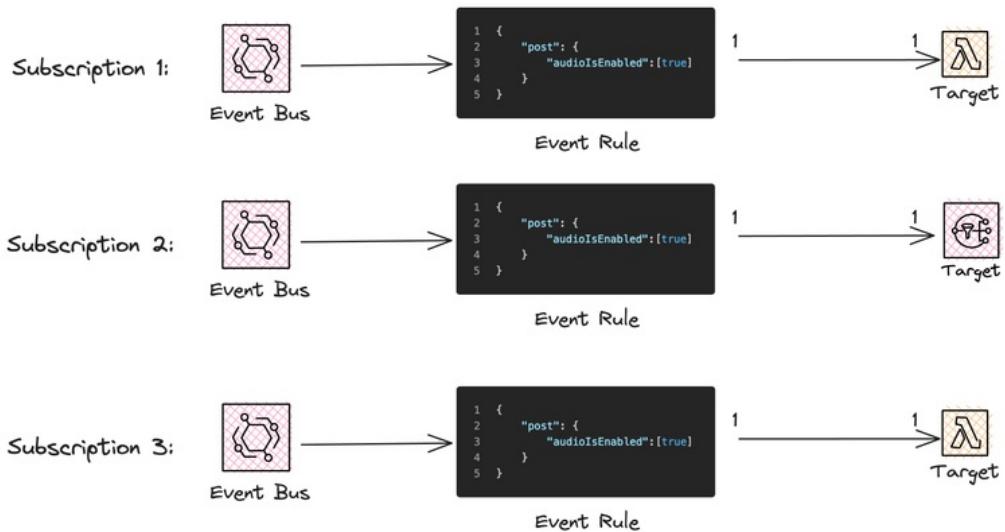
We use EventBridge to **decouple** our architecture. We don't want to increase coupling by

coupling targets to the rules. The consumers should be able to **flexibly subscribe and unsubscribe** without the need to change 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 each of them. 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 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 are getting 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.

## 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, and 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 formats, and they are versioned. Each time the schema changes, you can create a new version.

### Internal AWS Schemas Are Already Available

For internal AWS events, many schema bindings are available, making it really easy to get started working on AWS events.

You can see all schemas by going to the EventBridge console → Schema Registry → AWS event schema registry.

This shows you all the different schemas. If you're interested in events about ECS, search for **aws.ecs**.

The screenshot shows the AWS EventBridge Schema Registry interface. At the top, there are four 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 search input field containing 'aws.ecs'. To the right of the search input are navigation icons for back, forward, and search history. The main area displays a grid of schema entries. There are two rows of two columns each. The first column in the top row contains the schema name 'aws.ecs@ECSContainerInstanceStateChange' and its details: 'AWS event schema registry' and 'Found in version 1'. The second column in the top row contains the schema name 'aws.ecs@ECSServiceAction' and its details: 'AWS event schema registry' and 'Found in version 1'. The first column in the bottom row contains the schema name 'aws.ecs@ECSTaskStateChange' and its details: 'AWS event schema registry' and 'Found in version 1'. The second column in the bottom row contains the schema name 'aws.ecs@AWSAPICallViaCloudTrail' and its details: 'AWS event schema registry' and 'Found in version 1'.

Let's take the event **aws.ecs@ECSContainerInstanceStateChange** as an example.

It shows you the details of an `ECSContainerInstanceStateChange` event. This event occurs when a container in ECS changes its state, for example, from stopped to started.

Let's focus on the `detail` object since it is the most interesting one. The `detail` object refers to another object called `ECSContainerInstanceStateChange`.

```
"detail": {"$ref": "#/components/schemas/ECSContainerInstanceStateChange"}
```

If we look into the schema and find this object, we can 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 of 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 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 within 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 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. The top navigation bar has 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 labeled 'Search discovered event schemas' with a placeholder 'Search schema titles and contents.' To the right of the search bar are navigation icons for pages and a refresh symbol. The main content area displays a single schema entry: 'organization.api@PostPublish'. Below the title, it says 'Discovered schema registry 1 version' and '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 following attributes:

- `content`: a string
- `audioEnabled`: a boolean

### **Automatically Creating Code Bindings for Your Discovered Schema**

We can take this even further and create code bindings, which make the schema available in our source code so that we can work with it.

EventBridge allows code bindings to be created 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 the consumer or producer side.

### **Schema Discovery Cons - Long Time to Create a Schema, and No Optional Fields**

Schema discovery is an amazing functionality of EventBridge that makes many things easier. However, it also has some drawbacks that we have encountered in the past. The main ones are:

- **Time to change a schema** - changing or creating a schema takes about 5 minutes.  
While this is acceptable in a production scenario, it is a long time for actual development.
- **Optional fields** - unfortunately, the schema discovery doesn't work with optional fields.  
For example, if our `audioEnabled` 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 have seen when working with the discovery service. It still helps a lot with starting to work with EventBridge. However, it also forces you to use EventBridge in

a certain way that is not always optimal for every use case.

## **EventBridge Encrypts Your Data at Rest, IAM Secures All Services**

EventBridge encrypts your data at rest by default, and HTTPS secures encryption in transit. IAM secures access control between services.

EventBridge uses encryption at rest, which means the service encrypts all your data on the actual server. It uses a default 256-bit AES encryption, which 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 Safeguard 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 this:

```
{
  "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 for Event Buses, Publishing Events per Second, and More**

Let's take a look at some of the most common quotas in EventBridge. Quotas can vary by region. There are many quotas around API Destinations that we did not cover in detail. We outline 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>	Maximum of 2048 characters
<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.

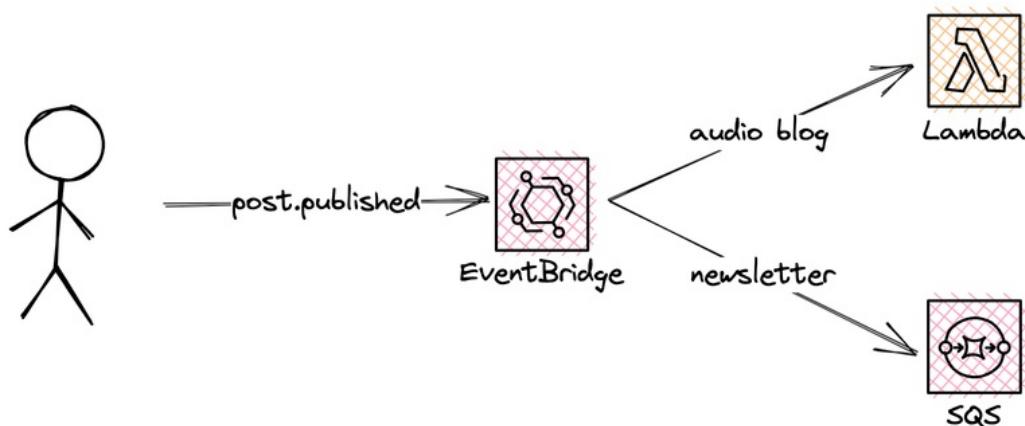
Description	Price
<b>AWS Service Events</b>	Free
<b>SaaS Partner Events</b>	\$1.00 / million events
<b>Cross Account Events</b>	\$1.00 / million events

### Three Example Use Cases for EventBridge

EventBridge is an amazing tool for building event-driven architectures. Its use cases include every task where you try to decouple your architecture. Let's take a look at three example use cases.

#### Use Case 1: Event-Driven Architecture with a Blogging Platform

We have seen the example of a blogging platform throughout this chapter. This is one of the standard use cases for using EventBridge. EventBridge helps you decouple your architecture by allowing you to work on tasks asynchronously.

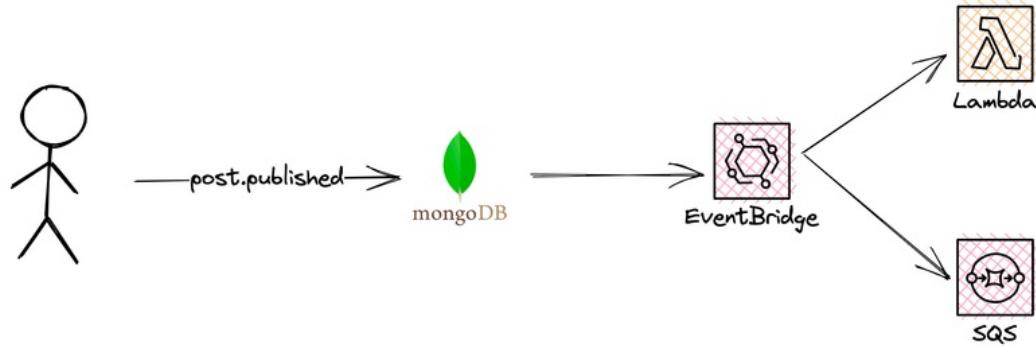


An example architecture could look like this. A user publishes a post and sends the event `post.published` to EventBridge. EventBridge has rules to match this event. The targets for this rule are one Lambda function to process audio blogs and an SQS queue to handle newsletters.

#### Use Case 2: Integrating Third-Party Vendors - Integrate MongoDB

The second use case is to integrate third-party vendors into AWS. Having all workloads in AWS often makes the observability and development of applications much easier.

However, there are cases where third-party vendors such as MongoDB, Autho, or Zendesk are used. EventBridge allows for easy integration with them.



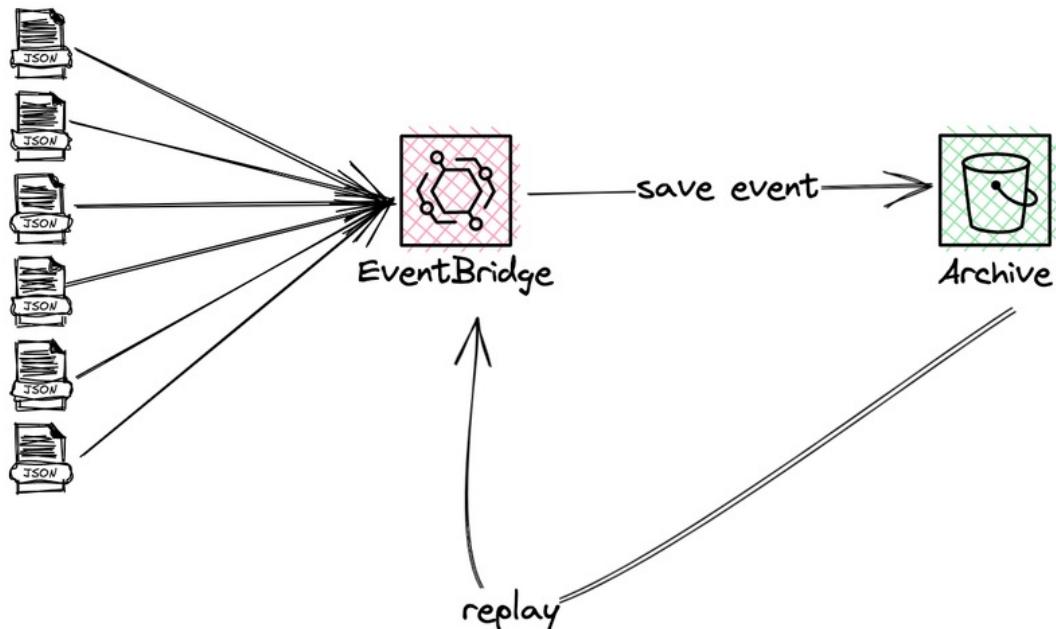
MongoDB, for example, has native integration with EventBridge. EventBridge can listen to triggers such as data being **inserted**, **updated**, or **removed**. This event will be present in EventBridge and can be handled like any other event.

In this example, we can listen to published posts on MongoDB. Once a post is published, we can take certain actions such as creating an audio version or sending out newsletters.

#### Use Case 3: Recovery after Incidents with Archive & Replay

Use Case 3 is about recovering from incidents. EventBridge has the ability to create an archive and replay all events from that archive. The archive saves all incoming events.

In the event of an incident, it is possible that all incoming events failed. With Archive & Replay, you can send all events from 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.
- Create a **validation** for event schemas and incoming events. If you use Node & TypeScript, we recommend looking at libraries such as ZOD and Middy.
- 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 homogeneous 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 maintain 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 is important to understand the fundamental principles of modern networking infrastructure, as 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 (VPC)**, and **CloudFront**.

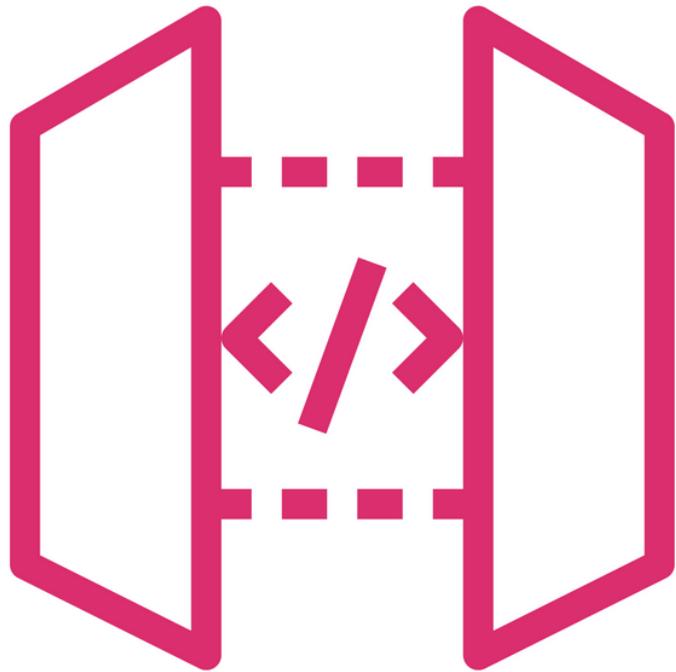
**API Gateway** is a fully managed service for creating, deploying, and managing APIs. It allows you to create RESTful and WebSocket APIs. It also comes with advanced features such as authentication, caching, and monitoring, which don't require you to write much or any code.

**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.

**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, creation of subnets, and configuration of route tables and network gateways. You can 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

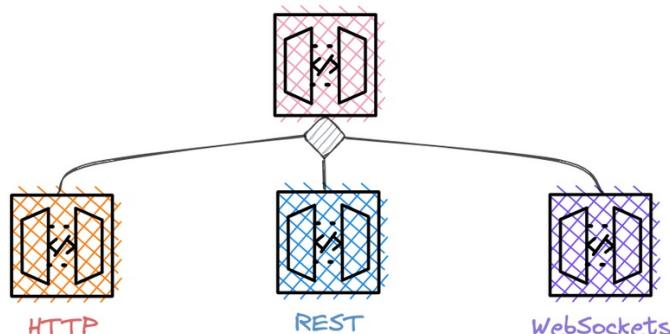
AWS API Gateway is a crucial component of Serverless applications on AWS. It acts as a fully-managed service that serves as a front door for your application's ecosystem to the internet.

Unlike other methods that allow internet exposure, such as Lambda function URLs, API Gateway not only offers a simple way to trigger invocations via external sources, but also provides access control, data transformations, rate-limiting, and much more. It is not limited to just Lambda functions, but can also be used with many other AWS services or any backend that speaks HTTP.

API Gateway is beginner-friendly, as it is easy to deploy your first API and there are no upfront or idling costs to worry about.

## 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 to other AWS services like Lambda. It is the most common and flexible gateway type and helps you perform many otherwise tedious tasks, such as request validation or data transformations, without writing any code.
- **HTTP** - comparable to REST but comes with a reduced set of features. It is 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 appropriate gateway choice depends on your specific requirements.

### **Rest Gateway for Full-Fledged 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, which reduces your operational costs. It is also about reducing the need for a lot of boilerplate implementation code that can be error-prone and conflicting in many ways.

And that is precisely what all AWS API Gateway types do very well. They come with many no or low-code but high-value features that do not require deep knowledge or long experience.

### **HTTP Gateway for Simple, Cheap, and Secure Application Exposure**

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 that allow you to map HTTP resources and methods to your AWS services.

One of the most famous features is the Default JWT Authorizer. It enables you to secure your API in a simple way with an identity provider that supports OAuth2 and OpenID Connect. You don't need to write custom code because the HTTP Gateway takes care of authentication and authorizing requests based on your configuration.

### **WebSockets for Real-Time Communication between Clients and Servers**

In web applications, we mostly think about clients like browsers or mobile apps that make HTTP calls to exposed services and then wait for the responses.

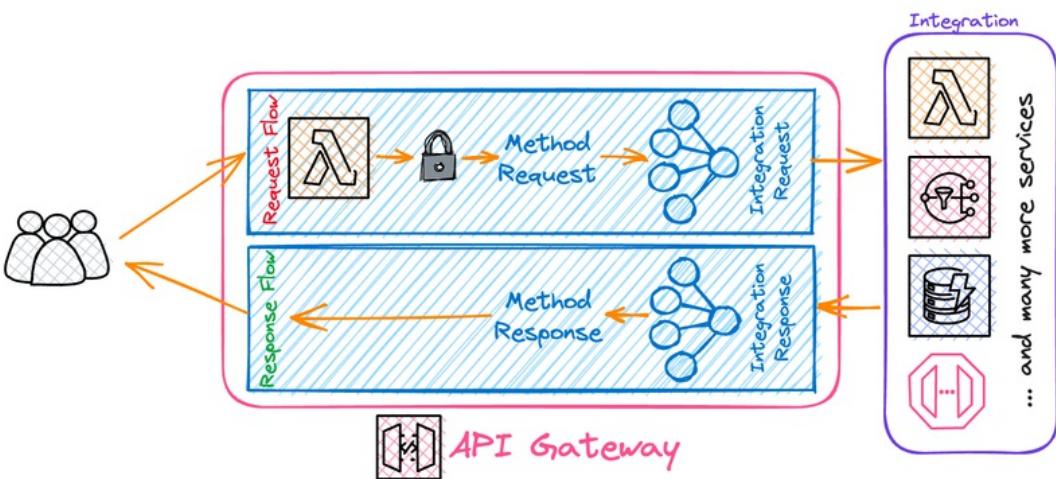
Depending on the connections between the client and the server, this includes low or high latencies as each request involves a complete roundtrip between the client and server. Also, connections are not kept open steadily, so initial requests can take longer due to TLS handshakes that are rather compute-intensive. If clients are actively waiting for data to update, they need to poll regularly and cause load on your servers, even if the data hasn't updated yet.

But that is 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 in both directions. 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 you are looking at REST or HTTP gateways, an API is a collection of programmable **resources**. These resources can be mapped to one or several HTTP methods and to a path that is either very specific or contains a proxy wildcard so it covers a set of paths.

Each of these 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 is returned to the client.

A summary of all these 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

secure multi-tenant usage of your application.

- **Integrations** - integrating your API method with a backend.
- **Request Validation** - validating your request before sending it to your destination and saving the boilerplate code at your backends.
- **Data Transformations** - applying data mapping templates to automatically convert request data to expected inputs.
- **Gateway Responses** - applying mappings to convert outputs to expected responses.
- **CORS** - setting up cross-domain rules to allow integration of your API endpoints in different web touchpoints.
- **Deploying APIs** - deploying your APIs to the internet.
- **Caching** - caching requests to lower latencies.
- **Monitoring** - tracking the requests to your API endpoints.

### **API Endpoints Contain Routes and Methods and Lead to Integrations**

After selecting the desired API Gateway type, creating a new endpoint is quick and easy. For example, when using 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, such as Lambda functions.
2. Configuring our routes - map HTTP methods and paths to our integrations.
3. Defining stages - we can run multiple stages in parallel and choose when to deploy which stages. The 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:functor
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. Versions 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 by going to your Lambda function and choosing 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.

The screenshot shows the 'Define stages' step of a four-step API creation wizard. On the left, a vertical navigation bar lists 'Step 1 Create an API', 'Step 2 Configure routes', 'Step 3 Define stages' (which is selected and bolded), and 'Step 4 Review and create'. The main content area is titled 'Configure stages' with an 'Info' link. It contains a descriptive text about stages and their deployment behavior. Below this is a table with one row for the '\$default' stage. The table has columns for 'Stage name' (containing '\$default'), 'Auto-deploy' (with a checked toggle switch), and 'Remove' (a button). A 'Cancel' button is at the bottom left, and 'Previous' and 'Next' buttons are at the bottom right.

Finally, we define the stage. In our case, there is only one stage with auto-deploy enabled. This means we don't need to worry about deployments, as changes will be automatically propagated.

After going through the `Review and create` process, your new API endpoint will be created and 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 actions they can perform. For example, you could grant read-only permission to all resources within the IP range of `192.0.2.0/24`. Additionally, it is only possible to use the HTTP `GET` or `OPTIONS` operations.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
```

```

    "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 or OPTIONS request will be denied, regardless of the IP address of the client.

### **IAM Policies to Grant Permissions for Invoking Other AWS Services**

API Gateways are often a central point used to connect and protect different parts of larger application ecosystems. Managing or creating API gateways requires IAM permissions, which are 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 Easy OAuth2 Integration

OAuth2 and OpenID Connect are two of the most commonly 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 composed of three parts: the header, the payload, and the signature, separated by dots.

- **Header** - contains information needed to validate the signature of the token, such as the algorithm (e.g. HS256) and the identifier of the key used to generate the signature.
- **Payload** - contains information about the token itself, such as **sub** for subject (the identity for which the token was issued) or **iat** for "issued at" (the Unix timestamp when the token was generated), as well as any custom information you want to transfer with the token.
- **Signature** - the signature that can be verified via the information from the header to ensure that the token is valid and was generated by an authorized authority.

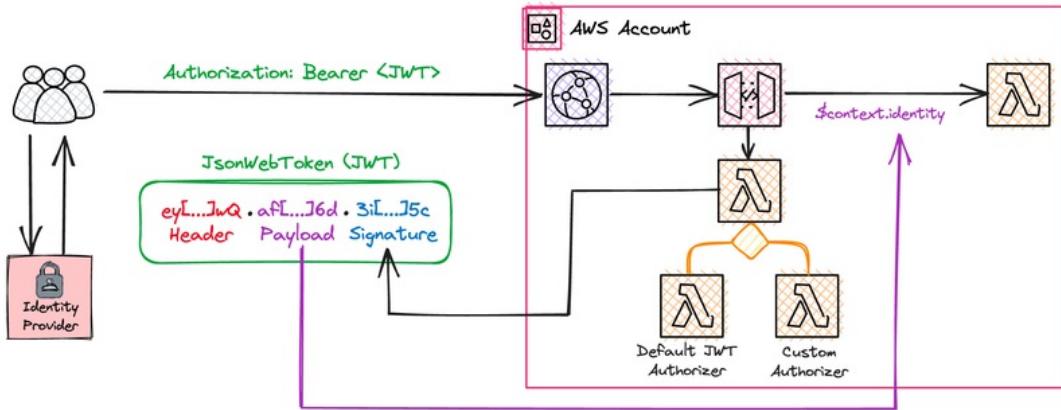
Your OpenID-supporting identity provider exposes you to the OpenID configuration endpoint, which contains all the necessary information to validate incoming JWTs.

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 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 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 pass both checks. Every other request will be declined with HTTP 401 and will not be 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 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 you haven't written any lines of custom code yet.

### Lambda Authorizers for Limitless Custom Authentication and Authorization

If you are using the REST gateway type, you cannot use the default JWT authorizer. In this case, you must rely on the Lambda Authorizer, which is a custom Lambda function that you must 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 is not subject to any restrictions. As long as you have assigned

additional permissions, you can invoke other AWS services or API calls. If you are using OAuth2, you can rely on standard open-source libraries like `jsonwebtoken` to validate the signature of incoming tokens and forward any authentication context you need.

### Mutual Authentication via Certificates and mTLS

API Gateway also supports mutual TLS, which enforces authentication and authorization via certificate ownership. 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 charge: Usage Plans. Usage plans allow you to restrict the number of API calls to your gateways, not on a global level, but per client. Each client will receive their own API key, which they must send with each request, and the usage plan will keep track of the number of invocations. If the number of invocations is 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 **won't be 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 targets you want to 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 forward the request to any backend. This is helpful for testing, 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 results 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 ●

**Request Validator** Validate body, query string parameters, and headers Edit ●

**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 validation is that you don't have to write additional code, use libraries, or consume compute resources from your backend services. AWS API Gateway will take care of the validation without additional costs and automatically reject invalid requests that won't trigger an integration request.

## Transforming Your Data to Meet Integration Expectations

In addition to validations, API Gateway can also 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 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 customer identifier 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 `$input`, you can access the request information, which includes the payload, path, and query parameters. With `$context`, you can access the authorization context provided by the Authorization Lambda.

Transformation templates can be difficult to build, read, and maintain, but they do not require writing any code, result in additional billing, or rely on your computing resources.

Data transformations can also be applied to responses, meaning that you can transform responses from your integrations to the expected format of your clients. It is also possible to assign different transformations based on the HTTP status code.

## Handling Invocation and Validation Errors

With gateway responses, you are 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 successful invocations.

Let's try this by focusing on Lambda execution errors.

First, we need to create a method response with an HTTP status. We are choosing HTTP 500 as we want to map all the invocation errors for Lambda. At this point, it is also possible to define a model that explicitly defines how the response should look. In comparison to the model for the request, it is not used for validation but only for informing 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. To accomplish this, we can use a regular expression that searches for all status codes starting with 5, which includes HTTP 500 (invocation error) and HTTP 503 (timeout), for example.

## [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

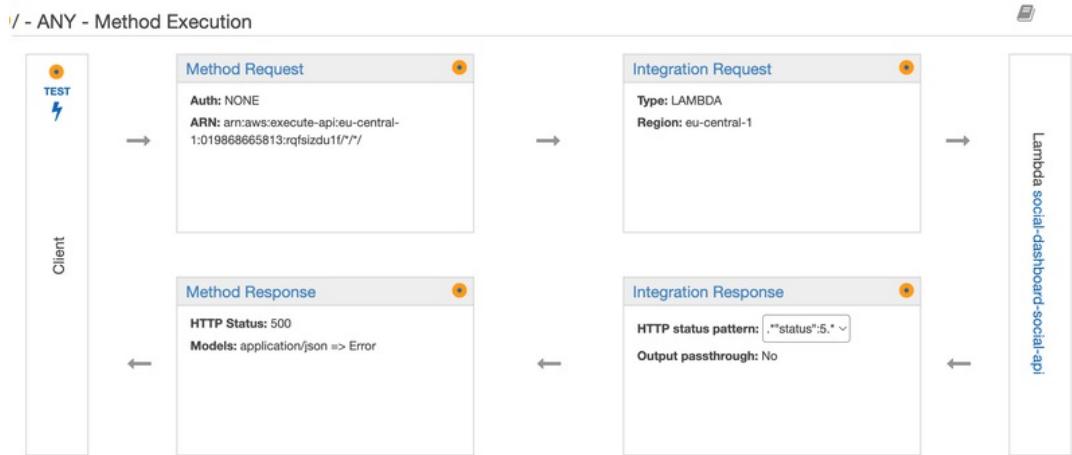
information we want to pass in the final gateway response.

To achieve this, we need to understand what invocation errors and timeouts look like in 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. Now, Lambda execution errors such as timeouts will be converted into responses that only contain the error message.

### Dealing with Cross-Origin Requests or "Not Fearing CORS"

If you have experience in web development, you may have encountered cross-origin issues with AJAX requests. Perhaps you've copied a StackOverflow response into some part of your code to resolve the errors.

Let's briefly discuss what CORS is and how to properly configure it in your API gateways.

### Cross-Origin Resource Sharing 1xi

CORS is not a bad thing; it's actually a good thing. It allows requests to be made to domains other than the one currently being browsed. The server can't enforce rules, but it can supply information to clients. If the client is a browser, it will take care of applying those rules.

To simplify, you can imagine this as a conversation 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 and 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 and 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 just as easy with a REST Gateway. After clicking on the corresponding field in the navigation, you can configure your rules. The interface is not as polished as with an HTTP Gateway, so you need to separate multiple headers and origins in the same input field using 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. If the requests match the expectations for your rules, mock integrations will be assigned to them.

## 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 make in the configuration. With the REST gateway, each of your stages needs to be deployed manually.

Think of a stage as 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 can 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, for example, 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. However, this feature 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 stores responses for a defined period of time. Consecutive requests that map down to the same key, such as 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 retrieve 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 retrieves orders of an authorized user:

```
GET /orders?status=...
Authorization: Bearer eyXf...
```

The results of the request will always depend on the user who took the request and the status of the order. Two requests from the same user that request 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 include at least the query parameter `status` and the `Authorization` header as a cache key.

### **Monitoring APIs to Immediately Become 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 is recommended to immediately set a retention period for your log to expire automatically after a given period of time.

### **Building Real-Time Communication Applications with WebSockets**

The WebSockets API Gateway is a 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 do not 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, for example, 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 us jump into the fun and create our first WebSockets gateway.

Compared to other examples in this book, this part may not be trivial and may involve a lot of steps. Do not feel bad if something does not work out in the beginning!

### Creating A WebSocket API Gateway

Go to the API Gateway Management Console and click on `Create API`. If you do not have any gateway in your current region, you will 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`.

**Specify API details**

**API name**

**API name**  
A unique ID will also be generated, and it can be used to programmatically refer to this API.  
`awsfundamentals-websockets`  
The name is cosmetic and does not have to be unique.

**Route selection expression [Info](#)**

**Route selection expression**  
A route selection expression tells API Gateway which route to call when a client sends a message.  
`$ request.body.action`

**Cancel** **Review and create** **Next**

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 connection events with a single Lambda function. Therefore, 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));
};
```

Afterwards, jump back to our API Gateway wizard and choose 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 AWS 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 you click **Next**, you will be taken to a final overview of the settings that will be applied to our new gateway. Confirm them to create our first WebSocket gateway.

Now we have a working WebSocket gateway that accepts new connections, keeps them alive,

and forwards every connection to our new Lambda function.

Let's click on the `Stages` tab and select our one and only stage, `production`, to get the URL to connect to our API gateway. It is prefixed with `wss`.



The screenshot shows the AWS Lambda 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, and under it, there is a single stage named "production". On the right side of the screen, there is a "Stage Editor" panel titled "production Stage Editor". Inside this panel, the "WebSocket URL" is listed as "wss://qu91138z2g.execute-api.eu-west-1.amazonaws.com/production" and the "Connection URL" is listed as "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` file in your favorite editor and add a simple script that will only establish 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 by running `node client.js`. You should see our success message, indicating that it has connected to the gateway. We have added a blocking promise to prevent the script from finishing execution immediately.

To check the outputs of our Lambda function, go to its corresponding CloudWatch log group and the latest log stream. You will find the event that we received from the gateway.

```
{  
  "headers": {...},  
  "multiValueHeaders": {...},  
  "requestContext": {  
    "routeKey": "$connect",  
    "eventType": "CONNECT",  
    "connectionId": "dmwL5duMDoECJPQ=",  
    ...  
  }  
}
```

I've removed everything that's not important to us at the moment. What we're interested in are the `routeKey` and the `connectionId`. The connection identifier is a unique identifier for this connection and is necessary to send messages to this specific client.

Let's stop the script and check which event we receive if our self-built client disconnects.

```
{  
  "headers": {...},  
  "multiValueHeaders": {...},  
  "requestContext": {  
    "routeKey": "$disconnect",  
    "disconnectStatusCode": 1001,  
    "eventType": "DISCONNECT",  
    "disconnectReason": "Going away",  
    "connectionId": "dmwL5duMDoECJPQ=",  
  }  
}
```

We can see our expected connection identifier and the route key `$disconnect`. Additionally, we can see the status code and reason for the disconnect.

### Using DynamoDB to Store Connection Information

We are now able to receive events about opening and closing connections. However, how do we work with open connections? First, we need to store the connection identifiers.

DynamoDB, being 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 that we can access all of the table's currently saved connections and later broadcast messages to all connected clients.

Let's modify the `index.js` file of your Lambda function that handles our connections. We can 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.

Afterward, open a new browser tab and navigate to our new DynamoDB table. By clicking on `Explore table items` on the right, we can enter 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 take another look at our table. We should see a new item!

Completed. Read capacity units consumed: 0.5	
Items returned (1)	
<input type="checkbox"/>	connectionId
<input type="checkbox"/>	dm9f-cDCjoECJEg=

If we close our client by interrupting our script (e.g. by pressing `control+c` on macOS), we will immediately see that the item is gone again after running another scan.

That is exactly what we wanted. Our table now always contains all connections that are currently active.

### Sending and Receiving Messages Between Clients

The fundamental operation is done now. Let's move on to the interesting part: 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 and receive messages.
2. We need to extend our Lambda 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}`);
  };
  webSocket.onmessage = (event) => {
    console.log(`Received message: ${event.data}`);
  };
  webSocket.onclose = () => {
    console.log(`Disconnected`);
  };
};

run();
```

```

        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();

```

To start our client with a specified name, we can enter the command `node client.js first-client` in the terminal. To start additional clients with different names, we can open new terminal tabs.

If we log the received event in our Lambda function, the output would resemble 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 work on 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 are currently broadcasting a received message to all clients, except for the client that sent the message.

To post messages to the gateway, we need to grant our Lambda function another permission. This permission is the `execute-api:ManageConnections` action. Navigate to your function's execution role and modify the policy to include this required permission. For simplicity, you can allow 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 any 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 two 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 output 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 receive messages from all others, except themselves.

If you've made it this far, great job! You've successfully assembled many small parts that now work together and can be used to build amazing applications!

### **Expected Costs for Different AWS API Gateway Types**

AWS API Gateway operates on a pay-per-use model, which makes it user-friendly for beginners because they don't have to worry about hourly charges for unused gateways.

If we exclude caching, the pricing is solely based on per-request charges. Looking at current rates, the following rates will be important to remember in 2023:

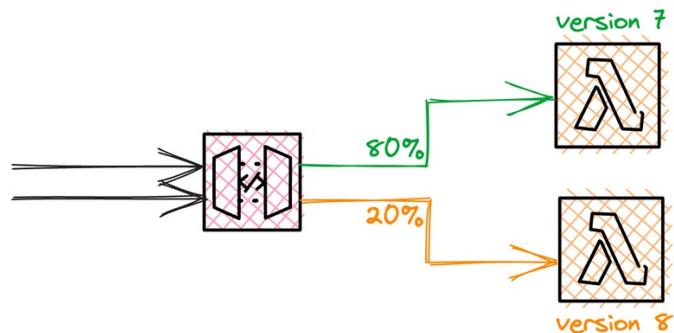
- **REST:** \$3.5 per 1 million requests.
- **HTTP Gateway:** \$1 per 1 million requests.
- **WebSockets:** \$1 per 1 million messages & \$0.25 per 1 million connection minutes.

The above examples assume the `us-east-1` region, and prices can vary slightly per region.

What stands out is that if the features of HTTP gateway meet your needs, you should always stick to HTTP gateway, because it is more than 71% cheaper than REST.

### **Deploying Safely with Canary Deployments - Even with Lambda**

Canary deployments with API Gateway involve gradually routing a portion of traffic to a new version of an API while still maintaining the majority of traffic on the existing version. This allows for testing and validation of the new version in a real-world environment before fully migrating. API Gateway provides canary deployment functionality through its deployment stages and stage variables. By using stage variables and traffic splitting rules, you can control the percentage of traffic sent to each version of the API.



When using Lambda as an integration with API Gateway for canary deployments, you can utilize aliases to achieve the desired behavior. As we've learned before, aliases are essentially pointers to different versions of your Lambda function.

To perform canary deployments with Lambda integration, you would typically follow these steps:

1. Create a new version of your Lambda function for the updated code.
2. Create an alias for the new version, specifying the desired percentage of traffic to be routed to it.
3. Configure API Gateway to use the alias as the integration target for your canary deployment stage.
4. Define traffic splitting rules in API Gateway to distribute the traffic between the existing version and the alias based on the specified percentages.
5. Gradually increase the traffic percentage to the alias to test and validate the new version.  
In our example above, we'll start by directing 20% of the live traffic to the new deployment.
6. Monitor the canary deployment and make any necessary adjustments.
7. Once you're satisfied with the new version, you can update the alias to point to the new version or promote it to be the default version.

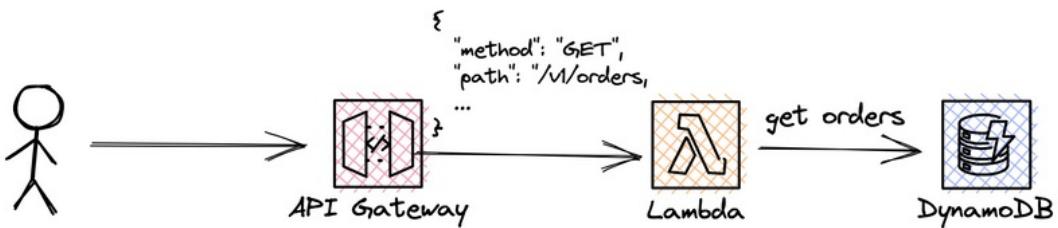
Using aliases allows you to easily switch between different versions of your Lambda function without modifying the API Gateway configuration.

## Use Cases for AWS API Gateway

We've learned that AWS API Gateway is not just a simple front door for applications. It's an enabler for other great features and plays a key role in many applications and ecosystems. Let's explore some 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 that could be a webpage or mobile app, which communicates via HTTP with a backend. On the server side, your backend is a construct of one or multiple Lambda functions that include your business logic.



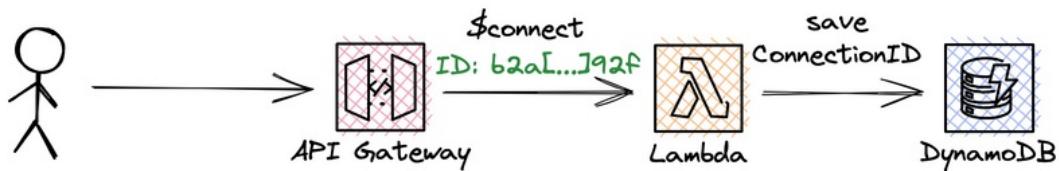
Exposing these functions to the internet is done through a REST or HTTP API Gateway. This allows you to actively enforce rate-limiting, request validation and transformation, routing, authentication, and authorization without writing much or any code at all.

### Use Case 2: Real-Time Communication Between Clients and Backends

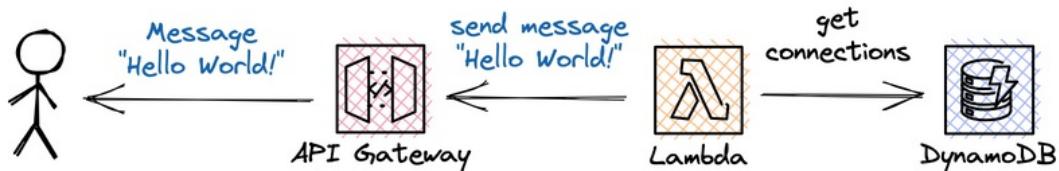
This was our hands-on example, but let's quickly revisit it.

Many applications require real-time communication between clients and backends. 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 high latency between a message that has been sent and the actual delivery at the receiver.

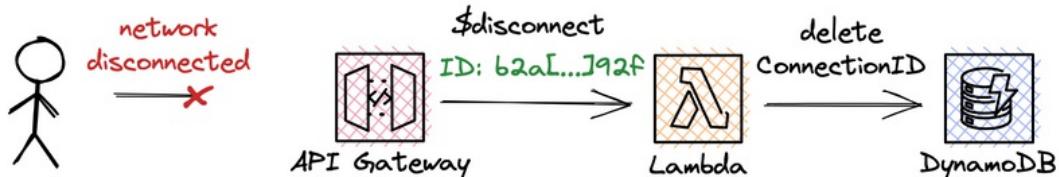
With WebSockets API Gateway, a client can open a connection to the gateway that is fully managed by the gateway itself. The only thing that needs to be actively implemented is how connection events are handled and what you want to do with those connections. A common pattern is to save the connection identifiers that are provided by the gateway to DynamoDB.



Using a connection identifier, you can actively push messages to the corresponding client.



When the client disconnects, either intentionally or due to a network timeout, API Gateway will send a disconnect event to your function so that you can delete the connection identifier from DynamoDB.



This pattern can be used for many different applications and doesn't require any infrastructure management on our side.

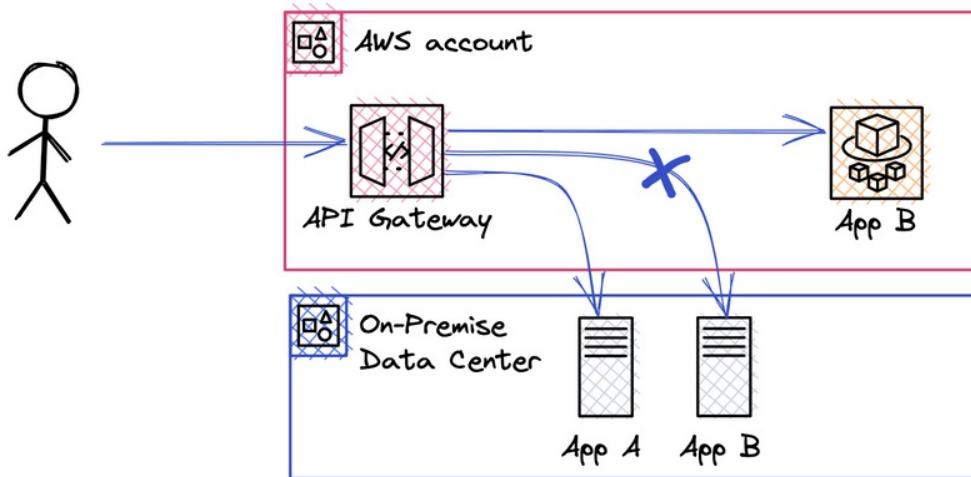
### Use Case 3: Using the Lift and Shift Approach to Migrate Applications to the Cloud

Many open positions for developers are created for projects that need to be migrated to the cloud. This is not a trivial task for applications that are already in use by customers and can't experience 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 building your application infrastructure in the cloud, for example, 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 have finished migrating a part of your application, you can start routing requests (for example, 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 is common to allow for third-party integrations by other developers. They can then start using your APIs to build new features or even whole applications on top of your existing infrastructure and data. This can help drive your product further and increase its range via new features you didn't even think of.

It is 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 many features, and it's not easy to remember all of them. Let's quickly go through some small, big, and lesser-known 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 straightforward when using

HTTP Gateway and selecting the JWT Default Authorizer. However, API Gateway is not limited to that: a custom Lambda Authorizer can be created to enable any authentication and authorization mechanism. Additionally, API Gateway supports mutual TLS.

- It's easy to attach a custom domain name to your gateway if you have a certificate 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 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 avoids many bugs as it reduces the lines of code you need to write and maintain.
- Usage Plans are 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

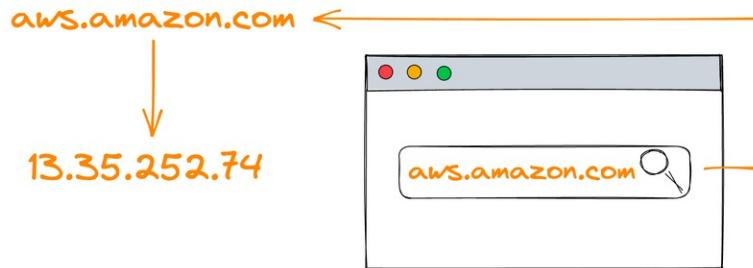
## Introduction

Route 53 is a managed service introduced by Amazon in 2010. It reliably redirects traffic via domain names to your applications, whether 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 someone, 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 high scalability and avoids a single point of failure. Furthermore, it has a hierarchical structure that is maintained by registries in different countries.

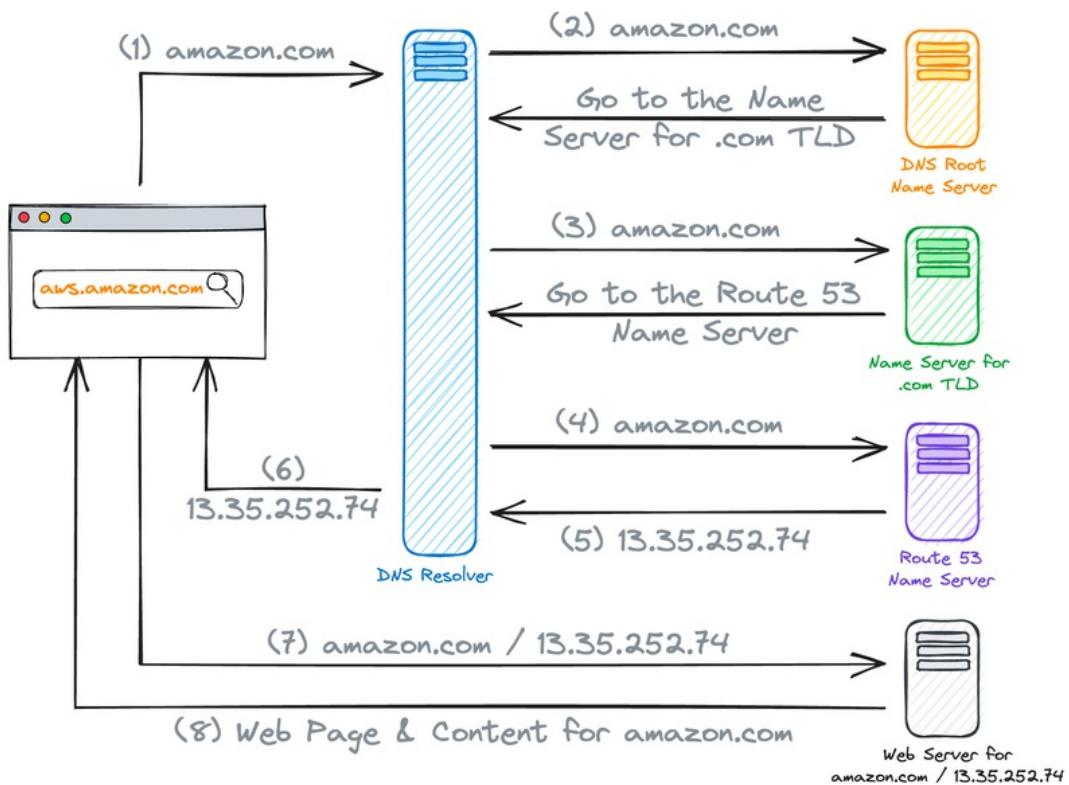
On top of the hierarchy are the authoritative name servers which hold the actual addresses for the DNS records.

When we descend, we also encounter top-level domain servers, root name servers, and recursors.

## Multiple Services on Different Levels Participate 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 has the final authority over a domain. These services provide IP address information to clients and 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 upstream or downstream systems that are part of a request.

There are different locations where caching takes place. The two most obvious caching locations are **your browser** and **your operating system**. The closer the caching is to your

browser, the better, as fewer processing steps are necessary. The Time-To-Live (TTL) attribute specifies how long a DNS record is stored.

## 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.

In 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 receives 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 obtain 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. **Obtain your current DNS configuration** - as described above, either manually or via the zone file.
2. **Create a hosted zone in Route 53** - The hosted zone needs to have the name of your domain and will later contain all of your records. AWS will automatically create a name server record (NS) and a start of authority (SOA). You will 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.

Route 53 > Hosted zones > Create hosted zone

## 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.  
  
Valid characters: a-z, 0-9, ! " # \$ % & ' ( ) \* + , - / ; < = > ? @ [ \ ] ^ \_ ` { | } . ~

**Description - optional** Info  
This value lets you distinguish hosted zones that have the same name.  
  
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. Otherwise, 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 process of having all your domain management in a single place, but it is not necessary.

#### Migrating to Route 53 for an Inactive Domain

To migrate a domain that is not receiving any live traffic, fewer steps are required. You only need to follow steps 1, 2, 3, and 6. You don't need to adjust 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 buy a domain directly through Amazon and Route 53's console interface.

The screenshot shows the AWS Route 53 Dashboard. On the left, there is a navigation sidebar with the following menu items:

- Dashboard
- 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 contains several sections:

- DNS management**: A hosted zone tells Route 53 how to respond to DNS queries for a domain such as example.com. It includes a "Create hosted zone" button.
- Traffic management**: A visual tool that lets you easily create policies for multiple endpoints in complex configurations. It includes a "Create policy" button.
- Availability monitoring**: Health checks monitor your applications and web resources, and direct DNS queries to healthy resources. It 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. It includes a "Register domain" button.
- Readiness check**: Shows 0 readiness checks.
- Routing control**: Shows 0 control panels.
- Register domain**: A section for finding and registering a domain. It has a text input field containing "awsfundamentals.com", a note about character limits, and a "Check" button.

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 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 policies that you can use for your domain names. Let's dive into those policies in the next paragraph.

### **Choosing the Right Routing Policy**

The record type determines how Amazon Route 53 responds to queries for those domain names. The selection of the best-fitting record type highly depends on your requirements.

#### **Simple Routing to Forward Requests to One or Multiple Resources**

This is the standard DNS record without any strings attached. It is typically used for single resources that perform specific functions for your domain.

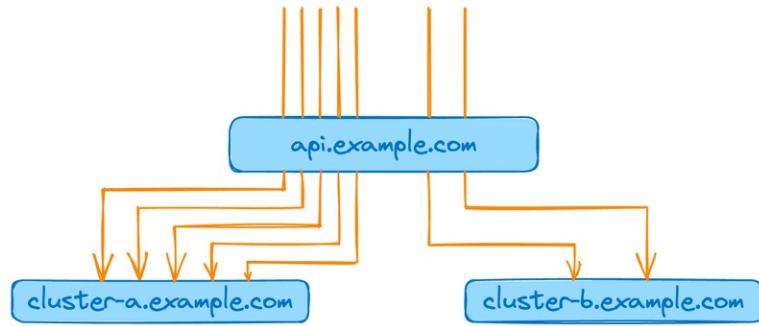
You cannot create multiple records with the same name for this record type. However, you can specify multiple values for your single record.

Route53 will return all values for an incoming query to the client, who can then choose one of those.

#### **Weighted Routing to Allocate Traffic Proportionally between Multiple Targets**

As the name suggests, it allows you to define multiple records for the same domain name. You can choose how much traffic is routed to each record by giving it a percentage.

Prominent use cases include 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.

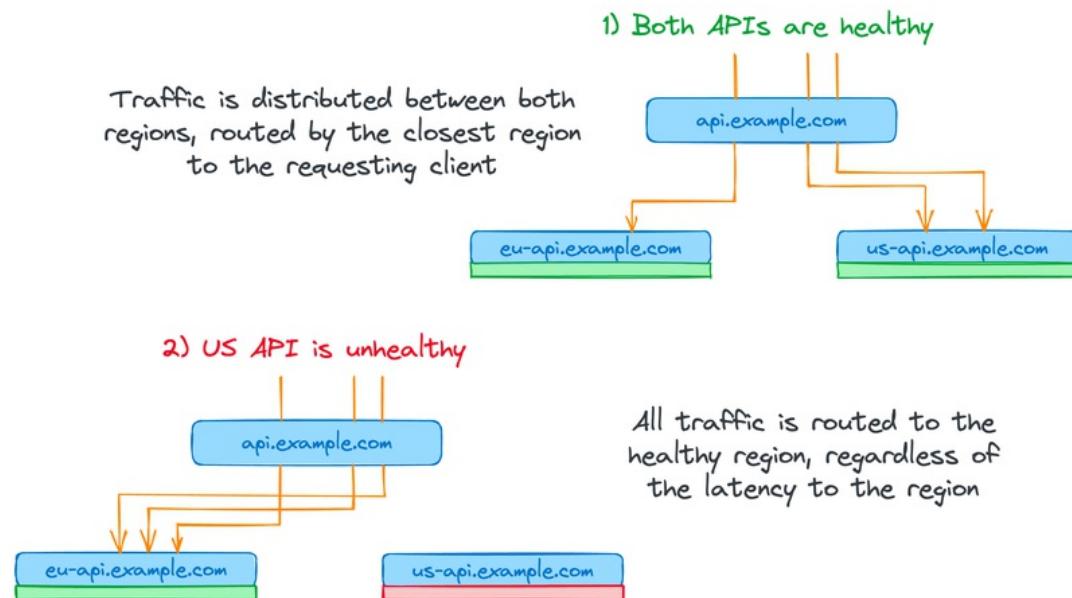
#### **Failover Routing to Send Traffic to a Fallback Resource If the Main Target Is Unhealthy**

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 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 relatively easy active-active multi-region failover setup.

#### **Latency-Based Routing for Forwarding Requests to a Resource That Is Closest to the Customer**

In a multi-region setup, you usually want to route requests to the closest regions as it will serve the fastest responses on average.

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.

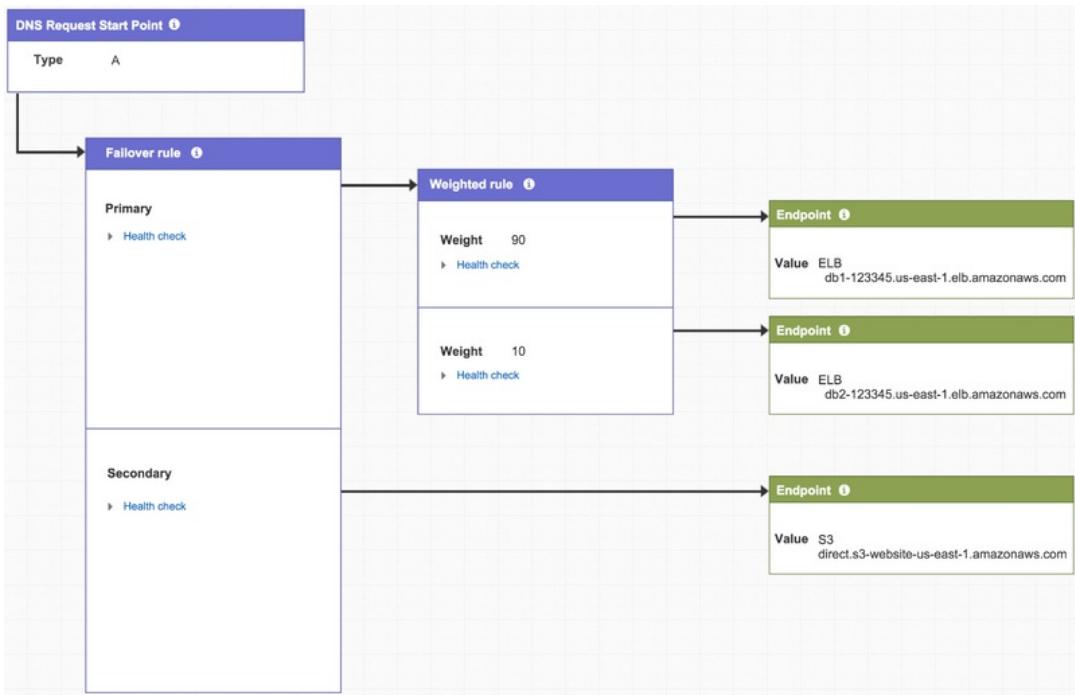
The granularity of locations is either by continent, country, or even US state.

#### **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 can operate even if drastic incidents in one or multiple AWS availability zones or regions occur.

#### **Building Sophisticated Routing Configurations in a Visual Editor**

The traffic flow visual editor allows you to create advanced routing configurations for your resources. It supports various routing types like failover and geolocation. You can save these configurations as traffic policies and create multiple policy records to route DNS queries for specific domains or subdomains.



Additionally, you can create multiple versions of a traffic policy and use them to easily roll out or roll back configuration changes.

Configurations with Route 53 can quickly become complicated. Fortunately, this is an immense helper for building and visualizing complex setups.

### 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. It is not just a simple domain name system, but an advanced tool that enables you to build resilient and highly available architectures.

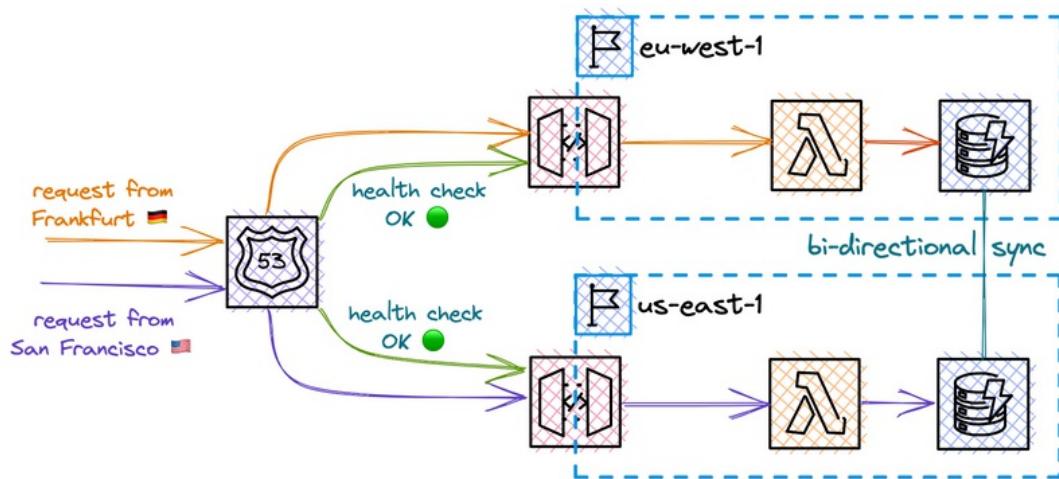
### Use Case 1: A Multi-Region Setup for Serverless Architectures

If you are focusing on serverless architectures, you can deploy your ecosystem around the world without drastically increasing your costs. This is due to the fact that unused resource capacities do not contribute to your bill. So, deploying your Lambda-powered application in more regions will not have much or any effect on your charges at the end of the month.

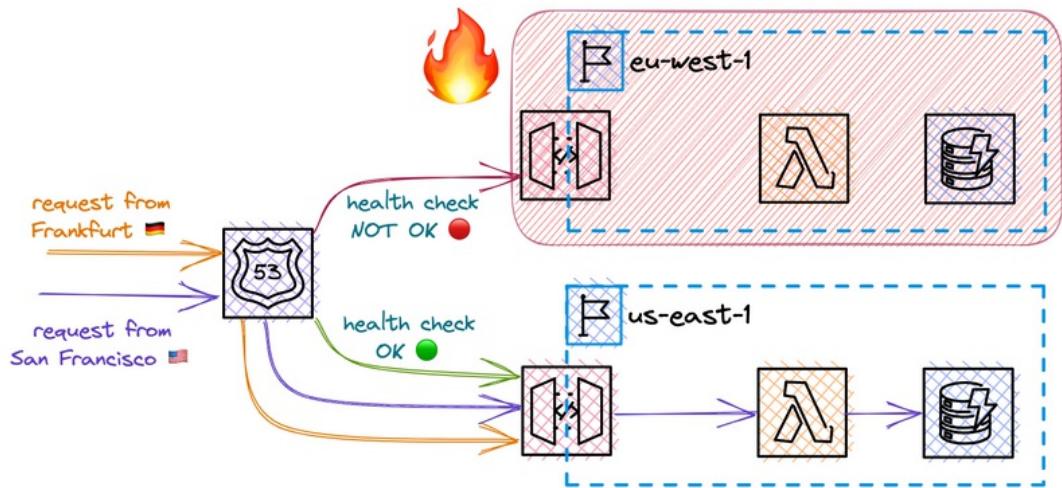
Building such a multi-region setup with Route 53 is straightforward and involves only a few steps:

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 is little or no traffic at all in a single region, it will not hurt in any way.



If a region experiences an outage, or if you accidentally break the application due to a faulty deployment, there won't be any significant downtime because Route 53 will quickly failover and won't return the unresponsive region in any DNS query.



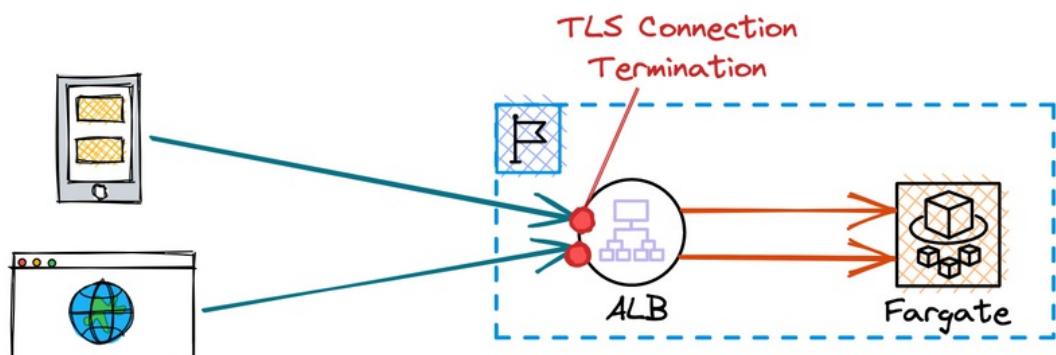
By doing so, requests for origins with long distances may become slower. However, at least the availability of the application is only affected for a very short period of time.

#### 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 base first to mitigate the risks of deploying an erroneous version to a large 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.

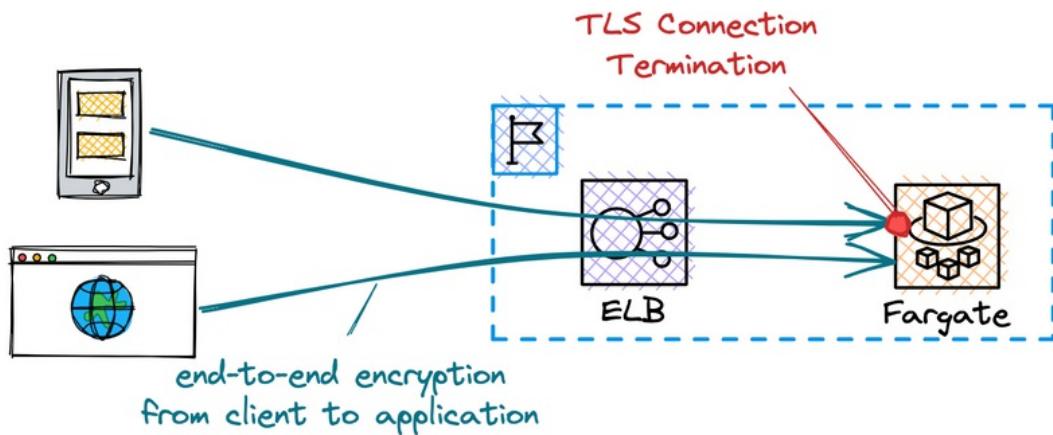
Let's do a quick recap of some networking fundamentals before we continue with our scenario.

Services like the Application Load Balancer (ALB) offer request routing based on request details out of the box. They 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.

However, some architectures do not support such simple traffic routing. This includes applications that require secure connections to the application container. For example, if you are 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, you need to use 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 security even further, as the connection has real end-to-end encryption from the client to the application itself.

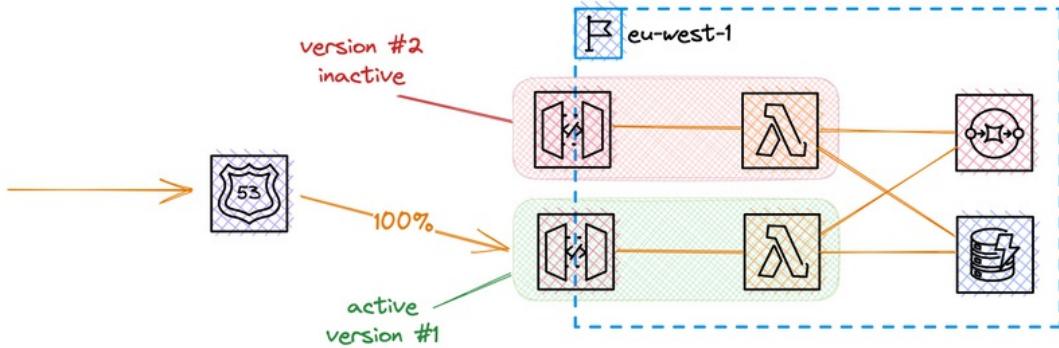


The obvious downside is that all services in between, including the ELB, cannot inspect the request details for HTTPS requests. Therefore, it is not possible to make any decisions based on the request for routing. However, we can set weights on target groups to route a percentage of traffic to each group.

Why does this matter? In Serverless architectures, we do not work with load balancers as there are no long-running containers. Instead, we work with AWS API Gateway, which 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.

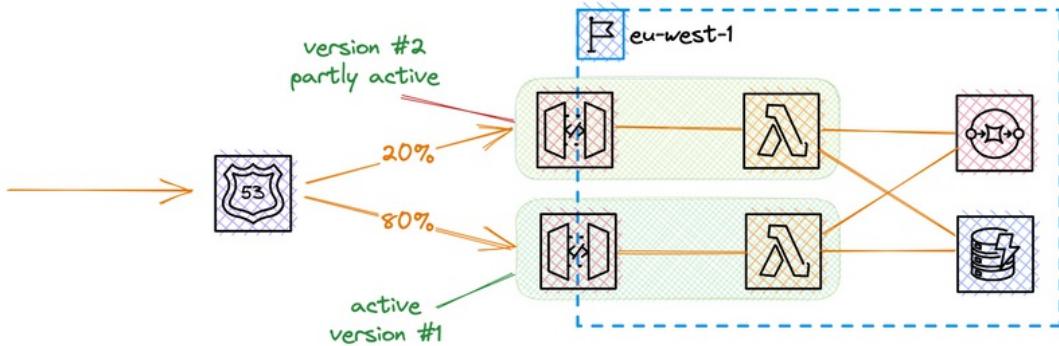
Often, our complete infrastructure deployed is coupled with our Lambda function's code, so setting up routing to different versions of functions is not easy. It is also not possible to roll back in cases of errors if the infrastructure is not strictly separated from our function's code.

But let's think back to the main advantage of Serverless architectures: we can easily replicate our whole infrastructure without worrying about costs. That's why we can run multiple stacks (i.e., 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.

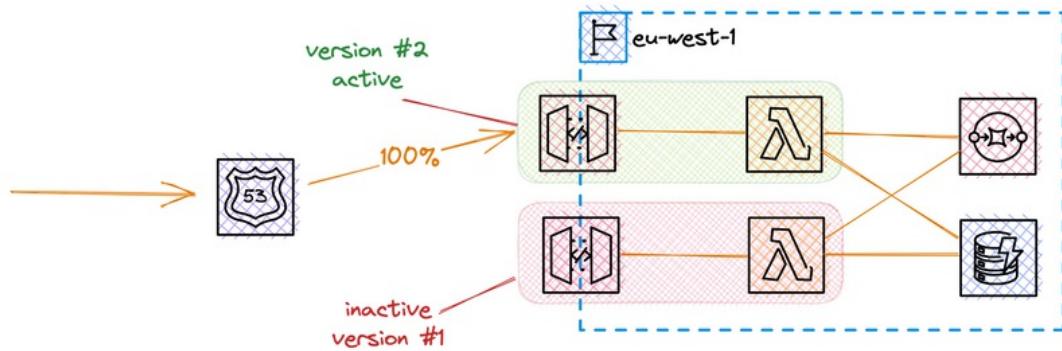


We will deploy multiple stacks of our application infrastructure. In this example, it contains an AWS API Gateway deployment and an AWS Lambda function. We have other global infrastructure that is independent of our application stack, such as SQS and DynamoDB.

Each of our regional API gateways will receive its weighted DNS record, with the inactive stack having a weight set to zero and the active one set to 100.



When we want to roll out a new code deployment, we deploy the inactive stack first. After it has successfully completed, we can adjust the weights in our DNS record. For example, we can set 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 can delegate the permissions to administer your domain to Route 53 from any DNS provider, including famous platforms like 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 won'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) allows you to create and deploy AWS resources in a logically isolated virtual network. It can simulate your local data center and provide 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 accessible to the world. This is not necessarily a downside, but it is mostly what you want. You want customers to be able to access what you have built.

However, this comes with the risk of attacks from the internet.

VPC helps you to **secure your resources and protect them in different ways** from the outside via multiple methods. Additionally, not all of your resources have to be publicly available or have access to the internet at all. This includes databases or caches, for example. VPC also enables you to protect those resources by restricting their network access and availability.

### Getting into Networking Fundamentals to Understand How VPCs Work

Before we delve into the capabilities of VPCs, it is important 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 further segment a VPC. This enables you to split resources into different segments, such as grouping resources that require or do not require 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 that contain a set of rules, called routes, which 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.

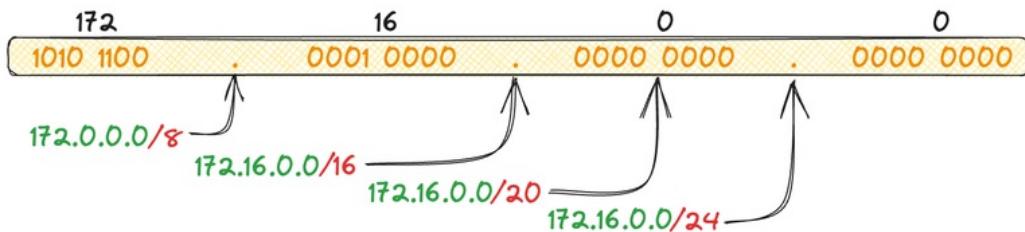
### Classless Inter-Domain Routing Blocks (CIDR)

Your VPC requires a range of IPv4 addresses that attached network interfaces can use. These ranges are defined as Classless Inter-Domain Routing (CIDR) blocks.

They consist of two sets of numbers:

- **The prefix** - the binary representation of the address.
- **The suffix** - the total number of bits in the entire address.

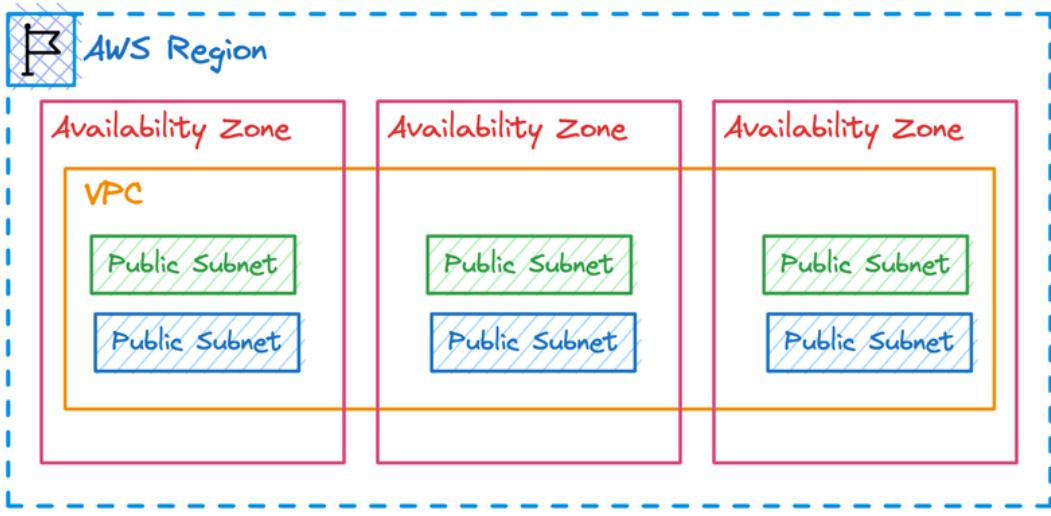
The allowed block size for a VPC is between **65,536** (netmask **/16**) and **16** IPs (netmask **/24**) addresses.



## Virtual Private Clouds

Each VPC is created for a region and always spans across all availability zones. Each availability zone can contain subnets, which are another breakdown of your VPC.

Subnets cannot span multiple availability zones but only a single one. For redundancy and availability reasons, it is therefore recommended to have at least two subnets for a single region so that you can have resources in at least two availability zones.



#### **Each Region Comes with a Default VPC and a Default Set of Subnets**

Every AWS account created after the end of 2013 comes with a default VPC per region. Each of these 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 divide 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, such as their own Security Groups or Network Access Control Lists.

Some AWS services require you to launch instances into a private subnet, such as ElastiCache.

#### **Adding Layers of Security with Network Access Control Lists and Security Groups**

Depending on the type of resource and your requirements, it is necessary to further restrict access to and between your resources.

Common requirements include:

- a resource should or should not have access to the internet
- a resource should be accessible by other resources in a different subnet
- a resource should only have access to a specific IP range

You can achieve this using 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.

Inbound Rules ↓		
Source	Protocol	Port Range
0.0.0.0/0	TCP	80
::/0	TCP	80
0.0.0.0/0	TCP	22

Outbound Rules ↑		
Source	Protocol	Port Range
0.0.0.0/0	TCP	1433
::/0	TCP	1433
0.0.0.0/0	TCP	3306

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 discussed 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 residing 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 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 to 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 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 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 that 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 that controls the traffic for subnets without a custom route table attached.

### **Monitoring the Traffic in Your Network**

If some requests are not reaching your instance in your VPC, you need to gain 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 through Flow Logs, which 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
```

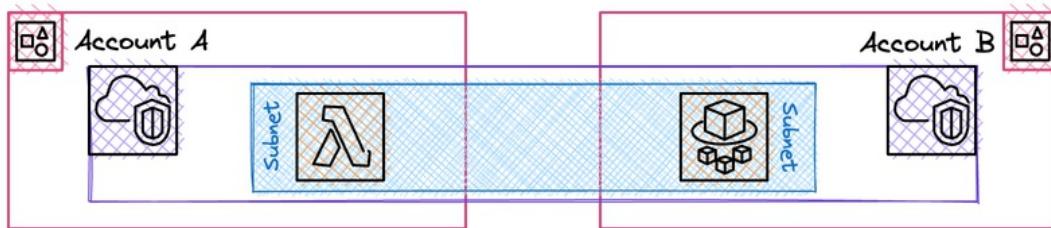
By analyzing the example flow logs provided above, we can see that an incoming request was accepted, but the response was rejected. This can occur even if you have defined allow rules for inbound traffic in your security group and network ACLs. Since security groups are stateful, responses are allowed. However, as ACLs are not stateful, a missing outbound allow rule results in a rejection.

## Connecting and Sharing Different VPCs with Peering

A well-structured application is often divided into separate AWS accounts for security or organizational reasons. VPC offers multiple methods to either connect VPCs within a single AWS account or share a single VPC over multiple AWS accounts without interfering with the multi-account concept.

### VPC Peering for Connecting with VPCs in Other Regions or AWS Accounts

Peering connections allow you to route traffic between two VPCs as if they were in the same VPC. This also allows you to connect not only to VPCs in other regions but also to those in other AWS accounts.

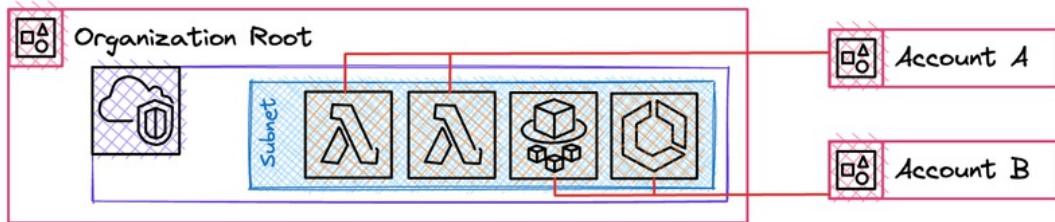


Important to remember: CIDR blocks for your connected VPCs can't overlap.

### VPC Sharing for Centrally Managed VPCs That Can Be Used by Multiple AWS Accounts

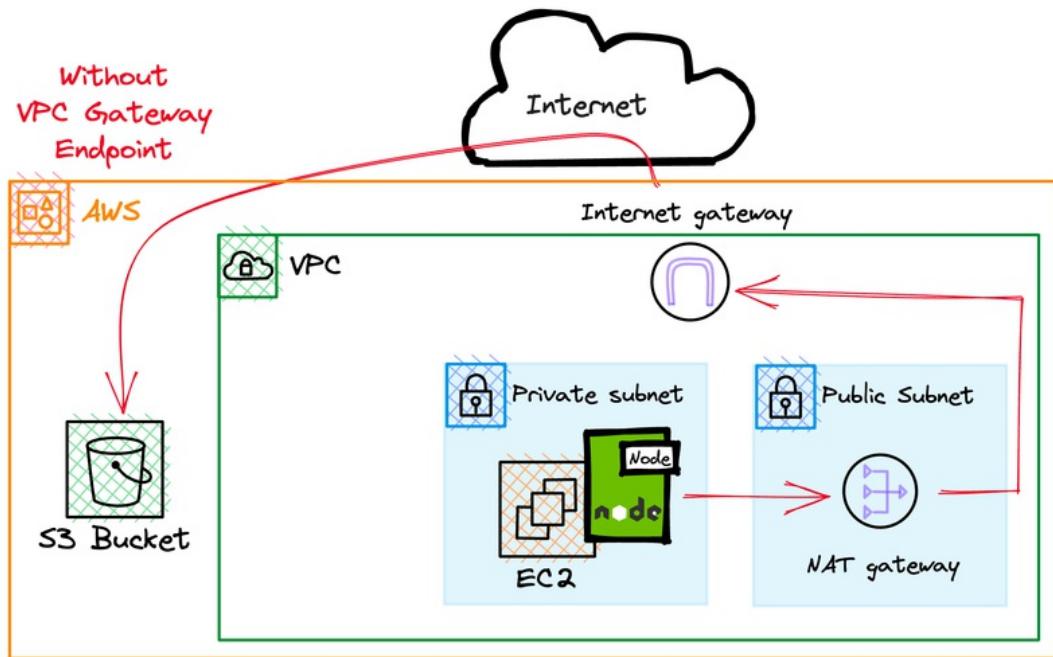
Share a VPC with other accounts that are part of the same AWS Organization so that multiple accounts can launch resources into the centrally managed subnets while still having full control of their resources. Participating accounts cannot modify resources in shared subnets that they do not own.

This allows for a fine-grained separation of accounts for billing and access control while still having components with high interconnectivity.



### Securely Accessing AWS Services within Your Virtual Network with VPC Endpoints

VPC Endpoints provide a secure and efficient way to connect AWS resources to specific AWS services, keeping data within the AWS network and minimizing exposure to the public internet.



Looking at the diagram above, we can see a common issue with a microservice architecture. In this example, a microservice runs in a private subnet and needs to communicate with an Amazon S3 bucket. Typically, the only way to communicate with anything outside of our VPC is through a NAT gateway and an internet gateway. However, this approach has several trade-offs. Firstly, the NAT gateway is charged per running minute and for the data that it handles. The cost of a managed NAT gateway can be very expensive and is a common cost trap. Secondly, our traffic leaves the AWS cloud, even though the services that we are communicating with reside in the same region.

VPC endpoints eliminate the need for public IP addresses, allowing resources within a private

subnet of a VPC to communicate within the AWS ecosystem without requiring a NAT device, VPN connection, internet gateway, or AWS Direct Connect.

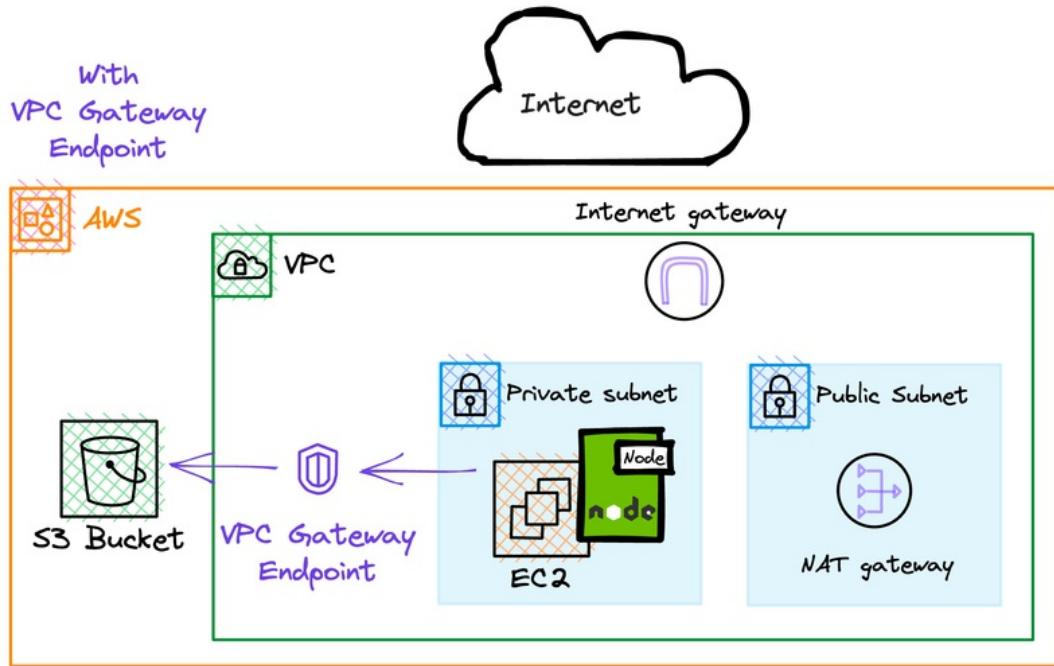
There are two types of VPC Endpoints: interface endpoints and gateway endpoints.

#### **Interface Endpoints to Access Services via AWS PrivateLink**

Interface endpoints facilitate communication with a wide variety of supported services, keeping traffic within the AWS cloud. They utilize AWS PrivateLink, and charges apply based on usage by the hour and data transfer. An interface endpoint acts as an entry point for traffic directed towards a supported service, consisting of one or more elastic network interfaces with private IP addresses. To set up an interface endpoint, you need to specify the AWS service or endpoint service you want to connect to, create a network interface, and select a subnet to associate with the interface endpoint.

#### **Gateway Endpoints to Access Amazon S3 Buckets and DynamoDB Tables**

Gateway endpoints are designed for specific services, ensuring traffic remains within the protected AWS network. Currently, only Amazon S3 and Amazon DynamoDB are supported, and they don't introduce any additional costs. A gateway endpoint directs traffic to specific IP routes in an Amazon VPC route table, typically for accessing Amazon DynamoDB or Amazon S3. To set up a gateway endpoint, you need to choose the AWS service you want to connect to, select the VPC to deploy the endpoint, identify the route table(s) to associate with the endpoint, define an access policy, and verify that the VPC security group has a rule allowing outbound traffic to the specified service.



If we compare the architecture of the updated diagram, which includes the VPC gateway endpoint, to the previous version, we can clearly see the improvements. Traffic no longer leaves the AWS cloud, and we don't even need a public DNS to resolve our records. Additionally, gateway endpoints come without additional charges, so we benefit from both a security and billing perspective.

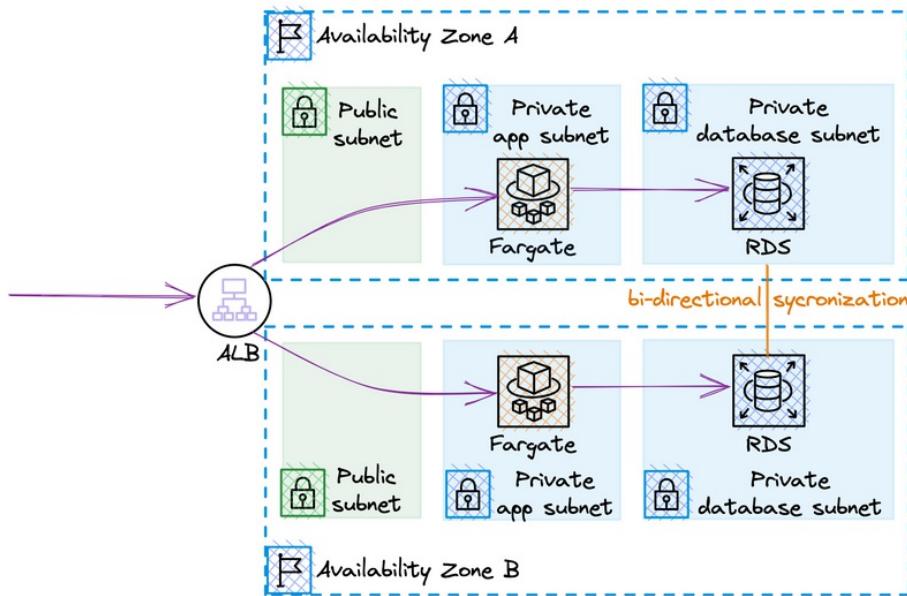
Generally, VPC Endpoints offer benefits such as enhanced security, reduced data transfer costs, potentially lower latency, simplified outbound firewall rules, reduced security risks, and enhanced compliance. By understanding the capabilities and differences between interface and gateway endpoints, you can establish secure and efficient connectivity within your VPC environment. Familiarity with VPC endpoints is a must-have, especially for enterprise-scale applications in large companies.

### The Various Use Cases for VPCs

Since VPC allows you to have full control over your network and resources, the use cases are countless.

## Use Case 1: Building and Securing Multi-Tier Architectures

With VPC, you can build multi-tier architectures that include public-facing resources like load balancers and DNS servers, as well as private application servers and databases. The application servers will still have a route to the Internet, but they are protected from the Internet since 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 only be 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. This is not only necessary from a security point of view but also from a compliance perspective. Many countries have laws that govern the storage, use, and collection of data, such as 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 in transit (while data flows between destinations) but also enforce strict access control mechanisms. These can be fulfilled by conscientiously making use of subnets, security groups, and network access control lists.

### **Use Case 3: Running Hybrid Architectures with On-Premises and the Cloud**

Many companies host their data and run their applications on-premises in their own data centers. They do this not only because they haven't finished migrating to the cloud, but also because they want to.

For these companies, it's often very useful to run a hybrid architecture between on-premises and cloud ecosystems. And that's perfectly doable with Amazon VPC, as on-premises infrastructure can be connected with AWS networks.

While sensitive workloads that require data to be stored locally can reside on-premises, 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 may be one of the least favorite services for beginners, as it requires a 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 the 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, makes 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 hand, 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

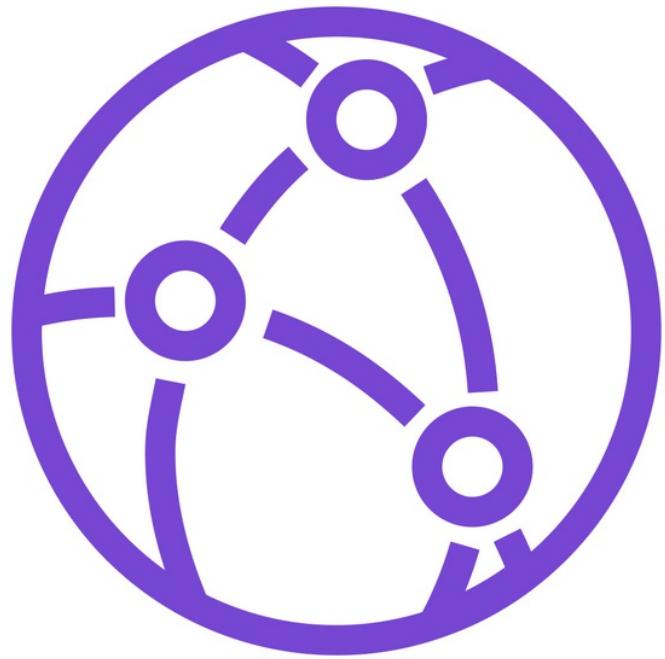
configure a fixed IP address for your instances, which makes it easier to connect to them.

- **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.
- **Make use of VPC endpoints** - Do not route your traffic through the internet if it is not necessary. For traffic that only comes from and goes to AWS services, use VPC gateway or interface endpoints that are deployed to your subnets.

## Final Words

VPC enables you to segment your network based on your security requirements, making it easy to 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. This ensures that you do not miss out on anything that an on-premise data center can offer.

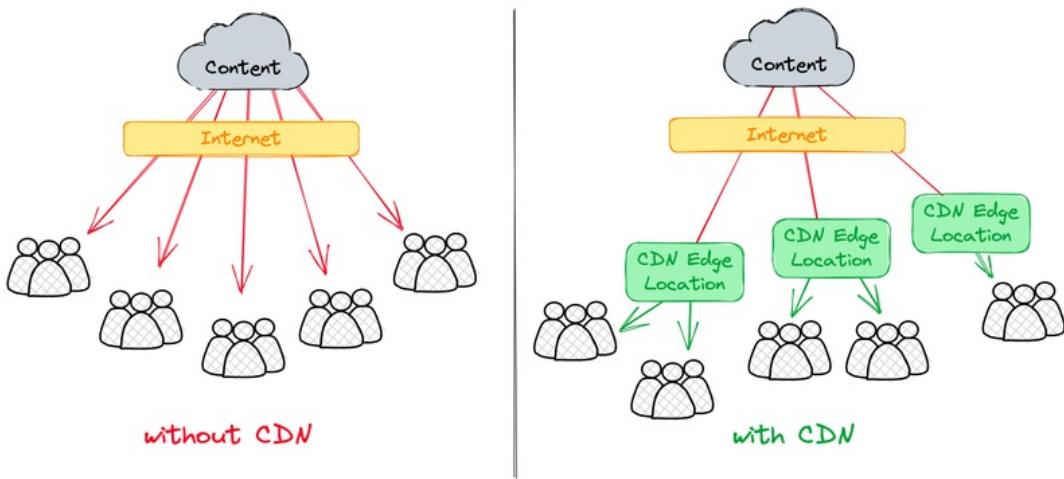


Amazon **CloudFront**

# 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 many benefits. The key aspects are:

- **Faster performance and reliability** - CloudFront supports several network-layer optimizations such as TCP fast-open, request collapsing, keep-alive connections, and 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 and CloudFront. Additionally, you can rely on geo-restrictions to prevent users from accessing your content from specific locations.
- **Customizable edge behaviors** - You are in full control of how CloudFront caches requests, accesses your origin servers, and 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, and 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 implementation of CloudFront. It includes the configuration of your origin servers - the actual endpoints that contain your content - as well as how CloudFront should cache that content based on request details.

AWS takes 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 can serve content via HTTP. A common and easy-to-set-up choice is S3.

In this example, public access to your S3 bucket is no longer necessary since your content will be distributed via CloudFront.

CloudFront will dynamically retrieve content 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.

## Controlling the Cache Behaviors within Your Edge Locations

Caching can bring a huge performance benefit for web applications, but it can also be a trap when not properly configured, as outdated content can be easily distributed.

You can choose between two caching policy types:

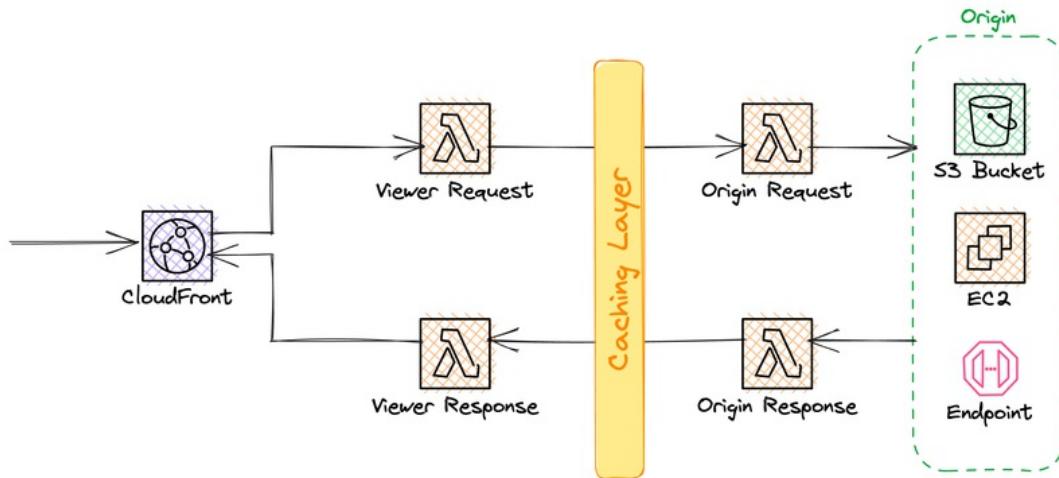
- **AWS-managed** - CloudFront will cache your requests by the domain name of your distribution and the requested path.
- **Custom** - Define which parts of the request should be used as a cache key, e.g., specific headers or cookies.

The custom policy allows for explicit configuration of your caching strategy so that you are 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 all be used 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** - executed before CloudFront requests your origin to populate its cache with the target content.
- **Origin Response** - executed 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 have total control over the requests and responses.

## **The Different Types of Functions You Can Run on the Edge**

There are two different types of functions that you can run in response to specific CloudFront events: Lambda@Edge and CloudFront functions. Both differ in their feature sets and pricing.

### **Lambda@Edge - An Extended Feature Set for Non-Basic Functionality**

Lambda@Edge comes with a 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 Inexpensive 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 network, file system, or request body access, so you are limited to working only with local variables and the request context. CloudFront functions are also not allowed to run longer than 1 ms, and you cannot set up origin request or response functions.

This means that 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, such as S3, protecting your origins is only possible via AWS IAM.

### **Origin Identities**

For our example of retrieving content from S3, you can create an **Origin Access Identity (OAI)** for CloudFront and grant this identity access to your bucket through 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-term credentials and resource-based policies, which provide more security. It also allows for other features, such as support for Server-Side Encryption via keys from KMS.

When creating your CloudFront distribution, if you choose OAC instead of OAI, CloudFront will provide you with the resource-based policy for S3.

### Protect Against Common Threats 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 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, such as SQL injection, cross-site scripting (XSS) attacks, or Denial of Service (DoS) attacks.

You can also 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.

### **Limits Access to Your Content by the Requester's Location**

CloudFront automatically detects the origin of client requests, which you can use to create approval or blocking lists.

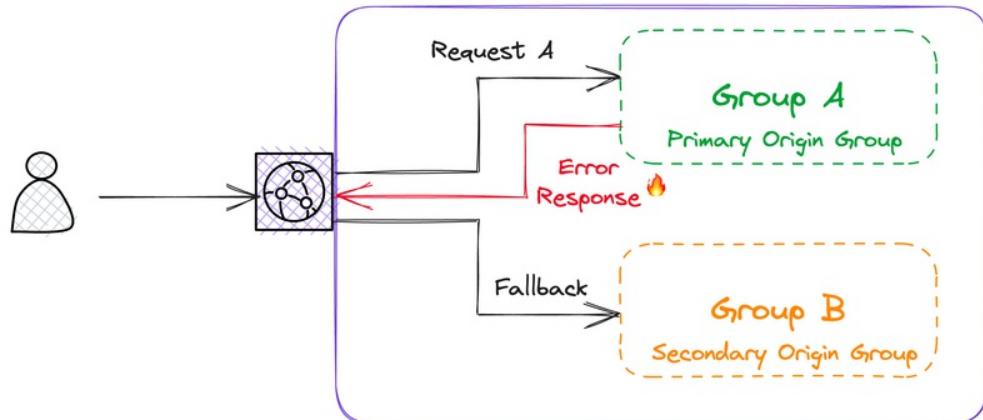
With this feature, you can either:

- Allow users to access your content only if they are in one of the approved countries.
- Block users from accessing your content if they are in one of the banned countries on your block list.

The accuracy of the mapping between IP addresses and countries is 99.8%. If CloudFront is unable to determine the location, it will always serve the requested content.

### **Redundancy and Failure-Safety via Origin Failovers**

CloudFront also provides Origin Failovers for high availability. You can define a primary and a secondary origin group and specify 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 have learned about all the fundamentals that are important to know about CloudFront. Let's jump into doing it again and create our first small setup that connects multiple services together: **CloudFront, S3, and CloudFront functions.**

What we need to do:

1. **Create a bucket** that contains a simple HTML. The bucket should block all public access, as we only want to serve the bucket's content via CloudFront.
2. **Create a CloudFront distribution** that uses our bucket as an origin.
3. **Attach a CloudFront function** to our distribution that will prompt the requester for basic authentication 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>
```

```
</body>  
</html>
```

Upload the file as `index.html` to our bucket's root directory.

The screenshot shows the AWS S3 console interface for the 'awsfundamentals-cf' bucket. The 'Objects' tab is selected. There is one object listed:

Name	Type	Last modified	Size	Storage class
index.html	html	December 23, 2022, 20:07:07 (UTC+01:00)	279.0 B	Standard

That's all we need to do in the first step.

### Setting up Our CloudFront Distribution

We can now proceed to CloudFront. Go to the CloudFront console and click on `Create distribution`. We will be prompted with a 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 creating our distribution.

## Create control setting

X

### Name

The name must be unique. Valid characters: letters, numbers and most special characters. Use up to 64 characters.

### Description - *optional*

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 have everything we need. We'll stick to the default cache settings and won't 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](#)

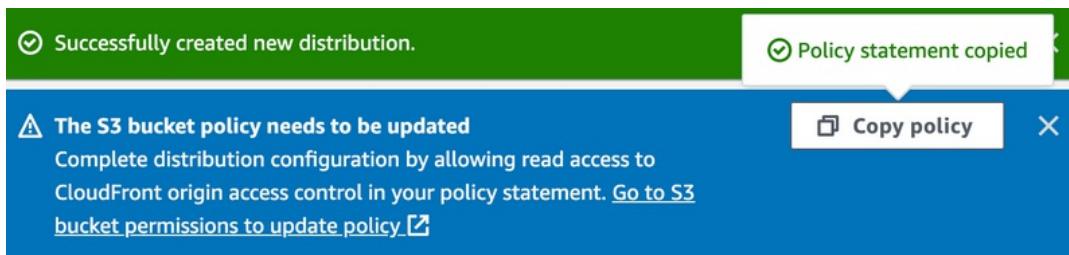
**Bucket policy**  
Policy must allow access to CloudFront IAM service principal role.

- 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 mentioned before, we are prompted that the generated control policy is ready to be copied.

Click on the `Copy policy` button and navigate to our S3 bucket.



Go to your bucket, then to the Permissions tab, and edit your bucket policy. Paste the required policy, which 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 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 "/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 authentication 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 is set to `awsfundamentals` and `Password1!`.

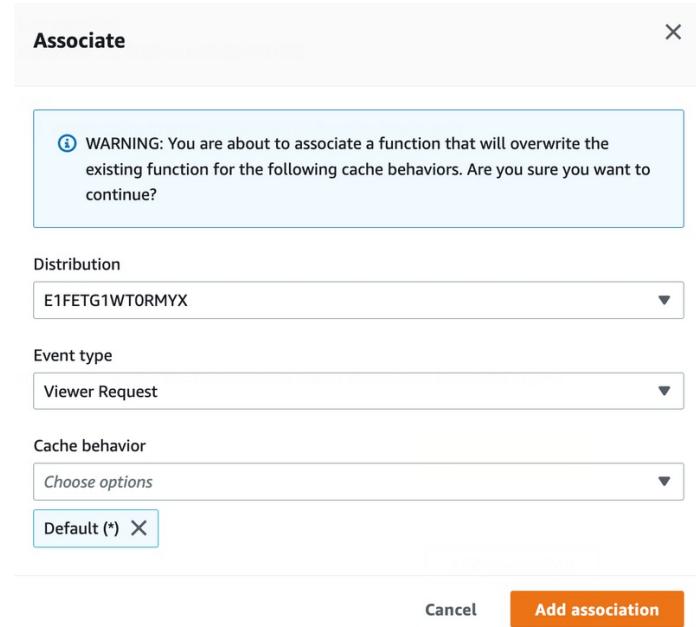
Let's navigate to the `Test` tab of our function and check if it performs as expected. If we provide the correct header, we should receive an HTTP response.

The screenshot shows the AWS Lambda Test function interface. At the top, there are fields for 'Header' (containing 'authorization' with value 'Basic YXdzZnVuZGFtZW50YWxzOlBhc3N3b3JkMSE='), 'Value' (with a 'Remove' button), and 'Add header' (button). Below these are sections for 'Request cookies - optional' (with 'Add cookie' button) and 'Query string - optional' (with 'Add query string' button). At the bottom right are 'Cancel' and 'Test function' buttons. The main area displays the 'Execution result' with a green checkmark. It shows 'Status: Succeeded', 'Stage: DEVELOPMENT', and a 'Compute Utilization Info' section with a green circle containing the number '16'. The 'Output' section contains a JSON object representing the request details. A 'Execution logs' section is also present.

When an authorization header is not provided or an invalid value is provided, the server returns HTTP 401. This behavior is expected.

The screenshot shows the AWS Lambda Test function interface. The 'Execution result' section has a red exclamation mark icon and displays 'Status: 401 Unauthorized'. It shows 'Stage: DEVELOPMENT' and a 'Compute Utilization Info' section with a green circle containing the number '19'. The 'Output' section is empty.

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.



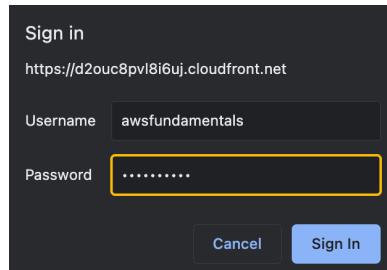
Select our distribution, choose `Viewer Request`, and select the default cache behavior.

Associated distributions		<a href="#">View distribution</a>	<a href="#">Remove association</a>	<a href="#">Add association</a>
Distribution ID	Description	Cache behavior	Event type	
E1FETG1WT0RMYX	-	Default (*)	Viewer Request	

That's it! If we go back to the distribution overview, we'll see that CloudFront is deploying our update. This may take several minutes, so don't be surprised if it takes a while.

Origins	Status	Last modified
awsfundamentals-cf.s3.eu-west-1.amazonaws.com	<span>Enabled</span>	<span>Deploying</span>

Once the deployment is successfully completed, we can revisit the URL of our distribution and retrieve the HTML file. At this point, we will be prompted to enter our credentials.



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 estimate of 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 that enable you to analyze how your distribution is used and by which audience.

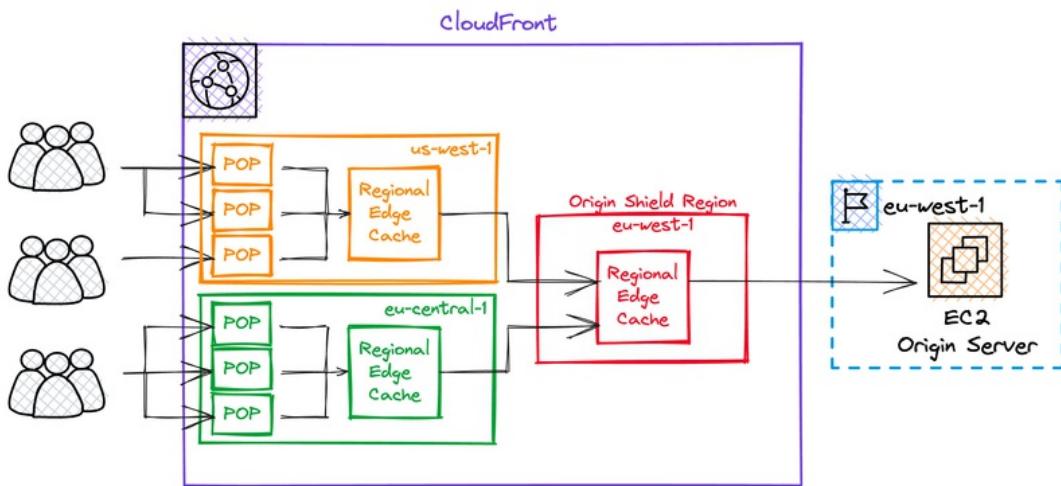
- **Cache Statistics** - provides an overview of requests by status code and method, cache

hits, misses, and errors.

- **Popular Objects** - shows the most requested files, including the cache-hit ratio for those files.
- **Top Referrers** - lists the top 25 sources for requests.
- **Usage** - shows the number of requests and transferred data by protocol and destination.
- **Viewers** - includes information on devices, browsers, operating systems, and locations.

### Leveraging Origin Shield to Further Improve CloudFront's Performance and Protect Your Origin Servers

Origin Shield is a feature provided by Amazon CloudFront that helps improve cache hit ratios and reduce the load on your origin servers. It acts as an additional caching layer between CloudFront edge locations and your origin servers. Origin Shield is a protective feature that safeguards your origin servers from overload, ensuring high availability.



When you enable Origin Shield, CloudFront selects a single edge location as the shield location. This location serves as the central cache for your origin content.

When a request is made to a CloudFront edge location, if the requested content is not already cached at that edge location, CloudFront forwards the request to the shield location. The shield location then fetches the content from your origin server, caches it, and sends the response back to the requesting edge location. The edge location then caches the content and serves it to the end user.

## The Many Use Cases of CloudFront

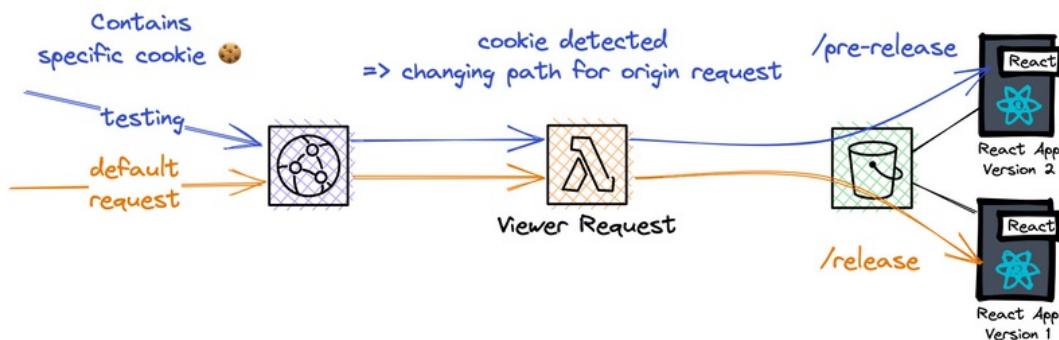
We have already discussed the benefits of using a CDN for distributing your content. This is true for any web application that delivers static content in any form. However, due to CloudFront's capability of running custom business logic at its edge locations, you can do a lot more with just a few lines of code.

Let's take 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 is common practice to have a preview stage for testing 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 is 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.

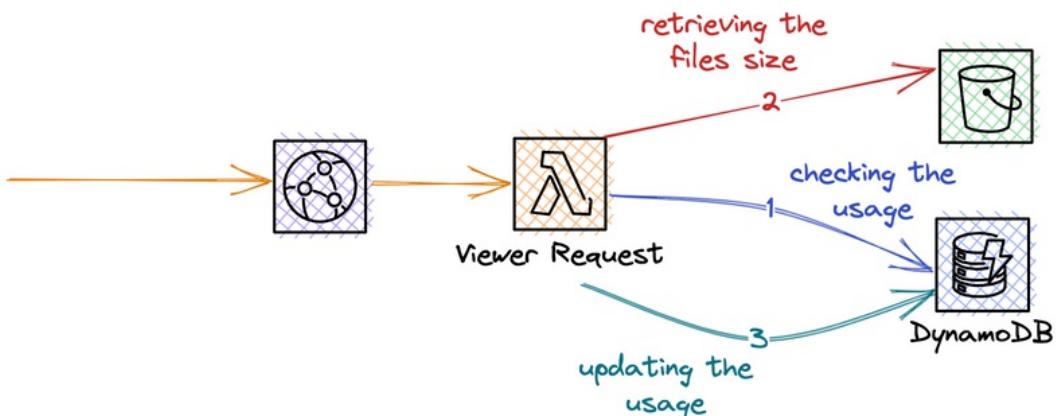
## Use Case 2: Counting Transferred Bytes and Limiting Downloads per Customer

When using Lambda@Edge, you are not just limited to the variables in your function and the request context. Your edge functions have network access, which allows them to access other AWS services if the required permissions are granted via IAM, resulting in almost limitless possibilities.

Let's 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 collect too many power users.

A simple Lambda@Edge function can 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 does not come with much or any additional costs if your application does not reach a very large scale.

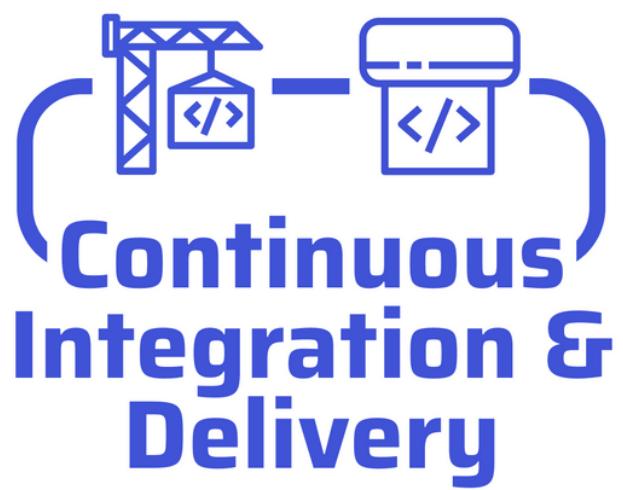
## Tips and Tricks for the Real World

CloudFront is a powerful tool that allows you to not only host simple websites close to anyone in the world, but much more. Let's go through some important reminders that will help you get the best out of CloudFront.

- **CloudWatch automatically integrates with CloudFront** - 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 be 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 have 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 still be 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 to give an easy way to distribute content to end-users with low latency and high transfer speeds.



# Continuous Integration & Delivery

Nowadays, software development is an iterative process. We have moved away from the waterfall model, where we design an application from A to Z, build all features in one go, and ship everything to production.

One of the key principles of agile development is to break the software development process into small, manageable chunks. A cross-functional team of developers, designers, and other stakeholders work together to deliver usable software in small cycles.

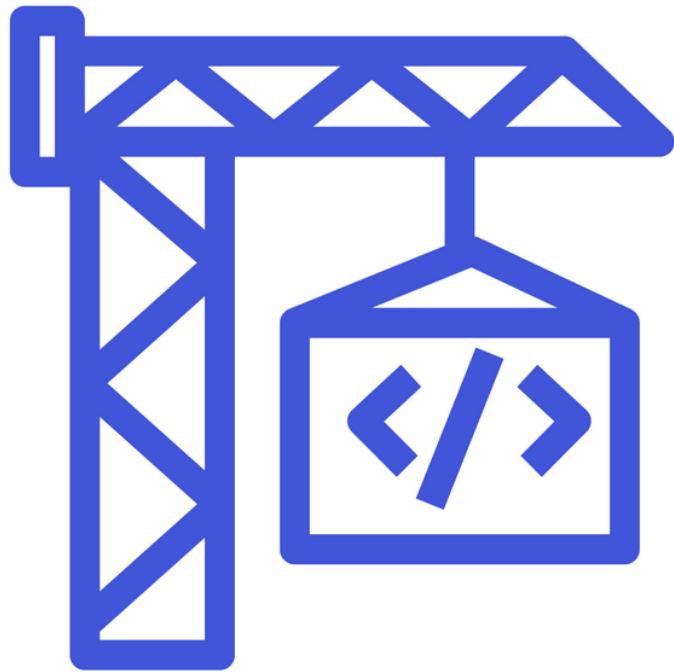
Releasing software has to be an automated process to ensure quality standards are met, and existing features are not broken with updates. This is where **continuous integration & delivery (CI/CD)** tools step in.

**Continuous integration** is the practice of continually integrating code changes into a shared code repository. As developers write code, they frequently push it to the repository, where it is automatically built and tested. This helps catch errors early in the development process and reduces the risk of integration problems later on.

**Continuous delivery** automates the deployment of code changes to different environments such as development, QA, and production. This enables 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 built on top of CodeBuild. It allows you to model and visualize the entire release process by building delivery pipelines that are constructed via different CodeBuild jobs and go from checking out source code to deploying infrastructure and the final distribution to production.

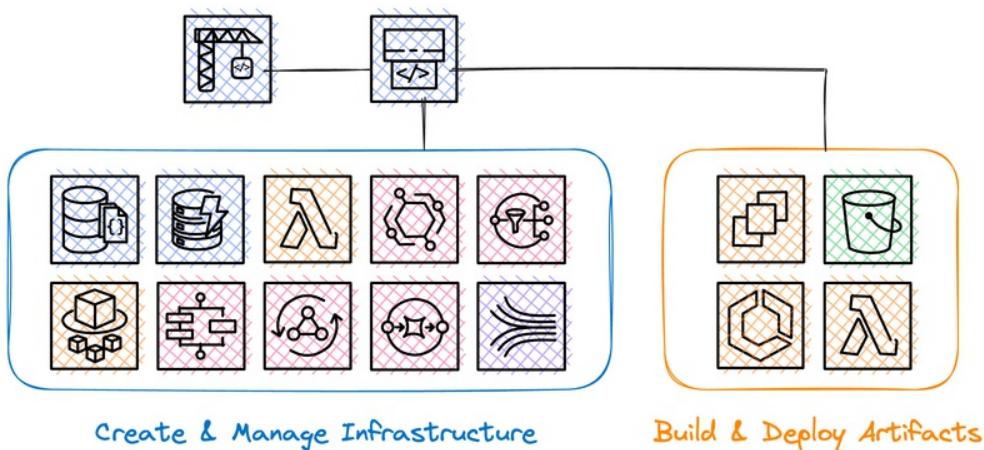


## AWS **CodeBuild** & **CodePipeline**

# Creating a Reliable Continuous Delivery Process with CodeBuild & CodePipeline

## Introduction

AWS CodeBuild and AWS CodePipeline are fully-managed continuous integration services that help you build, package, and deploy your application in a safe and reliable manner. They enable your team to focus on building the actual application and worry less about the effort or operations required to ship it to customers.



CodeBuild and CodePipeline help you build your deployment package and can also create and manage infrastructure via Infrastructure as Code tools such as Terraform, CloudFormation, Serverless Framework, CDK, and others.

## An Automated Continuous Integration & Delivery Process Ensures Reproducible and Reliable Application Releases

Compiling applications usually requires a specific environment that offers all the necessary tools, languages, and frameworks that are part of the build process. Additionally, you want to regularly deploy changes to your application or product to ensure that the current development version available for internal or external testing doesn't differ too much from the latest state of development. Thirdly, even a small team of developers is usually very diverse, not only from a cultural perspective but also from a technical point of view. Everyone has their own preferences for development tools or even operating systems.

We also want to decouple the development process from the release process and restrict manual production access by developers and team members. This is important to avoid human error. It's often 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 developers for our build processes**, 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 can be summarized as follows: We require a dedicated build environment with a stable toolchain that runs without the need for human interaction and is strictly separated from a security perspective.

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**

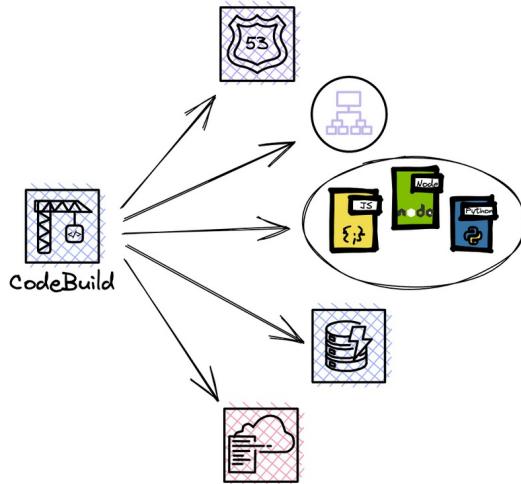
As the name implies, CodeBuild is the service that actually **builds**. But don't get too caught up in this phrase, as you can execute **anything** with CodeBuild. It doesn't just strictly map down to packaging and building applications.

You can also:

- 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.

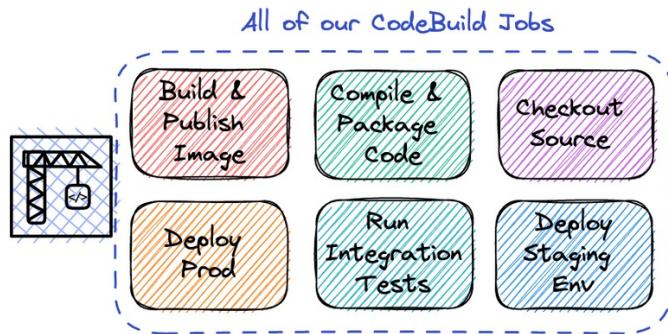
- Or any other process that can be poured into a script.

As CodeBuild can run any container image you want to provide, there are no limits on which tasks it can take over.

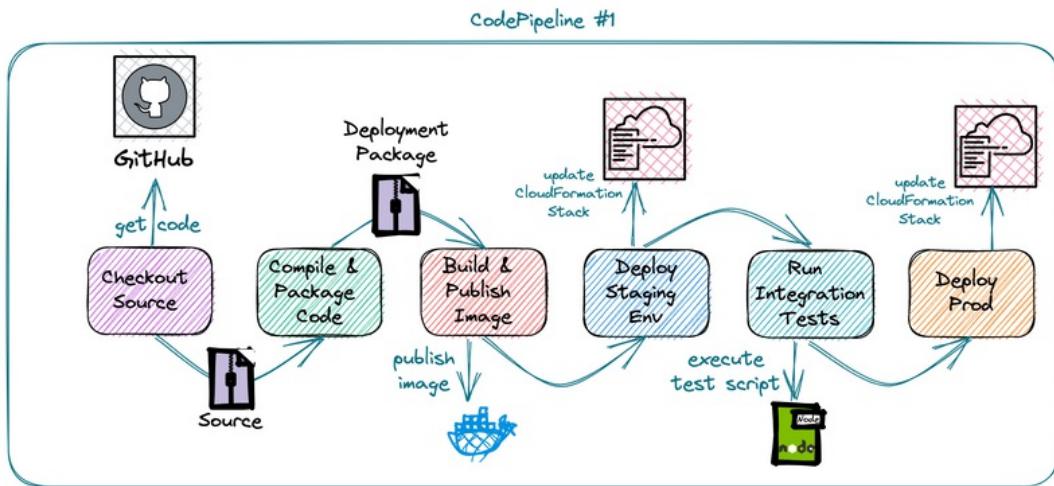


### AWS CodePipeline for Orchestrating All Your Build Jobs and Deployments

A structured and well-thought-out delivery process doesn't just include a single build job. It's an orchestration of many jobs that enables you to deploy code or infrastructure in a reliable and safe manner.



And that's exactly what AWS CodePipeline does: it is 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 and delivery processes.

These 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

To understand how CodeBuild and CodePipeline work, we need to review their key terms and fundamentals.

## **Build Images & Containers as the Underlying Base for Your Jobs**

Builds are executed within containers. For the images that run within the containers, you 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 do. Each spec file is a YAML file 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 into different phases which are executed by CodeBuild. Let's take 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 Structure 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 has been submitted to CodeBuild's internal job queue.
- (32s) **Queued** - The job is waiting for unreserved compute resources.
- (53s) **Provisioning** - Downloads your image and 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 upload them to S3.
- (2s) **Finalizing** - Completing the job and freeing resources.
- **Complete** - The build step has been completed.

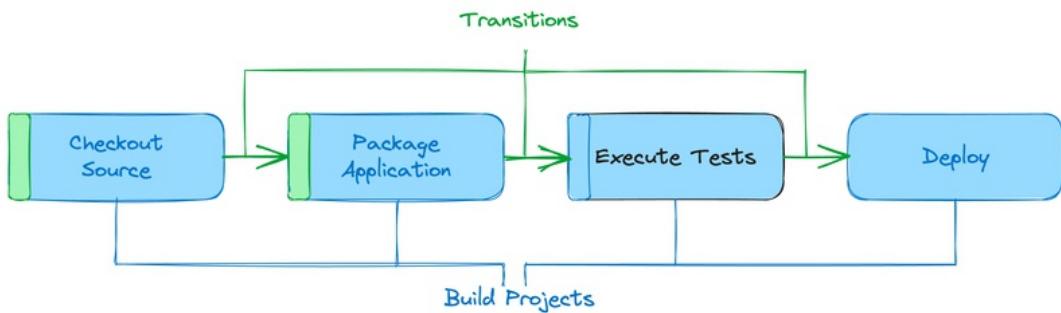
### Sources to Define Where Your Code Comes From and Triggers to Automatically Run Your Jobs Based on Conditions

CodeBuild can check out a repository at the beginning. As a source provider, you can use Amazon S3, AWS CodeCommit, GitHub, or BitBucket.

Moreover, you can define whether builds should be triggered automatically on source changes, such as 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. They allow for structuring continuous integration and deployment into phases, including builds, quality gates, and actual deployments.



The actual structure 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 is an option to create additional output files via `secondary-artifacts` with a unique name.

## **Monitoring Your Delivery Pipeline to Quickly Become Aware of Issues**

Your CodeBuild and CodePipeline projects execute the most critical actions in your AWS account. That's why monitoring is crucial, as it is with every other resource.

### **Logging**

All console output in CodeBuild is, by default, ingested in 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, such as Slack.

You can also 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 and 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 cannot 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 require access.

- Regression tests require directly verifying cached data in 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 does not need to be integrated with a private VPC as it is fully protected by IAM.

#### **Encrypting Your Artifacts to Add Additional Protection for Confidential Outputs of Your Build Jobs**

Your build outputs are sensitive information. As they will be saved in S3, you can enable encryption, just like with all other objects. You can either choose to use the S3-managed encryption key or opt for a dedicated customer-managed key from KMS specifically for your CodePipeline project.

This also applies to caching.

#### **Enforcing Manual User Interaction for Critical Stages via Approvals**

Your typical release pipeline does not run through all stages without user interaction. Most teams want to have a fixed release cycle that requires developers to manually approve the deployment to production.

CodePipeline meets this requirement via approval steps that can be integrated between two build projects and require a response from a user who has the necessary `codepipeline:PutApprovalResult` permissions assigned.

Users will also be asked to provide a message for either approval or rejection, so decisions can be traced later.

#### **Getting an Understanding of How CodeBuild and CodePipeline are Billed**

As mentioned earlier, CodeBuild and CodePipeline are pay-per-use services, except for a small fee per created CodePipeline project.

The price per build minute depends on the memory and CPU configuration of your build container. AWS specifies this as **small**, **medium**, or **large**.

## **Advantages and Downsides**

After reviewing the basics, it's important to summarize what AWS CodeBuild and AWS CodePipeline do well and what could be improved in the future. In any project, you'll spend a significant amount of time with continuous integration and delivery services, so it's important that they meet your requirements.

### **CodeBuild and CodePipeline Excel in On-Demand Pricing, Low Operations, and High Reliability**

Both services get a lot of things right:

- **Pay-as-you-go pricing** - You'll never pay for idle 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.
- **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.
- **Highly available** - Both services offer high availability and 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 look at the most prominent criticisms.

- **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 arranged** - 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 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 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 day Either 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 container image. Relying on non-managed container images results in operational costs in keeping them up to date by updating them regularly. 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 primary 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 went through an example of how to run applications wrapped in a Docker image in ECS. In a real-world scenario, the process of deploying a new version of our application would be automated.

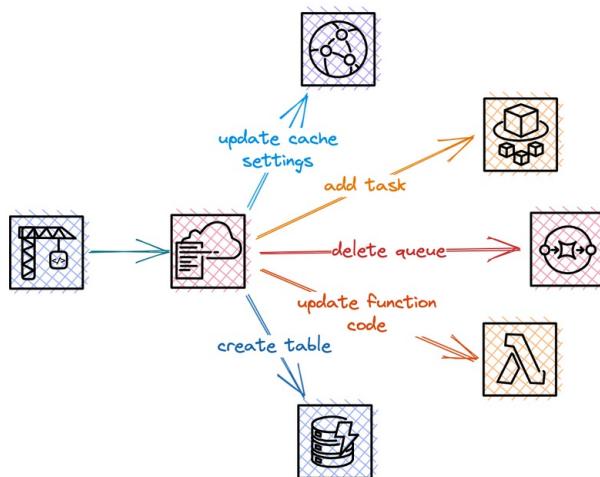
This involves several steps:

- 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 done solely 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, but they are also used to create and update your entire account's infrastructure via Infrastructure-as-Code tools like CloudFormation, AWS Cloud Development Kit (CDK), or Serverless Framework.



We will 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 is important to have multiple environments that are as similar as possible. It is also important to integrate automated quality gates into your release process so that you can ensure (to some degree) that things are working as expected.

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

It does not 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 us 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 will 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** - do not 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 do not get actively forwarded to places that get much attention, nobody will take care until it is 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 have 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 will not 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, comparatively slow build times, and a list of missing features.

But nevertheless, it is 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 is not just about building them; it's also about running and operating them. This daily task requires deep insights into how an application behaves by monitoring key metrics like request rates, response times, error rates, and resource utilization. It is also necessary to quickly detect and diagnose issues and bottlenecks that result in performance issues.

All of these topics can be bundled into the topic of observability, and Amazon **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 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.

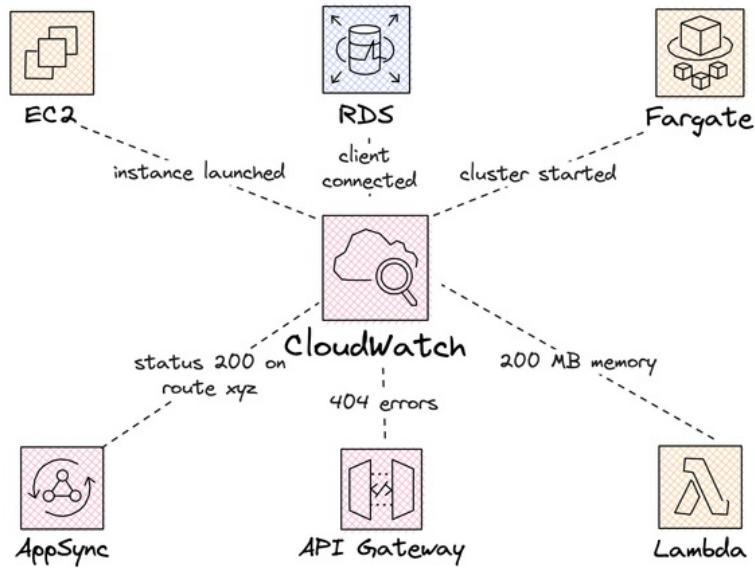
CloudWatch also collects metrics, such as the number of Lambda invocations, free space in your database, or how much CPU your ECS cluster uses. Based on these metrics, you can create **alarms** for certain thresholds. If your free space falls below a certain threshold, you will be notified.

CloudWatch is one of the **most underrated** services for beginners to learn. You will need to use CloudWatch **a lot**, especially to understand how your application performs and to debug it. It doesn't matter how you develop your application on AWS; you will need to use CloudWatch.



CloudWatch is divided into different products. We will mostly focus on **CloudWatch Logs**, **CloudWatch Metrics**, and **CloudWatch Alarms**. We will also briefly 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": "ghjkgkhjk-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.

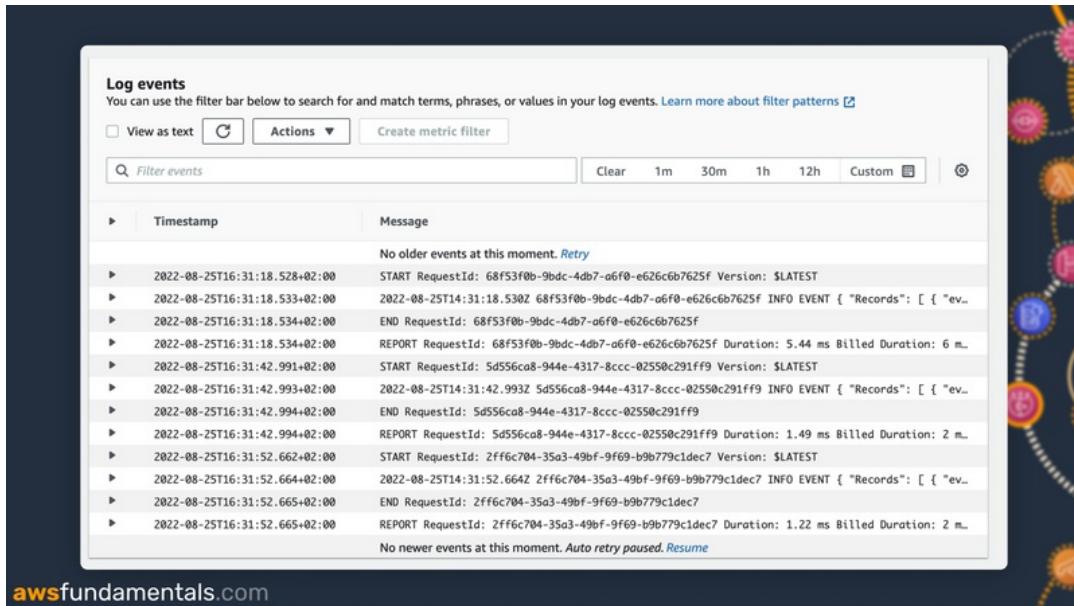


Name	Definition
<b>Log Events</b>	A log event is your actual log statement. It contains the timestamp of your log and the raw log statement you put into it.
<b>Log Streams</b>	A log stream contains one or more log events from the same source. For example, a log stream of a Lambda function can contain 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.

Name	Definition
Metric Filter	You can create metric filters to get metrics out of your log events.
Retention Setting	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 header with 'Log events' and a note: 'You can use the filter bar below to search for and match terms, phrases, or values in your log events. Learn more about filter patterns'. Below the header are buttons for 'View as text' (unchecked), 'Actions', and 'Create metric filter'. A search bar labeled 'Filter events' is followed by a 'Clear' button and time range buttons for '1m', '30m', '1h', '12h', and 'Custom'. To the right of the search bar is a 'Retry' button. The main area displays a table of log events:

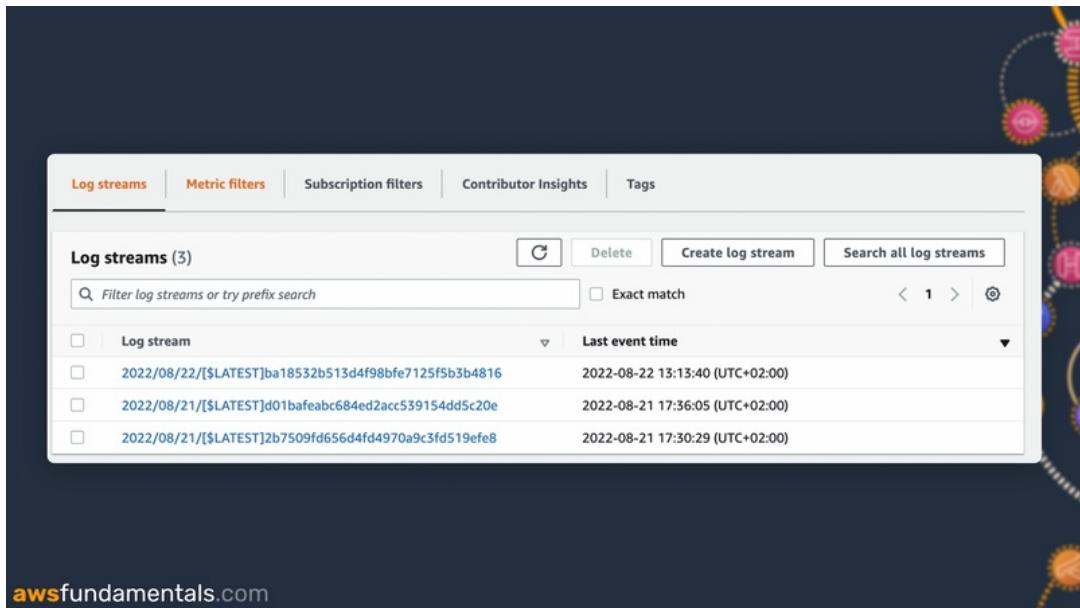
	Timestamp	Message
No older events at this moment. <a href="#">Retry</a>		
▶	2022-08-25T16:31:18.528+02:00	START RequestId: 68f53f0b-9bdc-4db7-a6f0-e626c6b7625f Version: \$LATEST
▶	2022-08-25T16:31:18.533+02:00	2022-08-25T14:31:18.530Z 68f53f0b-9bdc-4db7-a6f0-e626c6b7625f INFO EVENT { "Records": [ { "ev...
▶	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...
No newer events at this moment. <a href="#">Auto retry paused. Resume</a>		

At the bottom left, the URL 'awsfundamentals.com' is visible.

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

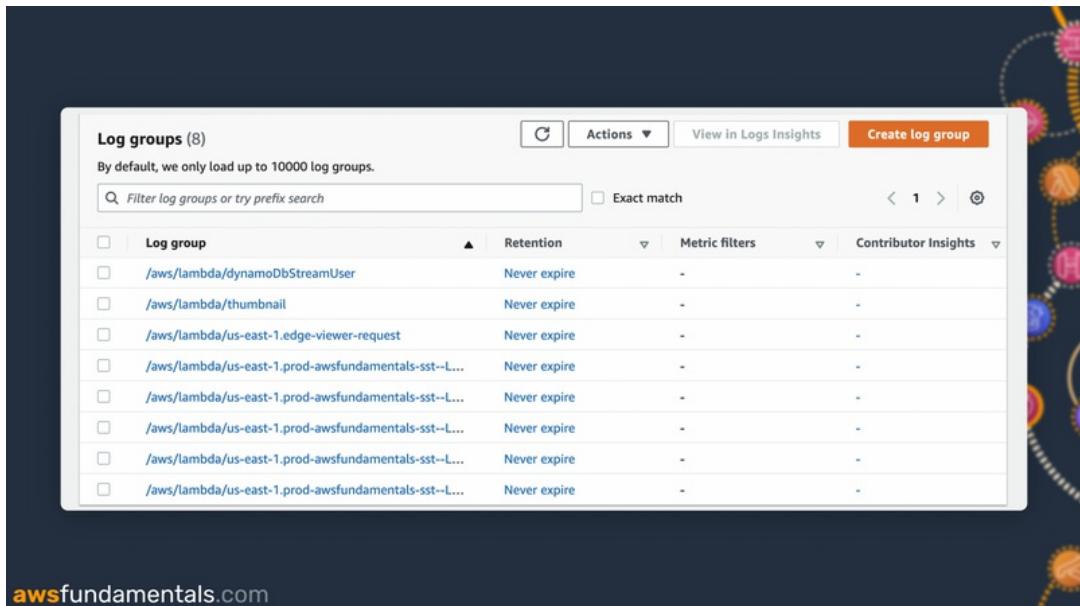
A log stream is a container for all log events.



In the case of Lambda, one log stream belongs to one **warm Lambda container**, meaning a Lambda container that has not been destroyed. The Lambda chapter explains cold and 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:



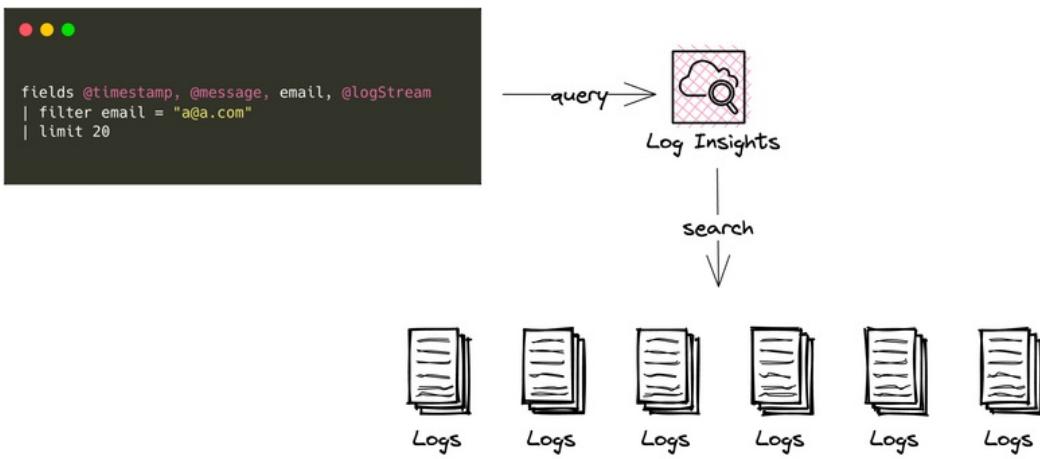
The name of a log group is prefixed with `/aws` and the service name. For Lambda, this is:

`/aws/lambda/<FUNCTION_NAME>`.

Each **Log Group** contains different **Log Streams**. The retention settings are associated with the Log Group, and 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, and clicking on "Open Logs" in CloudWatch.

#### CloudWatch Log Insights Allows You to Query Logs like a Database



CloudWatch Log Insights is an amazing product built on top of CloudWatch Logs. It allows you to query your logs using a SQL-like query language.

Tracing logs across different log groups can be quite challenging, but 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, which can be an autogenerated ID attached to every log or an ID from your business context, such as 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 similar to SQL at first glance. You describe what you want, add filters, and sort

arguments, and the engine (Logs Insights) gives you the results.

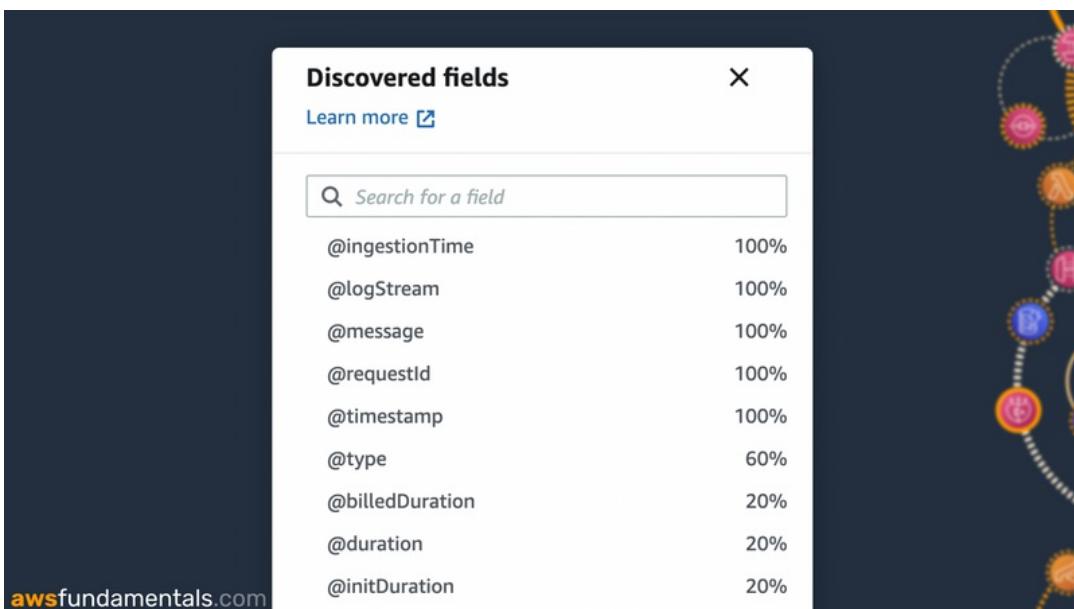
For this query, we want to see the `timestamp` and the message for the user `user_1`.

### 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 then shows you all the available fields with the percentage of how often they were available. For example, in 60% of all logs, a field named `@type` was available.



### You Can Build Queries with `fields`, `filters`, `sort`, and `limit`, and More

Let's build a quick query that filters on the `correlationId` field.

```
fields @timestamp, @message
| filter correlationId like "f96eea7e"
| sort @timestamp in descending order
| limit 20
```

What is happening here?

- `fields` selects the fields that you want to display, which in this case are the timestamp

and the actual message.

- `filter` filters all requests based on the expression that follows. In this case, we are filtering for a correlation ID.
- `sort` sorts based on the timestamp.
- `limit` limits all requests to 20.



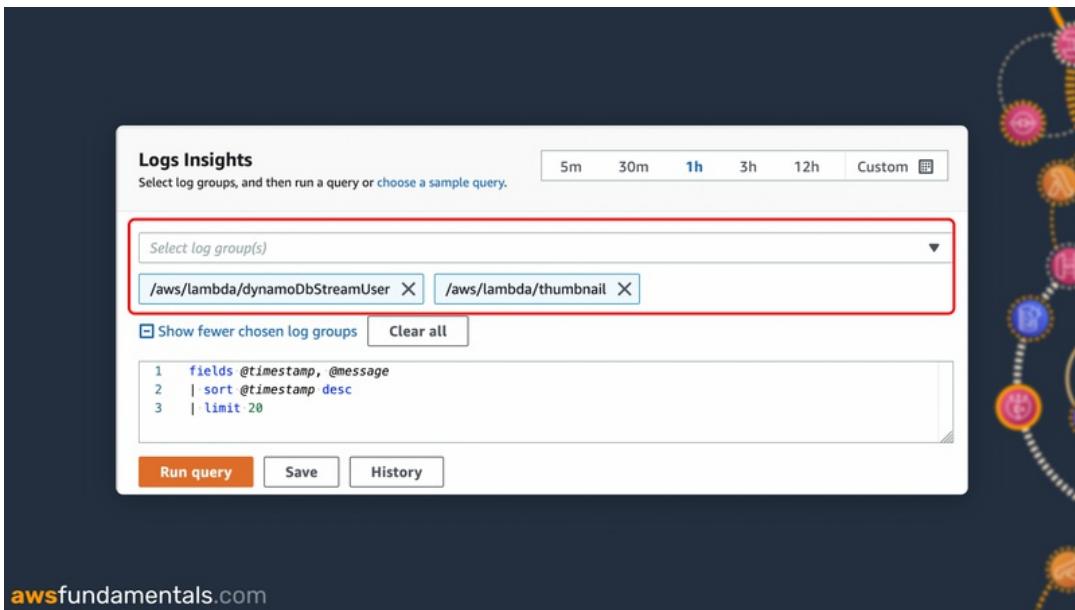
The final result looks like the image above. We have one matching event. By selecting the `@logstream` field, we have a direct link to the log stream. This only works if you select **one log group**.

### **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.



## Log Insights Has Pre-Made Queries Available

Log Insights also has some pre-made queries available on the right side. You can find examples of different AWS Services such as 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 create even more complex reports. You can use basic aggregations such as max, min, or average.

We won't dive deeper into Log Insights right now. However, if you want to properly understand and trace your requests, we definitely recommend using this service frequently to improve your skills.

These are the basics of CloudWatch Logs. Logs are an **essential** part of any cloud application,

and CloudWatch gives you the capability to understand your distributed application by consolidating all logs in one place.

## CloudWatch Metrics Store Metrics about Your Services

Next, we will look at the second basic functionality of **CloudWatch Metrics**.

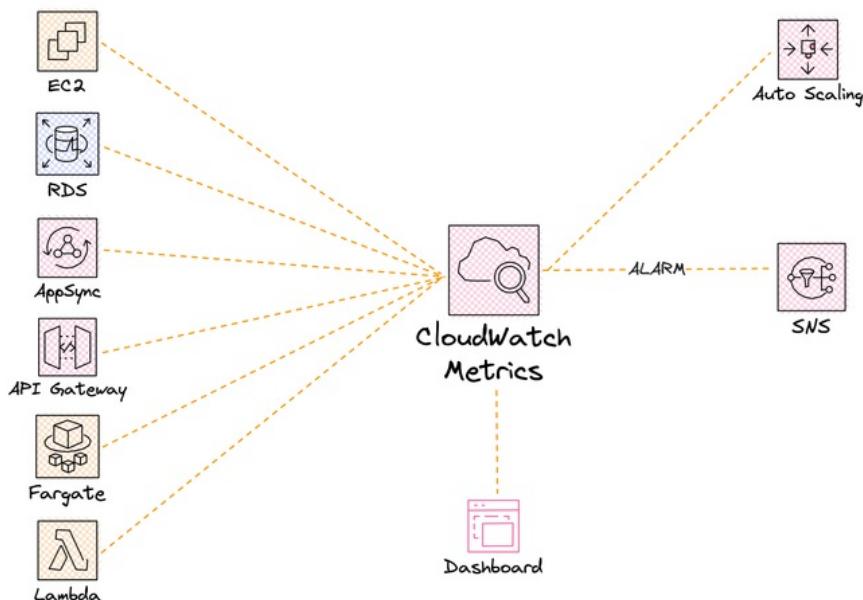
CloudWatch acts as a metric repository. Each application sends default metrics to CloudWatch. You can use CloudWatch to understand how your application behaves.

Let's use the Lambda service as an example again. Example metrics include:

- Number of invocations
- Execution time of the Lambda
- Number of errors

You can also use statistical functions such as averages, sums, or medians. You can create a dashboard to present these metrics to your stakeholders.

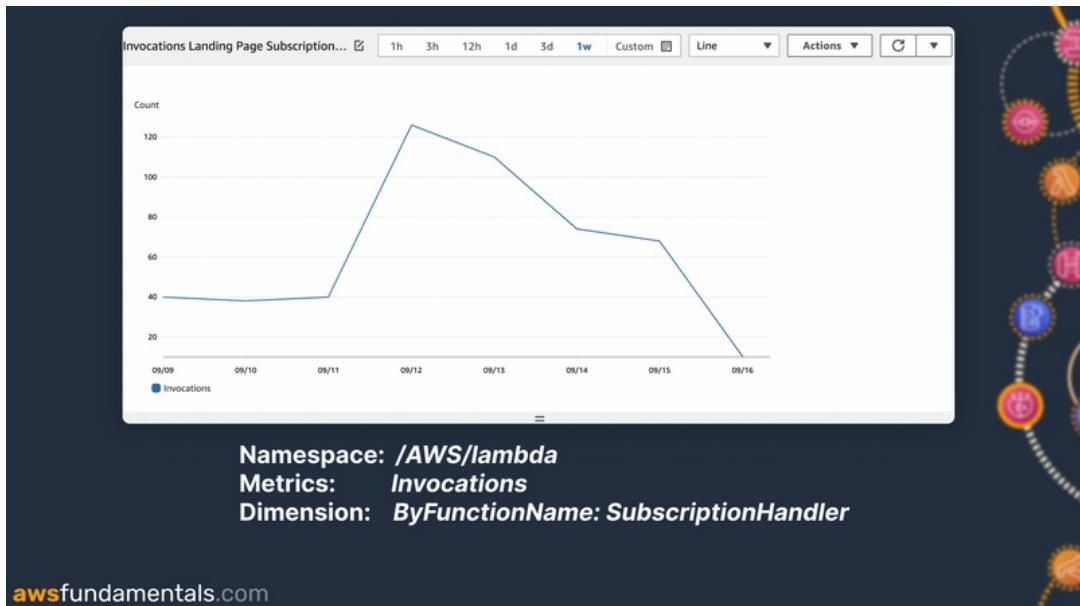
Based on the metrics, you can also create **alarms**. Alarms notify you if a metric meets a predefined threshold. CloudWatch Metrics even allows you to **scale your resources**. For example, if an EC2 instance is almost out of memory, you can scale it with CloudWatch.



This image shows how CloudWatch metrics work. You have several services that put metrics into CloudWatch. A CloudWatch alarm can use either SNS or Autoscaling as an action. SNS is used for notifying you and your colleagues, while Autoscaling is used 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 (o) 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.
<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.

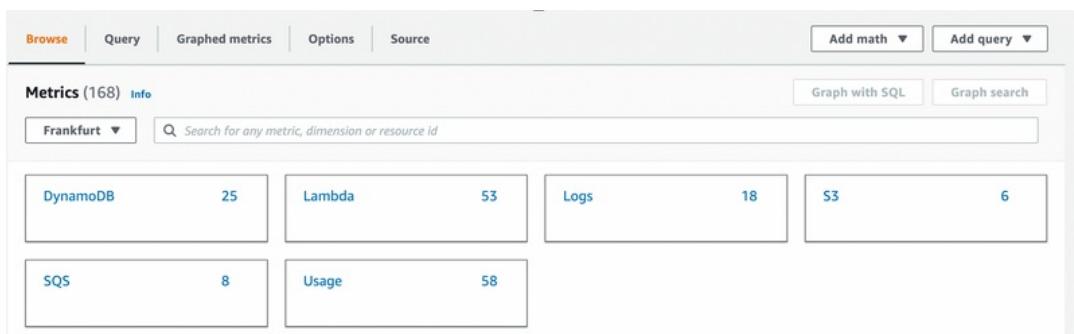
- Namespace: /AWS/lambda
- Metric: Invocations
- Dimension: byFunctionName: SubscriptionHandler

It displays the count of invocations per day over one week.

### Creating Metrics with the CloudWatch Browser

The CloudWatch console provides several options for viewing your metrics.

Let's take a look at an example. Click on CloudWatch Metrics and you will see this window:



In the **Browse** tab, you can see all the different services that have metrics available. You can

click on a service, such as Lambda, to see all its dimensions.

The screenshot shows the AWS CloudWatch Metrics console. At the top, it says "Metrics (55) Info". Below that, there's a search bar with "Frankfurt" selected and a dropdown for "All". A breadcrumb navigation shows "Lambda > Lambda". There's also a search bar with "Search for any metric, dimension or resource id". On the right, there are two buttons: "Graph with SQL" and "Graph search". Below the search bar, there are three categories: "By Resource" (26), "By Function Name" (21), and "Across All Functions" (6). The "Across All Functions" button is highlighted.

Dimensions here are:

- By Resource
- By Function Name
- Across All Functions

We chose **Across All Functions** because we want to see metrics across all of our Lambda functions.

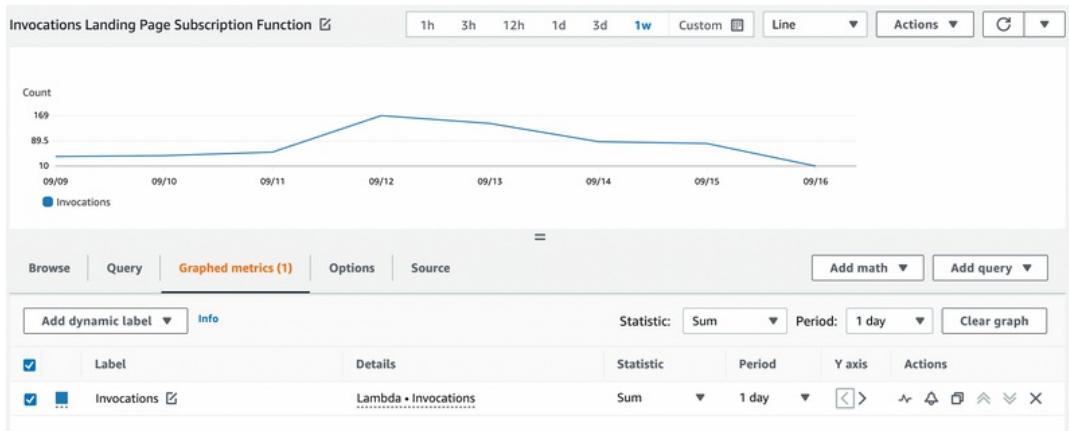
The screenshot shows the AWS CloudWatch Metrics console with "Metrics (6) Info". The breadcrumb navigation shows "Frankfurt > All > Lambda > Across All Functions". There's a search bar with "Search for any metric, dimension or resource id". On the right, there are two buttons: "Graph with SQL" and "Graph search". On the left, there's a list of metrics with checkboxes:

- Metric name (6)
- ConcurrentExecutions ▾
- Duration ▾
- Errors ▾
- Invocations ▾
- Throttles ▾
- UnreservedConcurrentExecutions ▾

Now, you can see the different metrics available for Lambda. These include:

- ConcurrentExecutions: the number of Lambda functions running at the same time
- Duration: the length of time the functions ran
- Errors: the number of errors that occurred
- Invocations: how often the Lambda was executed

By checking the checkbox of a metric, it will appear in the graph at the top of the screen and be added to the **Graphed Metrics** tab.



You can then proceed to adjust the following:

- The timeframe at the top of the page
- The statistical function (sum, average, min, max, etc.)
- The period at the top of the screen
- Rename labels, axes, and titles
- Change the type of chart (line, bar, number, etc.)

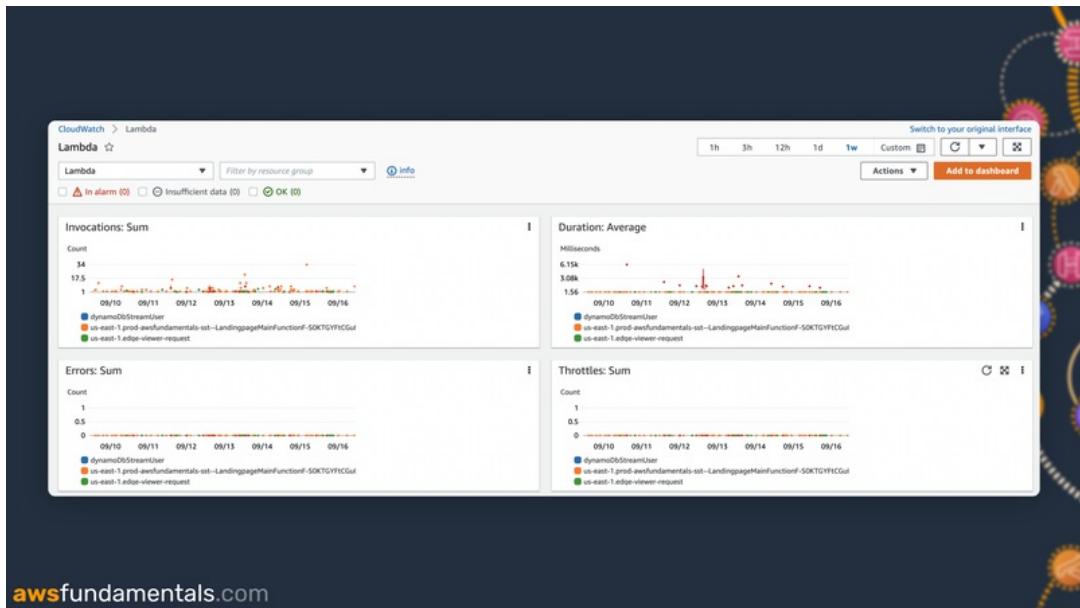
With this, you can get a powerful overview of all your services.

### **Dashboards Give You an Overview of Metrics and Alarms**

Dashboards provide an opportunity to get an overview of various metrics and alarms simultaneously.

CloudWatch creates automatic dashboards for you. You can access them by opening CloudWatch and heading to the tab **Automatic dashboards**. They are available for different services such as DynamoDB, Lambda, and CloudWatch (itself). However, you can also create custom dashboards.

Let's take 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, providing fast access to metrics for everyone.

Dashboards are similar to creating the graphs above for metrics, as they contain several of these graphs.

Dashboards cost \$1 per month after the first one, meaning that one is free in the free tier.

### **CloudWatch Alarms Notify You on Pre-Defined Thresholds like CPU Usage or Errors**

Let's consider a common bad scenario: your server goes down, and you only find out because customers are calling you. This is often typical for companies with poor monitoring systems. You want to be informed about downtimes **before** your customers experience them. CloudWatch Alarms can help you with that.

CloudWatch Alarms notify you when a system or service reaches a pre-defined threshold, such as errors or usage.

It is essential to spend time defining your alarm thresholds. By having the correct ones, you will know when something bad happens, even before your customers know.

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

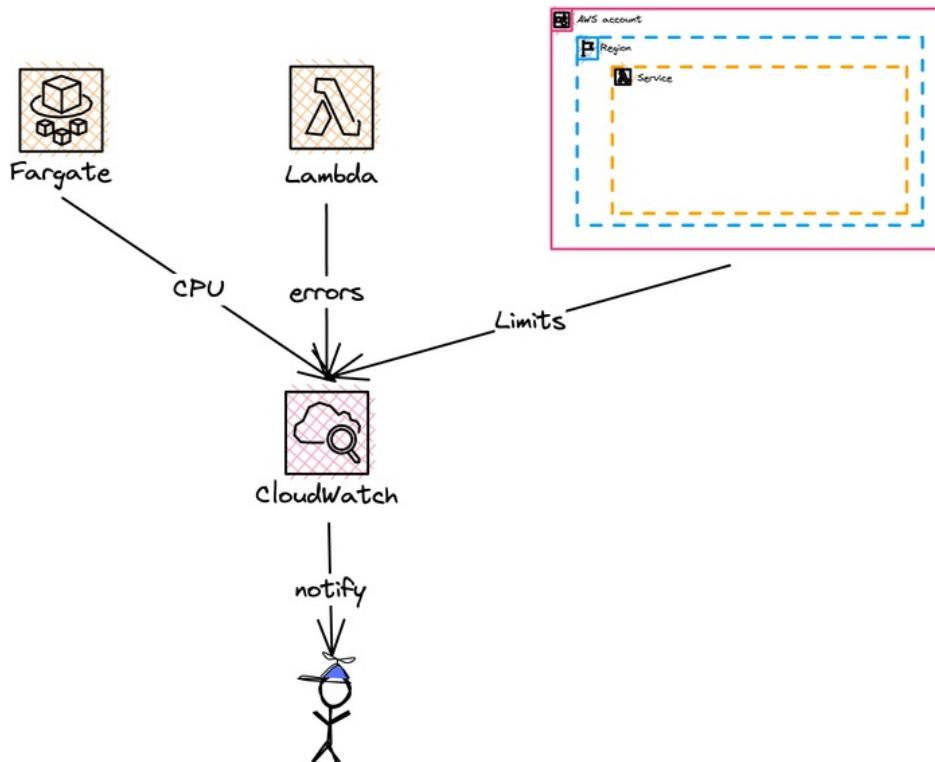
### The Different Alarm States

There are two alarm states:

- IN\_ALARM: The alarm is active, and you need to do something
- OK: The alarm is not active, and everything is 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, including:

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 avoid 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.

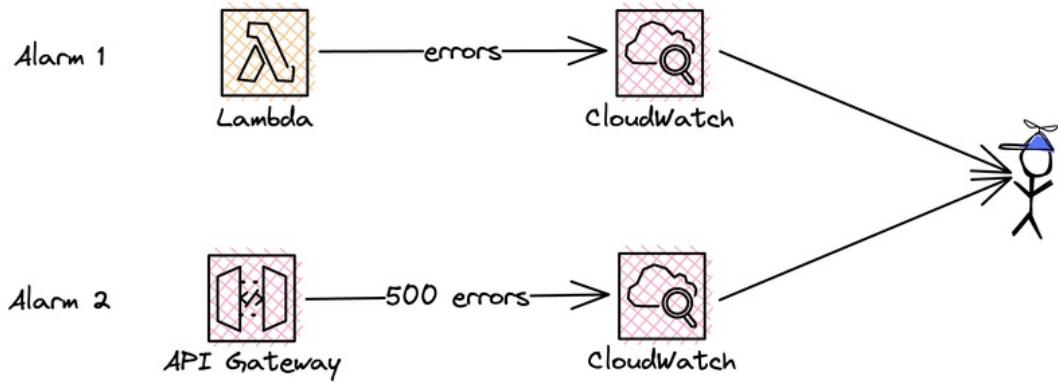
#### **Metric Alarm**

This is an alarm based on one metric.

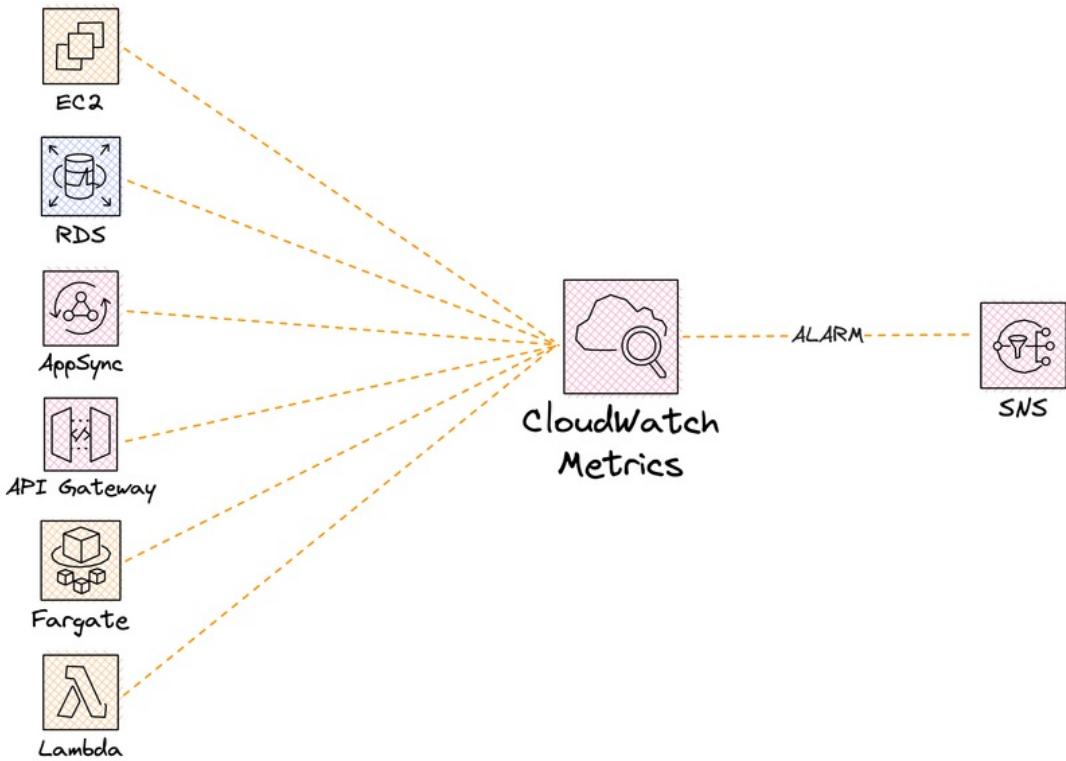


## Composite Alarm

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 `IN_ALARM`

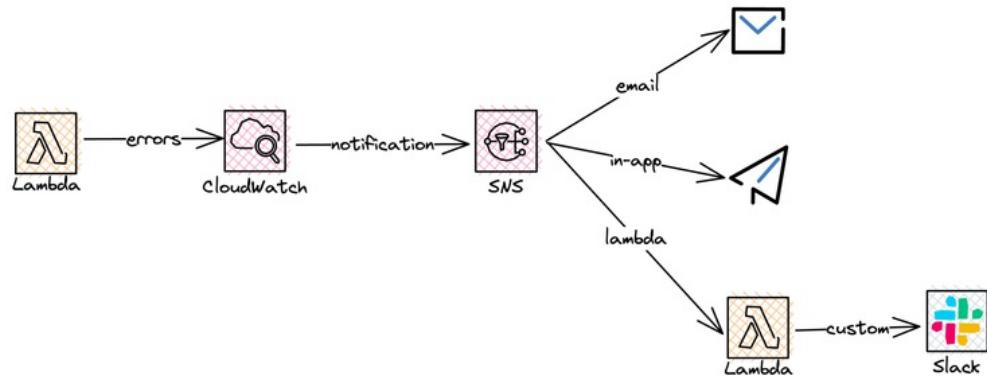


## SNS Topics Inform You in Case of Alarms



The Simple Notification Service (SNS) is 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 excellent integration with the email component

of SNS. You can add your email address and get notified of alarms.



Since SNS is highly flexible, you can also attach a Lambda function to the SNS topic and call any API or service you want in case of alarms. The Lambda function can perform multiple actions to alert you. Some standard use cases include sending notifications via Slack, MS Teams, or Discord. You can also use PagerDuty to stay informed about changes.

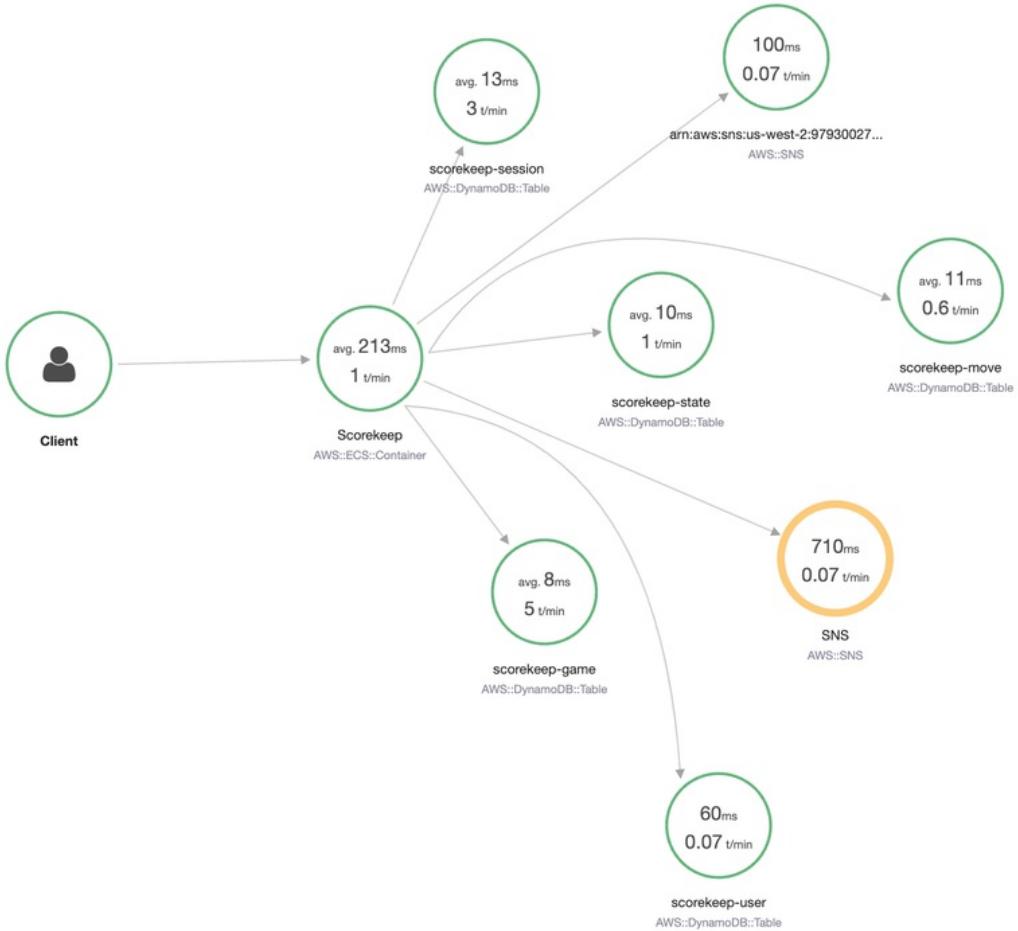
### X-Ray Gives You the Ability to Trace User Requests Throughout Your Distributed System

As you can see throughout the book, there are many AWS services that perform various tasks. Understanding a single user request can become quite challenging.

For instance, if a user reports an issue, you need to understand the entire flow that 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` that you can follow in the system. You can then see how users interacted with your system and check all logs attached to this trace.

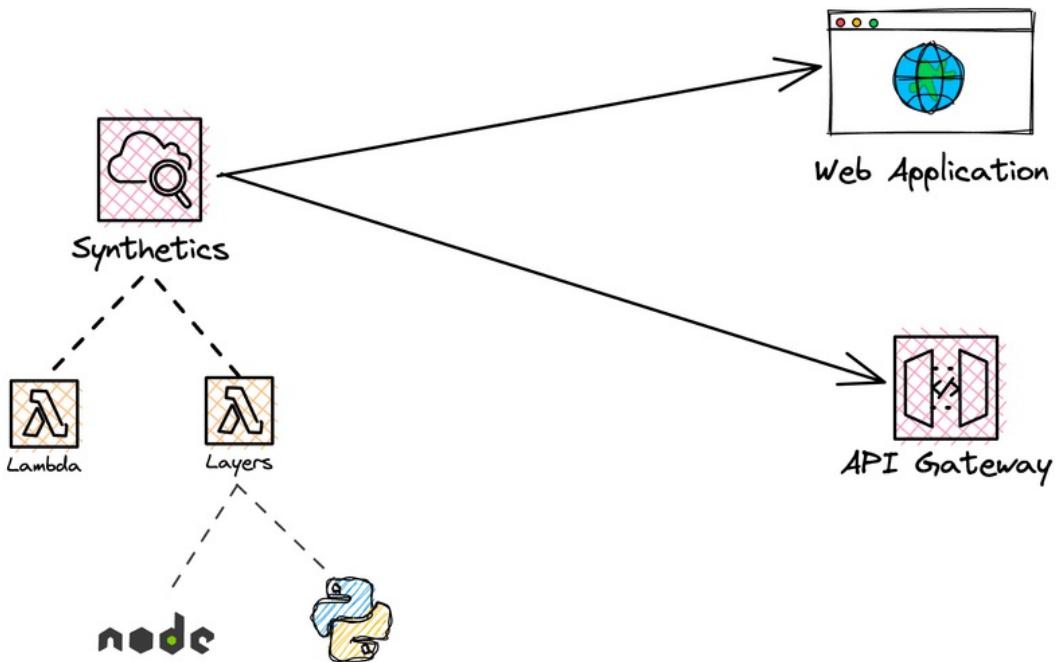
Let's take a look at an example service map that X-Ray built for us:



This example from the AWS Developer Documentation shows how a client calls ECS and how ECS makes several calls to SNS and 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 purely serverless systems, it can be quite challenging to build a complete service map. Some services are not fully supported, such as the integration between SQS and Lambda. Therefore, X-Ray may lose the vision of these services. However, 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 "canaries." Canaries are scripts that run on a schedule to check endpoints and APIs for regular health checks of your web applications. The synthetics service falls into the category of **Application Monitoring** and supports using programmatic web browsers like Puppeteer.

A canary can be of the following types:

- **Heartbeat monitor** - checks URLs regularly
- **API Canary** - checks API endpoints
- **Broken Link Checker** - checks for broken links
- **Visual Monitoring** - opens a webpage in the browser and checks elements on that page like buttons
- **Canary recorder** - records sessions in your browser and replaces 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-fledged test suite cost-effectively in AWS.

## CloudWatch is Priced Based on Ingestion and Storage

CloudWatch can be quite expensive. We mentioned earlier that CloudWatch is one of the most underrated services. This is not only true when learning about this service, but also when estimating application costs.

Ingesting logs into CloudWatch can become really expensive. Let's see how much logs cost:

What	How much?
Ingestion Data	\$0.50 / GB
Storage	\$0.03 / GB
Analysis	\$0.005 / GB scanned

The most expensive point here is **ingesting data**. Storing data can be reduced by setting up a log retention policy. However, 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 afterward.

## Standard Metrics are Free - Custom Metrics are Not

Metrics are different. Default metrics of services like EC2, Lambda, etc. are stored automatically at a standard granularity (1 minute). You only pay for these metrics if you require a more granular view (1 second).

Custom metrics are **more expensive**.

Tier	Cost per metric/month
0 - 10,000	\$0.30
Next 240,000	\$0.10
Next 750,000	\$0.05
> 1,000,000	\$0.02

If you have any application with high traffic, make sure to understand the pricing first. There are really good examples on the AWS CloudWatch Pricing page that show 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.

## Following Logs Interactively via Live Tail

In June 2023, Amazon CloudWatch Logs introduced Live Tail, an innovative feature that offers users an interactive analytics experience. Live Tail allows you to monitor and analyze your CloudWatch logs data in real time, providing you with up-to-the-minute insights.

In order to initiate a Live Tail session, besides the `logs:Get` you also need the new permissions `logs:StartLiveTail` and `logs:StopLiveTail`.

### Actively Following Multiple Log Groups in Real-Time

In the CloudWatch management console, you have multiple ways of starting a Live Tail.

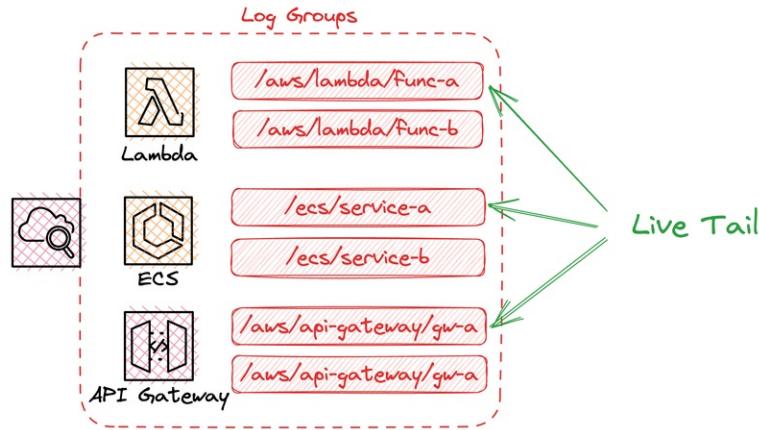
1. Via `Logs > Live Tail` and by selecting the log groups you want to receive events from.
2. Via `Logs > Log groups` where you can also select one or multiple groups.
3. Via `Logs > Log Insights`, also enabling you to choose your desired log groups.

All of the methods above allow you to select `Start tailing` which also allows you to optionally specify log streams or add filter patterns.

The screenshot shows the AWS CloudWatch Live Tail interface. At the top, there's a navigation bar with 'CloudWatch > Log groups > Live Tail'. Below that is a toolbar with 'Filter', 'Actions', 'Clear', 'Cancel', and 'Start' buttons. A status bar indicates '0 events/sec, 100% displayed' and the current time '00:00:34'. On the left, there's a 'Filter' sidebar with sections for 'Select log groups' (containing a dropdown menu with '/aws/lambda/pp-fra-auth-lambda'), 'Select log streams - optional' (with a dropdown menu), and 'Add filter patterns (Case sensitive) - optional' (with a dropdown menu). The main area displays a list of log messages. Each message includes a timestamp, a detailed log entry, and a 'Log stream' column with a link icon. The log entries show various requests and reports from different log streams.

Timestamp (Local)	Message	Log stream
2023-08-25T07:51:26.103+02:00	START RequestId: 9a328b50-0014-4c65-b484-241...	Link
2023-08-25T07:51:26.104+02:00	{"@fields.method":"GET","@fields.url":"/auth...	Link
2023-08-25T07:51:26.106+02:00	END RequestId: 9a328b50-0014-4c65-b484-24179...	Link
2023-08-25T07:51:26.106+02:00	REPORT RequestId: 9a328b50-0014-4c65-b484-24...	Link
2023-08-25T07:51:27.038+02:00	START RequestId: ff1c002f-0c53-4a22-a281-868...	Link
2023-08-25T07:51:27.039+02:00	{"@fields.method":"GET","@fields.url":"/auth...	Link
2023-08-25T07:51:27.041+02:00	END RequestId: ff1c002f-0c53-4a22-a281-868f8...	Link
2023-08-25T07:51:27.041+02:00	REPORT RequestId: ff1c002f-0c53-4a22-a281-86...	Link
2023-08-25T07:51:32.954+02:00	START RequestId: bc7efb1a-9e23-48f4-9310-a57...	Link
2023-08-25T07:51:32.956+02:00	{"@fields.method":"GET","@fields.url":"/auth...	Link
2023-08-25T07:51:32.958+02:00	END RequestId: bc7efb1a-9e23-48f4-9310-a57b9...	Link
2023-08-25T07:51:32.958+02:00	REPORT RequestId: bc7efb1a-9e23-48f4-9310-a5...	Link
2023-08-25T07:51:40.657+02:00	START RequestId: c48b3c98-e2f9-4e0a-ae38-47e...	Link
2023-08-25T07:51:40.659+02:00	{"@fields.method":"GET","@fields.url":"/auth...	Link

As you can watch multiple groups simultaneously, this immensely helps in debugging sessions.



You no longer need to jump between groups and streams but can stick to a single window.

### **Bringing Live Trail Capabilities to Your Terminal with CloudWatch Logs for Humans**

CloudWatch's Live Trail feature is a great improvement, but the overall management console experience still feels sluggish at most points. And surely you don't want to jump into the management console for every small log you want to check.

Luckily, there is **Jorge Bastida's AWS CloudWatch logs for Humans** which enables you to stream your logs from CloudWatch to your local terminal.

### **Finding Your Log Groups via the Groups Command**

Finding the specific log group I need can be quite challenging due to the multitude of functions, stages, and environments.

However, with the help of `awslogs groups`, you can conveniently list and filter log groups based on specific requirements. This allows you to easily locate the log group I am looking for amidst the complexity of my setup.

```
# Listing log groups for eu-central-1
awslogs groups --aws-region=eu-central-1

# Showing log groups that contain 'myfunction' in their name
awslogs groups | grep 'myfunction'
```

### **Streaming Logs to Your Terminal**

The process of browsing logs in the AWS Console can be quite tedious. You are limited to loading a specific number of logs at a time, and each time you scroll to the end, the next batch

is loaded. This can be particularly frustrating if you have a large volume of logs to go through. However, with the introduction of `awslogs get`, this task becomes much more enjoyable and efficient.

```
# Displaying the latest logs of a specific Lambda function
awslogs get /aws/lambda/myfunction

# Starting a continuous stream of our logs to our terminal
# New logs will be shown after they are received by CloudWatch
awslogs get /aws/lambda/myfunction --watch

# Receiving the last 15 minutes of our log stream
awslogs get /aws/lambda/myfunction --start='15m ago'

# Querying logs between 22/08/2023 07:00 and 14:00
awslogs get /aws/lambda/myfunction \
    --start='22/08/2023 07:00' \
    --end='22/08/2023 14:00'
```

## Filtering Logs via Patterns and Queries

You can also apply filters to your logs using the "filter\_pattern" parameter and selectively display specific fields when logging JSON using the "query" parameter.

```
# Messages where the JSON field "upstream_status" is set to "502"
awslogs get /aws/lambda/myfunction --filter-pattern='{ $.upstream_status =
502 }'

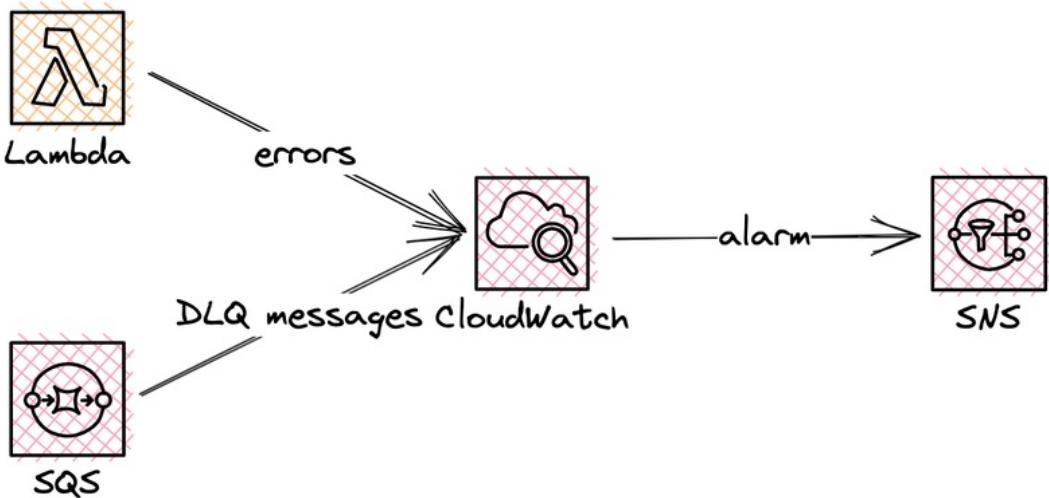
# Only showing the field 'logMessage' of the JSON log
awslogs get /aws/lambda/myfunction --query=logMessage
```

## Use Cases for CloudWatch

CloudWatch use cases provide a range of capabilities for monitoring, alerting, and debugging. Let's take a look at 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.

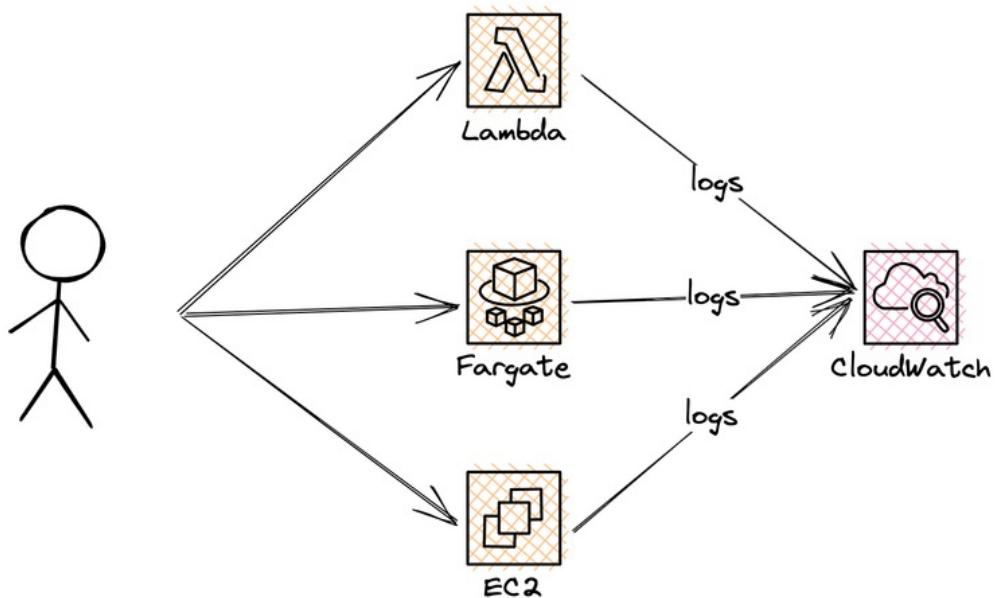


For example, you can create a CloudWatch alarm based on Lambda errors and messages available in a dead letter queue. Once this alarm is triggered, an SNS topic will be called. This SNS topic informs the developer or operations team of ongoing errors.

### **Use Case 2: Debugging and Tracing Logs**

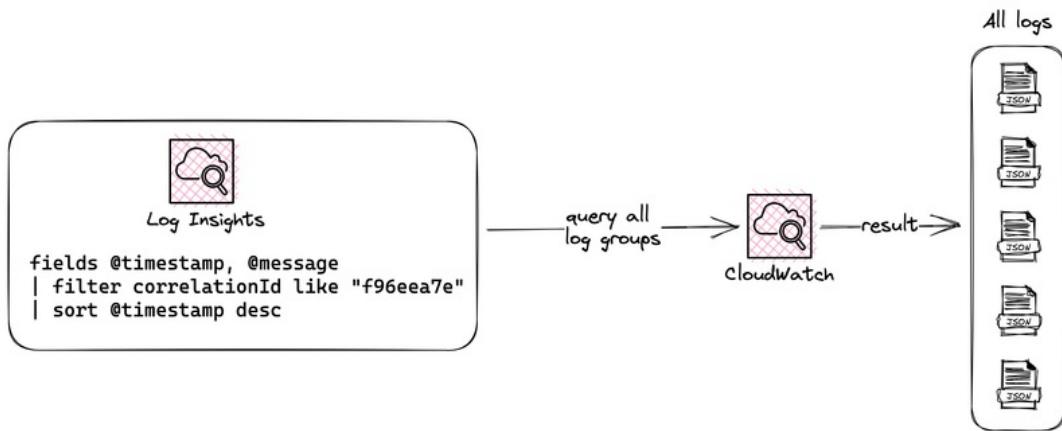
Another useful use case for CloudWatch is debugging and tracing logs across different services. Cloud environments can become quite complex with a variety of services and machines involved.

Users typically do not use just one service, but several. Debugging a whole user session can be quite challenging.



CloudWatch collects log data from all different sources and persists it in a structured way. This data can be very useful when trying 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 logs. You can query across multiple log groups, resulting in several log statements that match the query. This 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 resource scaling based on metrics such as CPU utilization. However, any available metric can be used. Once the metric reaches a certain level, resources such as ECS or EC2 instances can be scaled up.



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:

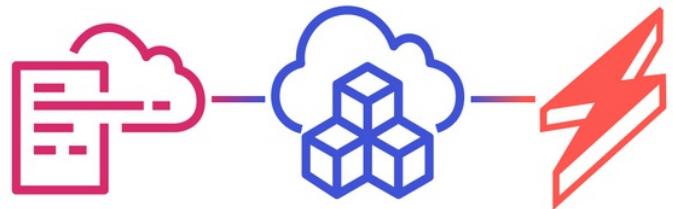
- Use a **structured logger**. Structured logging allows you to trace and query your logs much more efficiently. Use Lambda Powertools if they are available for your programming language.
- Create **alarms** and use **best practice alarms**. There are typical alarms you should create for every service. 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 for building applications.

This second part is all about Infrastructure as Code (IaC). There are almost no professional Cloud Developers who don't use IaC tools. That's why we cover this topic in part two.

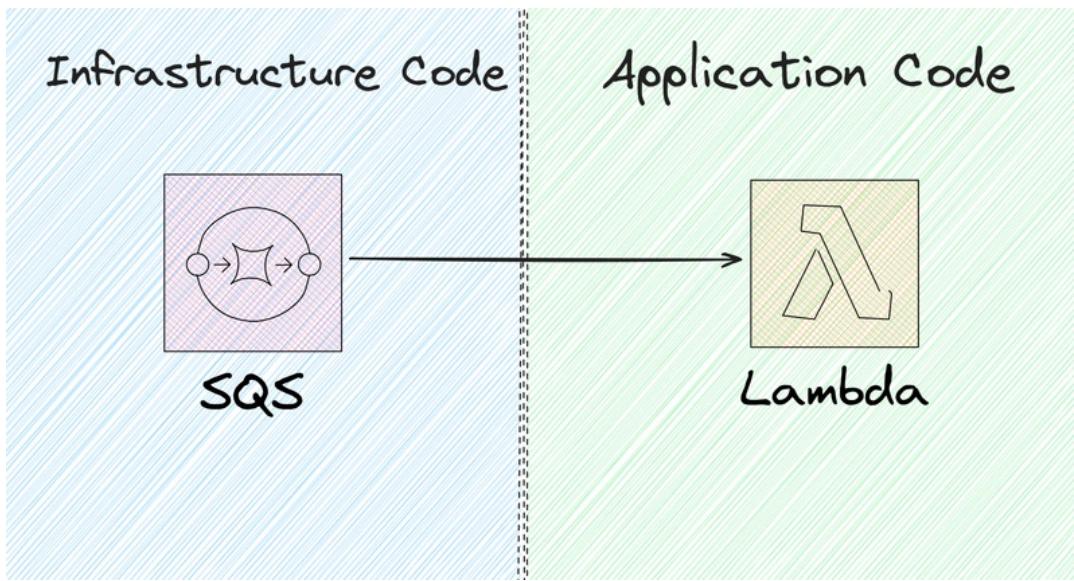
We will cover the basics of IaC and introduce some of the most common tools. However, we won't do a deep dive on IaC. This book is still about the basics of AWS.

The goal of this part is to teach you the fundamentals that you need to get started. We will introduce the three most common tools in the industry:

1. CloudFormation
2. Serverless Framework
3. Cloud Development Kit (CDK)

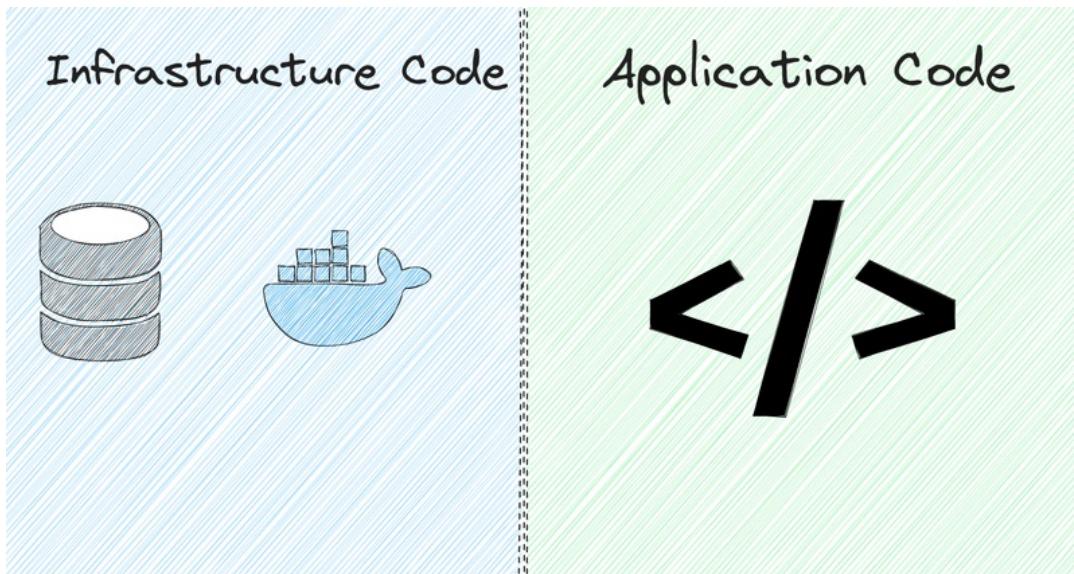
## **Application Code is Infrastructure Code**

One difference that comes with cloud computing is the distinction between application and infrastructure code. This line can become blurry in a cloud-native world.



Let's take SQS and Lambda as an example. We cannot distinguish one service as only infrastructure or only application code. AWS manages the underlying infrastructure for both services, and 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 between infrastructure and applications 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 gain 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, such as automated testing, continuous deployment, and feature flagging. All of these practices are possible.
- You get a **reproducible** setup. There are no manual checklists or steps to remember. Executing the code is sufficient.

There are many more benefits. You cannot use AWS professionally without being able to provision resources via Code.

## History of IAC - From Manual Provisioning to Componentized Solutions

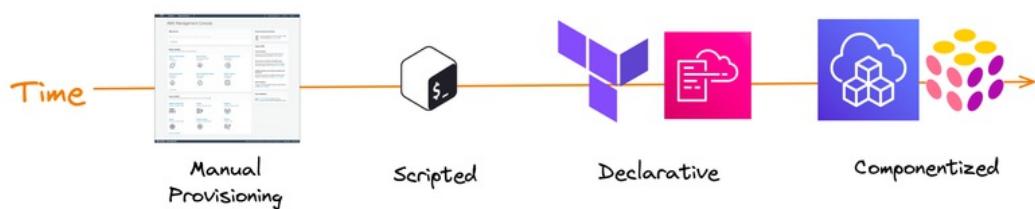
"If you want to understand today, you have to search yesterday."

**Pearl S. Buck, American novelist (1892-1973)**

There are many IaC tools available, and each has its place. 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 has only been evolving for the last two decades, and software development has undergone many changes during that time.

The image below shows some of the main categories of different IaC tools.



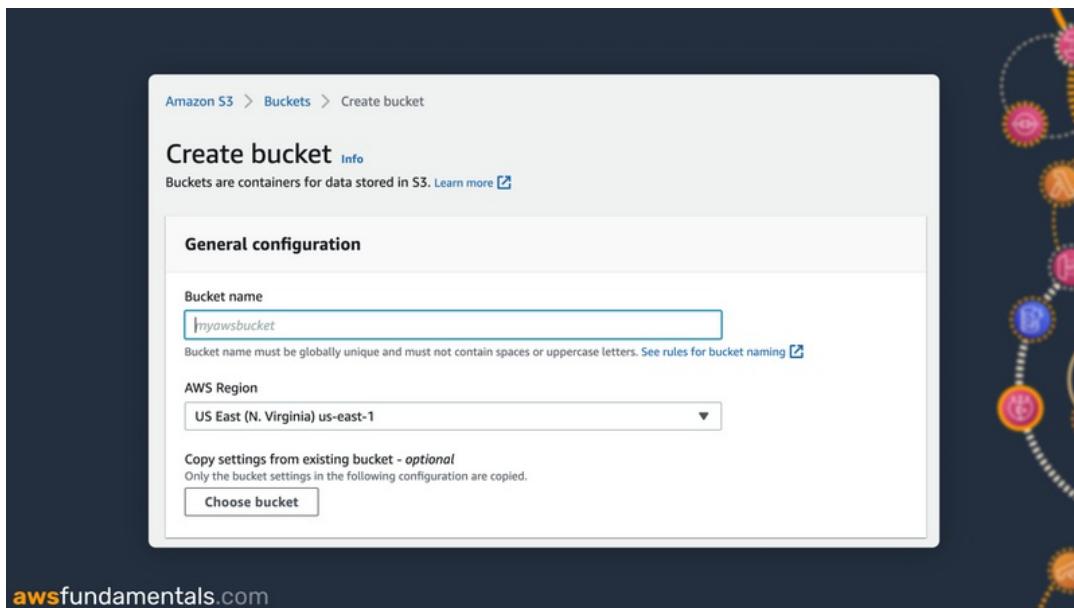
We started with a manual approach (clicking in the console), and now we are at a point where

we can use our favorite programming language with frameworks like CDK.

Let's take a closer look at each category.

### 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, history, or way to govern your architecture. Working together meant creating checklists and screenshots. Only with those, it was possible to reproduce the same setup.

After AWS launched many more services, this approach was no longer workable. Launching new regions or development accounts became almost impossible.

### Scripted - Provision AWS Resources with the CLI

The second step in the history was creating scripts using the AWS CLI. The CLI is one of the main interfaces to the AWS API. For example, you can create an S3 bucket with the CLI like this:

```
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 logic for existing infrastructure. This was not a good approach.

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 this:

```
{
  "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. 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 necessary to understand the basics of CloudFormation.

These two are currently some 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 that software developers do 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, CloudFormation is still often used.

Popular frameworks in this 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 still work.

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 specify 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 this:

```
new s3.Bucket(this, 'MyFirstBucket', {});
```

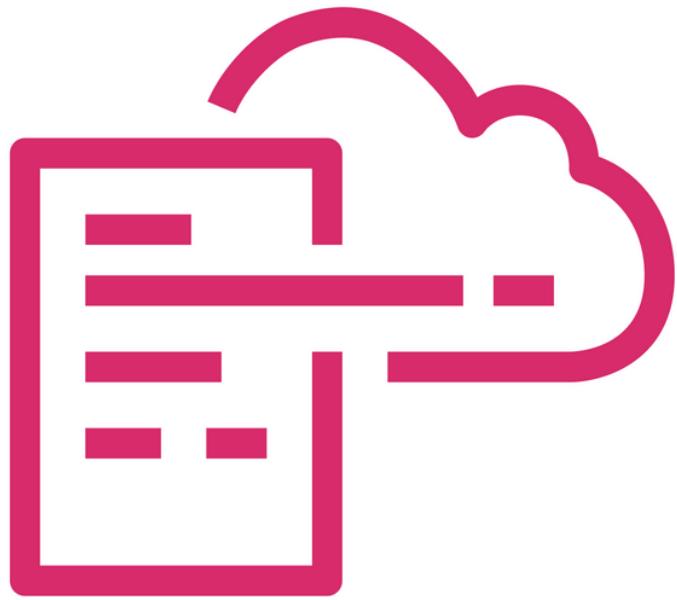
## Benefits

- 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 history, let's dive into the first set of tools.



## AWS **CloudFormation**

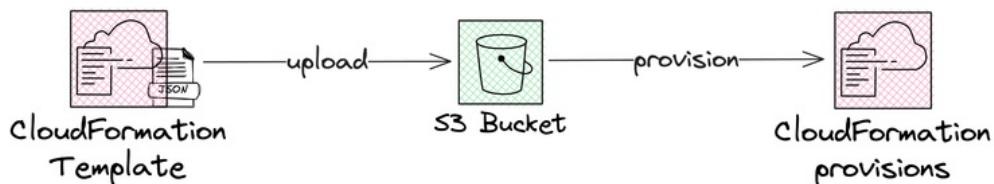
# CloudFormation Is the Underlying Service for Provisioning Your Infrastructure

## 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 written in either `JSON` or `YAML` language. CloudFormation uses these templates to provision your infrastructure, taking care of creating or updating your infrastructure as you have defined it.

CloudFormation follows a declarative approach, meaning that you only need to tell it which resource you want. It will find ways of creating the services and infrastructure, so you don't need to define how to create it; you only need to define the final state you want to have.



You can pass the templates to CloudFormation via an S3 bucket or upload them directly to the service. You will see an overview of all the services that CloudFormation creates. After that, it starts provisioning them.

The great thing about CloudFormation is that infrastructure management becomes much easier. Let's say you have an 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 no longer need it, you can remove all services with one click on CloudFormation.

It is also possible to replicate the entire 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 `awsfundame1stestbucket-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.

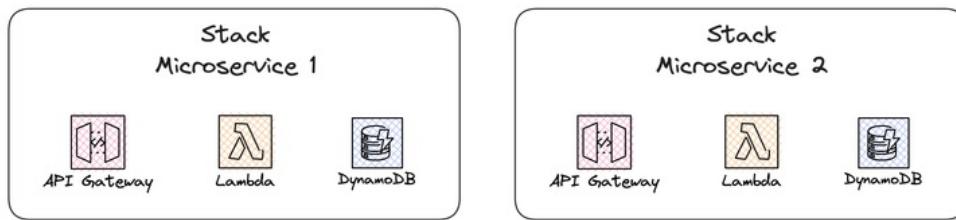
```
{
  "AWSTemplateFormatVersion": "2010-09-09",
  "Description": "Sample CloudFormation Template",
  "Resources": {
    "s3Bucket": {
      "Type": "AWS::S3::Bucket",
      "Properties": {
        "BucketName": "awsfundamentalstestbucket-xyz"
      }
    }
  },
  "Outputs": {
    "BucketArn": {
      "Value": {
        "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.

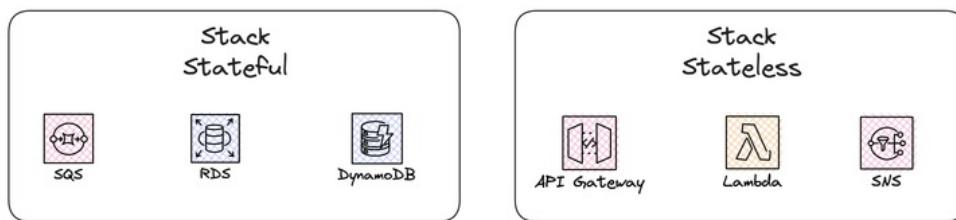
There are different approaches to designing stacks. A common approach is to group one **microservice** into one stack. One microservice could contain the following services:

1. Database
2. API Gateway
3. Lambda Function
4. Frontend



Another approach is to distinguish between **stateful** and **stateless** resources.

1. **Stateful** - DynamoDB, RDS, and SQS.
2. **Stateless** - API Gateway, Lambda, and SNS.



It all depends on your architecture, but these are common approaches.

You can perform several actions with stacks, such as:

- **Deleting** the entire stack.
- Viewing 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 **Create stack** button. We will create a stack together later on.

### **Change Sets Only Deploy Changes to Your Current Infrastructure**

Change Sets are responsible for modifying already deployed infrastructure. CloudFormation detects your changes and only applies the necessary changes to your infrastructure. With a change set, you can see the exact changes and how they would impact your infrastructure.

A good example is RDS. If you want to enable backups for RDS, you don't need to delete your entire database. With CloudFormation, you can create change sets to update only your current infrastructure. Modern frameworks like CDK are doing this, but CloudFormation gives them the power to do it. With this, you can enable backups without losing any of 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": "awsfundamentals-testbucket-xyz"
      }
    }
  },
  "Outputs": {
    "BucketArn": {
      "Value": {
        "Ref": "s3Bucket"
      }
    }
  }
}
```

```
        "Value": {
            "Fn::GetAtt": ["s3Bucket", "Arn"]
        }
    }
}
```

This is the template we saw earlier. This template creates an S3 bucket with the name `awsfundamentalstestbucket-xyz`.

## Template Version

`AWSTemplateFormatVersion` 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 can 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.

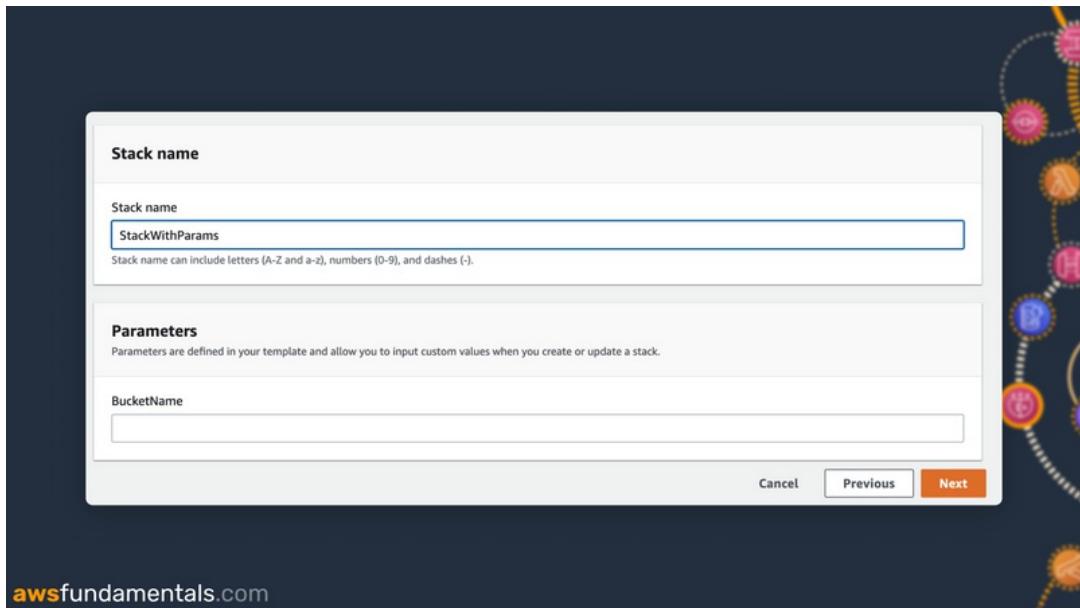
For example, if you deploy infrastructure to different stages like development and production, you often want to change some properties for these stages. In the development stage, you want to name your resources with the prefix `Dev-`. In the production stage, you want to add the prefix `Prod-`. Or you 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.

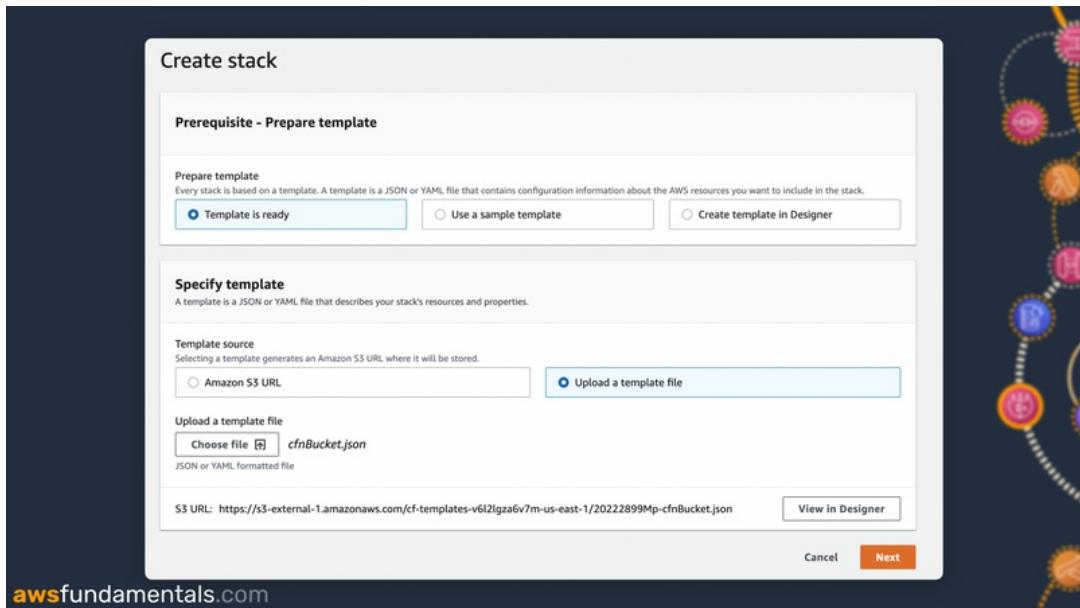


## Let's Deploy

Let's deploy our example template now. We'll show you how to deploy the template using both the web console and the CLI. Let's start with the console.

### Management Console

Go to the CloudFormation service 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 (which is really good for getting some inspiration).
3. **Create a template in Designer** - AWS has a visual designer for creating templates.

We chose option one and uploaded the template directly.

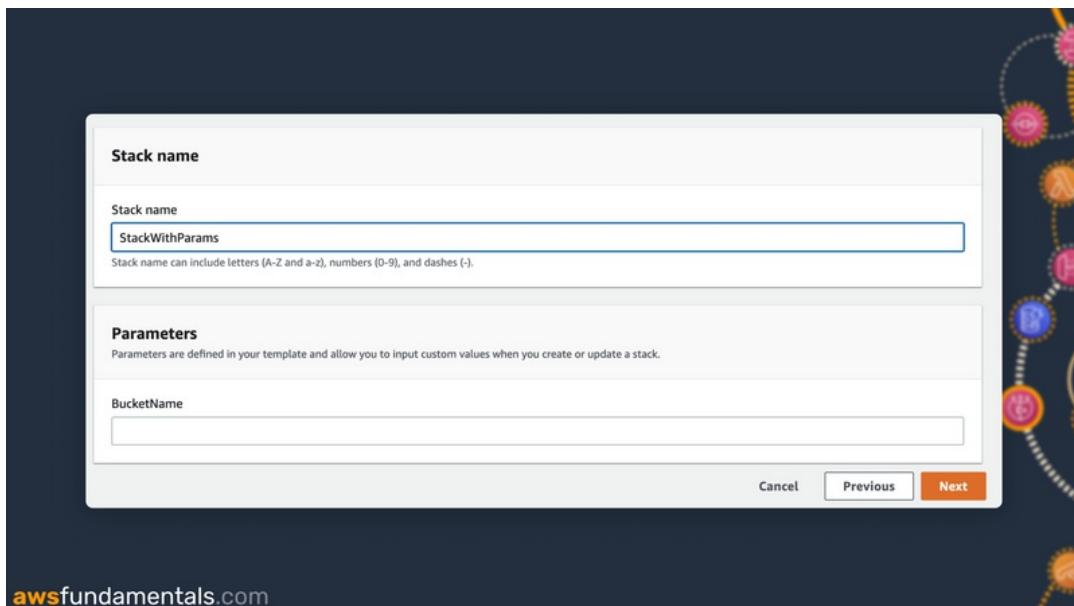
This is the template we will use:

```
{  
    "AWSTemplateFormatVersion": "2010-09-09",  
    "Description": "Sample CloudFormation Template",  
  
    "Parameters": {  
        "BucketName": {  
            "Type": "String"  
        }  
    },  
  
    "Resources": {  
        "s3Bucket": {  
            "Type": "AWS::S3::Bucket",  
            "Properties": {  
                "BucketName": {  
                    "Ref": "BucketName"  
                }  
            }  
        }  
    },  
  
    "Outputs": {  
        "BucketArn": {  
            "Value": {  
                "Fn::GetAtt": ["s3Bucket", "Arn"]  
            }  
        }  
    }  
}
```

```
    }  
}
```

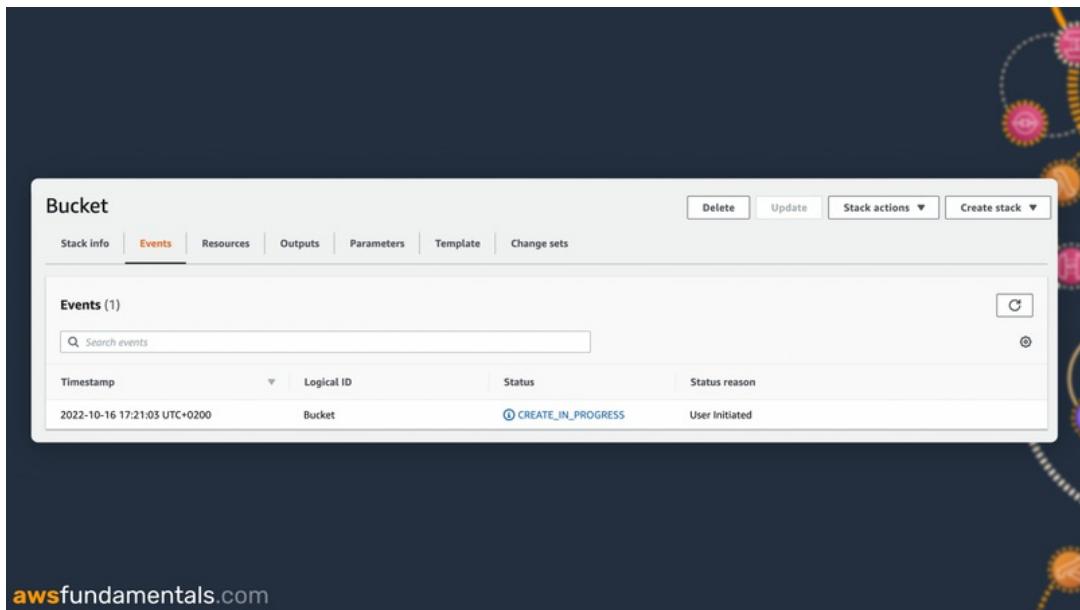
Please copy that, save it as a JSON file, and upload it here.

After you click on next, you can upload your template. On the next screen, you can enter your defined parameter. For this one, we defined the use of a dynamic bucket name. Enter whatever name you want. 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 enter the bucket name, click on Next. You can add more metadata and IAM permissions in the next screens, but we will skip that and click on **Create Stack**.



Now you can see an overview of your stack. When working 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, such as:

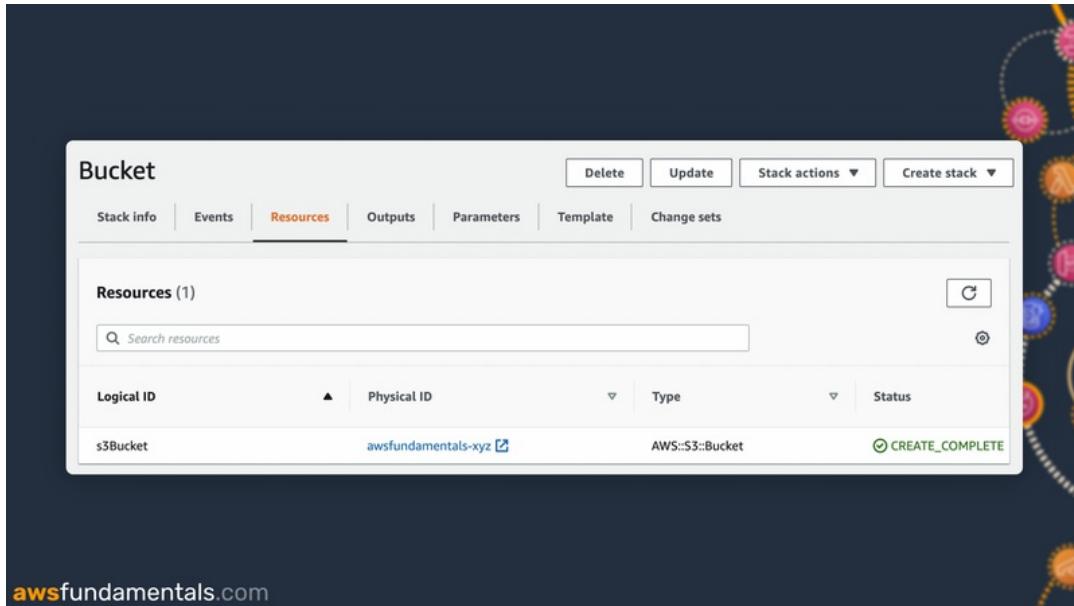
- CREATE\_IN\_PROGRESS
- CREATE\_COMPLETE
- CREATE\_FAILED
- DELETE\_COMPLETED
- DELETE\_FAILED
- DELETE\_IN\_PROGRESS
- ROLLBACK\_COMPLETE

I think the status codes are self-explanatory.

- **Resources** - here you can see all the available resources. In our stack, we will only see one resource of the type AWS::S3::Bucket.
- **Outputs** - these are the outputs we defined earlier. We can 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 open.

Congratulations, you have deployed your first IAC resources.

## Deploy with AWS CLI

Let's deploy the template using AWS CLI. We saved the file in the current folder as `cfn.json`, but you can save it wherever you want. Just make sure to update the path here.

Please ensure that you have configured your named profiles correctly on your CLI and that you have permission to execute the command.

We will execute the following command and explain what it does in detail.

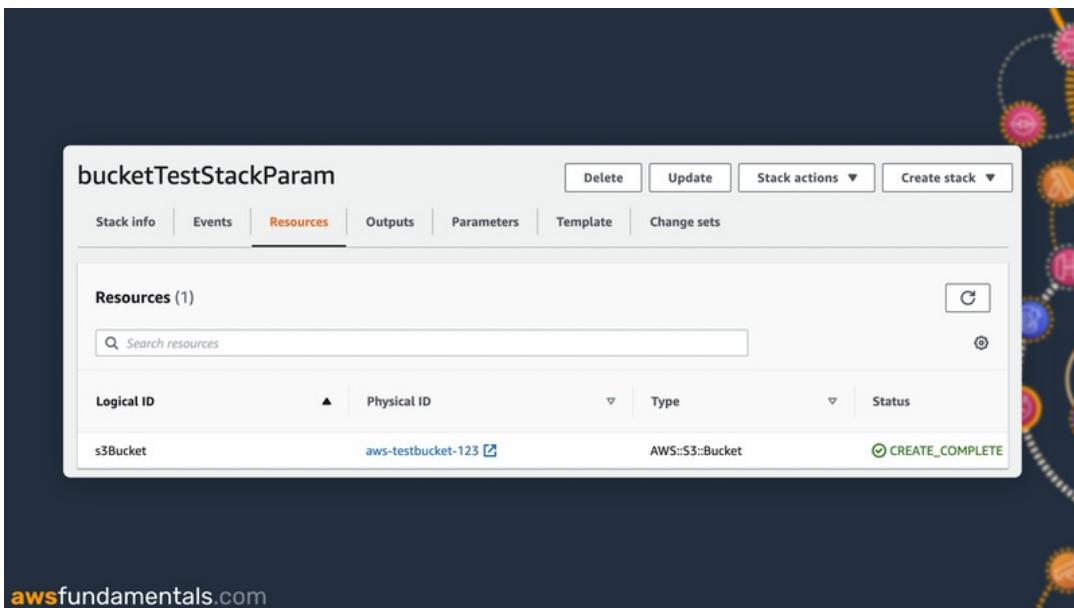
```
aws cloudformation create-stack \
--stack-name bucketTestStackParam \
--template-body file://cfn.json \
--parameters ParameterKey=BucketName,ParameterValue=Aws-testbucket-123
```

```
aws cloudformation create-stack
```

This is the command you need to execute. The rest are parameters that we pass to the `create-stack` command.

- `-stack-name bucketTestStackParam`: This is the name of the stack.
- `-template-body file://cfn.json`: This is the path to your file.
- `-parameters`: These are the parameters you defined, in this 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.

## Building Advanced and Large Eco-Systems with Nested Stacks

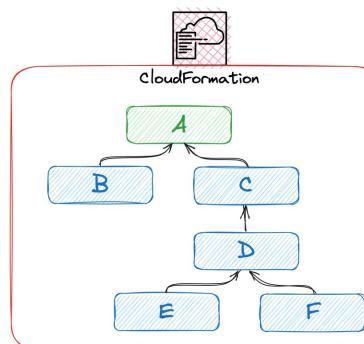
Building large applications quickly results in having hundreds, even thousands of cloud resources that work together. Having all of them in a single CloudFormation stack can become very messy, hard to read and extend, and even dangerous as updates potentially affect your whole ecosystem. As your infrastructure grows, you may find yourself repeating the same components in multiple templates. To avoid duplication and improve maintainability, you can extract these common components into separate templates and reference them as **nested stacks** in your main template.

Nested stacks are a powerful feature in AWS CloudFormation that allows you to create stacks within other stacks. By using the `AWS::CloudFormation::Stack` resource, you can easily create a nested stack within an existing stack.

For instance, let's say you have a configuration that is used across multiple stacks. A common example may be an Application Load Balancer, a Security Group, a Network Access Control and Target Group configuration, and an Elastic Container Service setup that runs your application from a Docker image on Fargate.

Instead of duplicating the configuration in each template, you can create a dedicated template specifically for this distinct part of your infrastructure. Then, you can reference this template as a nested stack in your other templates.

Nested stacks can also be nested within other nested stacks, creating a hierarchical structure of stacks. In this hierarchy, the root stack serves as the top-level stack that encompasses all the nested stacks. Each nested stack has an immediate parent stack, which can be another nested stack or the root stack itself.



In this example, stack A serves as the root stack, while stacks B and C are immediate children of the root stack. This means that stack A is both the root and parent of stacks B and C. The example also nests stacks further down the hierarchy, so that stack D, a child of stack C, also

has its own children.

By leveraging nested stacks, you can organize and manage your infrastructure more efficiently, promoting reusability and reducing duplication.

## Final Words

CloudFormation is **the** basic IaC service. It is often more common to use a framework on top of CloudFormation, such as CDK, Serverless Framework, SAM, Serverless Stack, or Amplify.

These are all valid choices, and most of them use CloudFormation as the provisioning engine.

Terraform and Pulumi are two of the main exceptions 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 that 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.



AWS **CDK**

# Using Your Favorite Programming Language with CDK to Build Cloud Apps

## Introduction

Now we will look at the last IaC tool, which 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** you use. 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 the 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: 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 the Integrated Development Environment. For example, VS Code, Webstorm, or IntelliJ.

Let's see some benefits in more detail.

## The IDE Improves the Developer Experience

CDK offers a great improvement in the 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 are familiar with their programming languages inside and out, so there are no new syntax or primitives to learn. This makes it much easier to get started, as the hurdle of learning the syntax of a tool like Terraform, for example, is completely gone.

### You can use your IDE

Developers use and 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 within your company or even with the whole world (see [constructs.dev](#)).

This is often an argument against CDK because you can abstract too much. While this is true, it 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 them. You can 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 can 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 (Mostly) 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. More on that later.

This is an important step in deciding which tool to use. You don't want to use any IaC tool that won't support new properties or services quickly. Of course, there are examples where AWS takes a long time to support new constructs (e.g. Distributed Maps for Step Functions). But in general, the support and development have been very good so far.

## **CDK Uses CloudFormation**

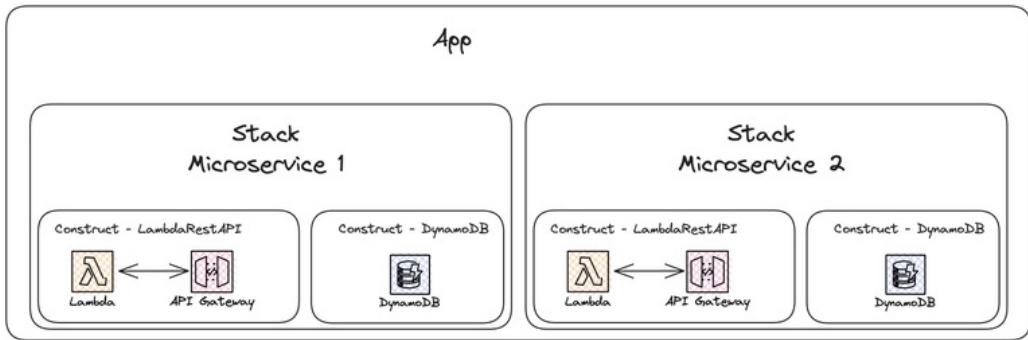
In the background, all CloudFormation concepts are still present, including Stacks, Change Sets, and everything related to them. This can be both good and bad, but it means you don't need to learn another provisioning methodology. Instead, you can build on top of CloudFormation.

The main concepts of having a stack or a change set still apply. That's why it's 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, an 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 blocks 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 a Level 1 Construct. Level 1 constructs always start with Cfn, like the CfnBucket.
<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 Bucket 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

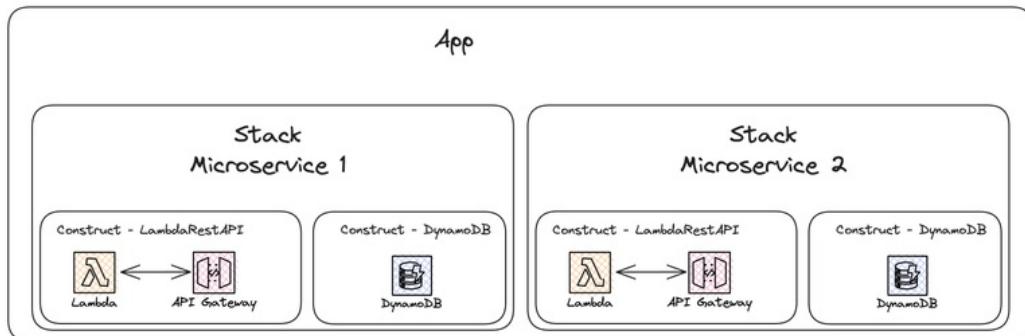
```
new CloudFrontToS3(this, 'cloudfront-s3', {});
```

Level 3 constructs are more like patterns than simple resources. The `CloudFrontToS3` construct, for example, connects a CloudFront distribution (CDN) 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 is already a pattern and could be seen as a Level 3 construct. However, AWS also offers this as its own construct in `LambdaRestAPI`.

## Stacks - Deployable Units

Stacks bundle multiple Constructs together 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 it.

### Model Your Stacks Either by Services or Stateless Vs. Stateful

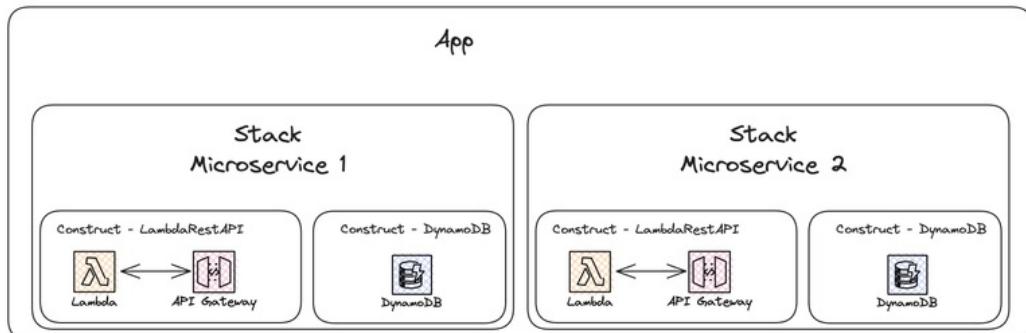
Modeling your stacks is not easy. It is not always straightforward what belongs together and what does not. We showed you some examples in the CloudFormation chapter already.

The general recommendation from 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.

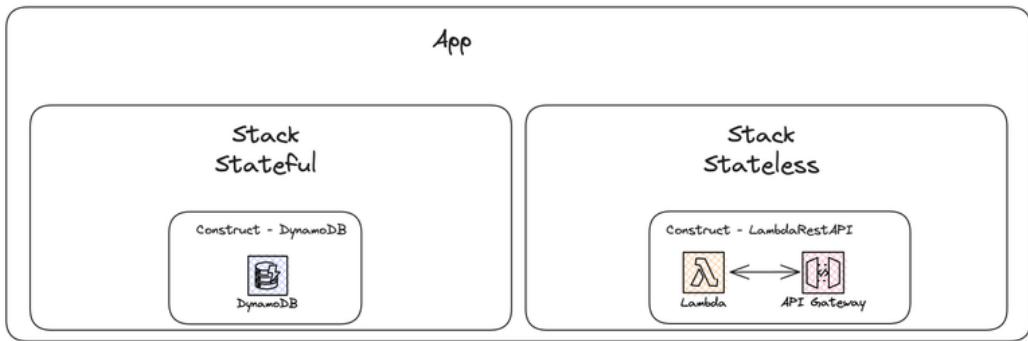
You still need to model your stacks.

### Stack per Microservice



One approach to modeling your stacks is by using **microservices**. Each microservice represents one stack. The example image above shows one microservice with an API, a Lambda function, and a database.

### Stack per Stateful vs. Stateless



Another approach to modeling your stacks is by using the distinction between stateful and stateless. One stack would be for stateful (DynamoDB) resources, and another one for stateless (Lambda, API Gateway) resources. I hope you can see the similarity to the CloudFormation chapter. This is why it is very useful to learn the basics of CloudFormation for using CDK.

You can determine if a stack is stateful or stateless by thinking about whether you can delete and redeploy a stack without any impact on your users. If this is the case, you have 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 it, or you can find yourself sometimes in bad CloudFormation states. Keep an especially close eye out for circular dependencies. Find your resources that belong together and put them into one stack. Model the

rest with constructs.

## Apps - Bundling 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: constructing, validating, synthesizing, and deploying your infrastructure. The details here are not important to get started with the CDK.

## CDK CLI - Initialize, Synthesize, and Deploy Your App

CDK comes with its own CLI. Typical tasks for the CLI include synthesizing, deploying, removing stacks, and much more. Let's go through some of the most important commands.

### `cdk init`

Init allows you to 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 can 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 deploys 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 we 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 that we want to highlight is how you can grant permissions to roles or users in CDK. As we know, AWS follows the principle of **least privilege**. That means your roles should grant **the fewest privileges 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 that 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 this:

```
{
    "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 ever work on a larger codebase, 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 alarms automatically for you.

There are many constructs like that out there. Make sure to check them out.

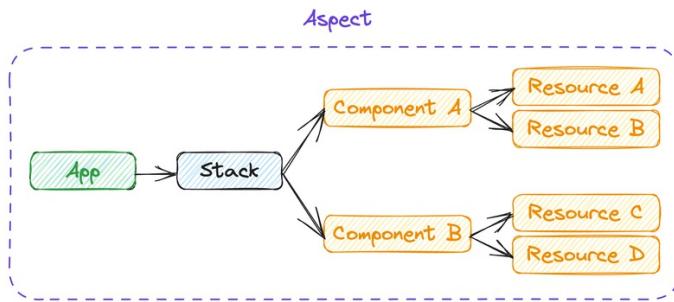
## CDK Aspects

Aspects provide a way to execute an operation on every element within a specific scope.

These operations can involve modifying the elements, such as by appending tags, or verifying certain conditions about their state, such as ensuring that a security group doesn't allow incoming traffic on SSH ports.

## Creating Custom Aspects To Apply Rules Across Your Stack

The Visitor Pattern is used to implement aspects. When a CDK Aspect is applied to a particular scope, it grants access to each child node contained within that scope.



Custom aspects require the implementation of a concise interface. Here are the necessary steps to create your own custom aspects:

```
interface IAspect {  
    visit(node: IConstruct): void;  
}
```

For each node within the scope where the aspect is applied, the `visit` method will be invoked.

## A Simple Example: Applying Tags to your Resources

Here is an example of how a custom `ApplyTags` Aspect could be implemented:

```
class TagAspect implements IAspect {  
    tags: Record<string, string>;  
  
    constructor(tags: Tags) {  
        this.tags = tags;  
    }  
  
    visit(node: IConstruct) {  
        // do nothing if resource doesn't support tags  
        if (!TagManager.isTaggable(node)) return;  
        Object.entries(this.tags).forEach(([key, value]) =>  
            // apply all tags to the node  
            node.tags.setTag(key, value);  
        );  
    }  
}
```

```
    }  
}
```

In the `visit` function, we first determine if the node is taggable by using the `TagManager.isTaggable(node)` method. If it returns true, we continue to add the desired tags.

To implement the `TagAspect` Aspect at the application level, you can follow these steps:

```
const app = new cdk.App();  
const myStack = new MyStack(app, 'MyStack');  
  
const aspects = Aspects.of(app);  
  
aspects.add(new TagAspect({  
    environment: 'develop',  
    managedVia: 'CDK'  
}));
```

With this, we can easily add our two tags to all taggable resources within our stack.

## A Powerful Way of Enforcing Rules Across Your Stack

CDK aspects are a powerful and easy way of enforcing rules across a stack because they provide a declarative and centralized approach to applying cross-cutting concerns to resources.

1. **Centralized enforcement:** Aspects allow you to define and enforce rules in a centralized manner. You can define an aspect once and apply it to multiple resources across your stack, ensuring consistent behavior and adherence to your defined rules.
2. **Declarative approach:** Aspects use a declarative approach, allowing you to define rules using familiar programming constructs. This makes it easier to express complex logic and conditions for applying rules to resources.
3. **Separation of concerns:** Aspects enable you to separate concerns by encapsulating specific rules or behaviors in separate aspects. This promotes modularity and reusability, as aspects can be applied to different stacks or shared across multiple projects.
4. **Flexible and extensible:** Aspects provide flexibility and extensibility by allowing you to define custom rules and behaviors. You can create your own aspects tailored to your specific requirements, enabling you to enforce custom policies or best practices across your

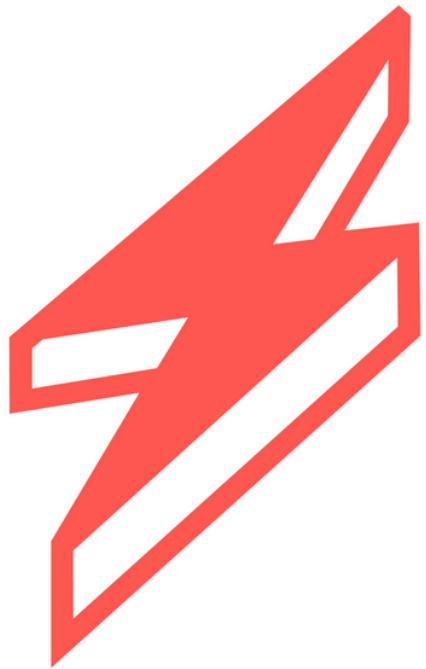
stack.

## 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 necessary infrastructure for a simple service powered by Lambda and exposed to the internet via API Gateway is not a trivial task, even though it's a common use case.

While we can click together things in the AWS console, that's not something we want to do for serious projects because it's not easily reproducible and is very error-prone.

Even when using CloudFormation, a lot of code, resources, and correct references are needed to achieve this goal. What if there were a framework that solely focuses on everything around Lambda?

And that's what the Serverless Framework does, beautifully. It abstracts away much of the boilerplate configuration needed for Lambda, API Gateway, and other native integrations with AWS services. It also handles the packaging for you and comes with a powerful CLI that enables you to easily execute Lambda function code deployments without touching the infrastructure of your stack.

## **Serverless Framework Comes with Different Providers and Formats to Define Your Infrastructure and a Huge Set of Templates**

The entry barrier for the Serverless Framework is as low as it gets. All you need to do to install Serverless Framework either globally or locally via NPM (`npm i -g serverless`) or YARN (`yarn global add serverless`) and you're good to go.

If you have 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. However, 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 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 where you define your functions, events, and all other AWS resources you might want to deploy.

The magic behind Serverless is that 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 has got you covered as you can also define your configuration using:

- JSON (`serverless.json`)
- JavaScript (`serverless.js`)
- TypeScript (`serverless.ts`)

When working extensively with JavaScript/TypeScript, this is quite convenient as IDEs usually have better support for it 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 architectural examples. They are a great way to bootstrap your base application without needing to fiddle with the boilerplate foundations. They also save you from copying and pasting code from tutorials and stumbling through the documentation.

As the configuration semantics are really descriptive, this offers you a great way to explore 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 many constructs that not only help you easily bootstrap 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 is later translated into a CloudFormation template to create your resources.

By running `sls`, you will create a new service configuration.

### Functions - Your Compute Layer Running on AWS Lambda

Functions are your compute layer, running on Lambda. They are independent in both execution and deployment. Often, they serve a single purpose, but they don't have to.

```
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 referred to an externally packaged deployment unit. However, Serverless can also handle packaging your function's code. Instead of using `artifact`, you can use `patterns` to indicate which files should be included in your deployment unit ZIP file.

```
package:
  patterns:
    - '!node_modules/**'
    - 'myfunction.js'
```

Run `sls package` to package your function and create the CloudFormation template, without actually deploying your stack. This helps validate your configuration beforehand by looking into the hidden `.serverless` folder, which now includes 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, for example, HTTP calls through API Gateway or messages via SNS. What would otherwise be a tedious task, Serverless creates the necessary infrastructure to easily attach those events to your functions.

Here's a look at what needs to be created for API Gateway:

- The gateway itself
- Mock responses for CORS
- Gateway resources and 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
```

The Serverless Framework has eliminated all of our hassle.

## Custom Resources Defined in Plain CloudFormation

The Serverless Framework focuses on serverless architectures, which mainly revolve 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 the 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 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.lambdaRuntime}
    layers:
      - { Ref: CommonLambdaLayer }
```

## Deploying Your Infrastructure and Code to AWS

Serverless will deploy your resources via CloudFormation for managed state and other benefits that CloudFormation offers, such as automated rollbacks in case of erroneous deployments.

Using `sls deploy` will always deploy all of your resources, including the code of your functions. Since version 4, Serverless also allows you to use CloudFormation's direct deployments, which have no downsides and speed up deployments. It will become the default in version 4.

```
provider:
  name: aws
  deploymentMethod: direct
```

You can append `--verbose` to receive detailed output about the steps Serverless is taking, as well as to see the outputs of CloudFormation.

If not specified otherwise, Serverless will always default to the `dev` stage and `us-east-1` region. To pass a custom stage name or region via the CLI, use `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 change the code of your functions. Additionally, deploying with CloudFormation can be time-consuming.

Serverless has your back here: you can deploy your functions independently.

```
sls deploy function --function myfunction
```

Serverless will package your deployment unit (or use your referenced unit directly), compare the hash with the currently uploaded function, and deploy the function directly via the AWS CLI if they do not match. This is extremely fast and, depending on the package size and the upload speed of your function, can take only a few seconds.

Especially during the development process, this is tremendously helpful.

### Externalizing Your Dependencies via Layers

Packaging and 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 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 just 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 powerful tool with only the onboard tools being shipped out of the box. However, it also offers many community plugins for anything you can think of, which extend its functionality 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 a proper plugin for almost any use case. There's no need to reinvent the wheel.

## Marrying Other Infrastructure-as-Code Tools, like Terraform

Serverless Framework focuses on Serverless services that revolve around Lambda. However, a fairly large project will often require many other core services as well. While you can manage everything via the custom CloudFormation templates within your configuration file, this quickly becomes burdensome 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 meets 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 is 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, 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]
    }
}
```

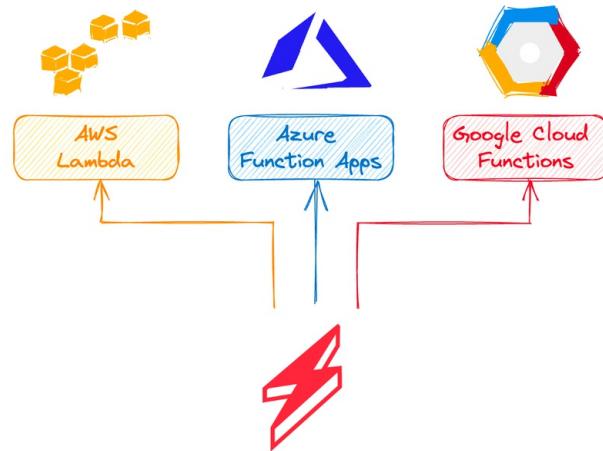
After exporting the ARN to the parameter store with Terraform, we can import it into our Serverless Framework template.

```
custom:
  terraform:
    chatBotAlertTopicArn: ${ssm:slack-alerts-chatbot}
```

The only precondition is that we always need to deploy Terraform first and only depend on Serverless resources for Terraform resources - not the other way around. Otherwise, we would get a dependency cycle that cannot be resolved.

## Using Serverless Framework to Create Multi-Cloud Solutions

As mentioned earlier, Serverless Framework is a tool that abstracts away the underlying cloud provider, allowing you to write serverless applications without being tightly coupled to a specific cloud platform. But it doesn't force you to only use AWS. It supports multiple cloud providers, also including Microsoft Azure, and Google Cloud Platform (GCP).



Additionally, the Serverless Framework allows you to create a setup where you can deploy resources into multiple clouds simultaneously. This can help increase the robustness of your application by leveraging the strengths of different cloud providers and ensuring high availability.

In summary, the Serverless Framework provides flexibility by decoupling your application from a specific cloud provider, enabling you to use it with AWS, Azure, GCP, or even deploy resources across multiple clouds.

## Final Words

The Serverless Framework is here to stay. It does an excellent job of quickly spinning up small or large Lambda-powered applications.

We have been in love with it for several years and expect it to become even more powerful in the future.

# Credits & Acknowledgements

We wanted to take a moment to express our sincerest gratitude for your support in purchasing this book. It means the world to us and we are incredibly thankful for your contribution to our work.

Sharing our thoughts, experiences, and work with the world was a dream that we never thought would come true. However, we are grateful for this opportunity and are so happy to have you along for the ride.

We would also like to extend our heartfelt appreciation to the following individuals and organizations:

- Our wonderful friends and family who have supported us throughout the writing process and cheered us on every step of the way.
- Our amazing colleagues and fellow creators who have inspired us with their creativity and hard work.
- The talented team at Freepik, who provided 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 grateful for the support and encouragement we have received from everyone. Your contributions have been invaluable in helping us bring this project to life.

Thank you from the bottom of our hearts.

# About the Authors



Tobias loves to build and break things while teaching his learnings along the way. He is a passionate software engineer and educator with several years of professional experience, having worked for small tech startups as well as large car manufacturers. While migrating existing production workloads, he gained the necessary hands-on experience to explore and understand the tricks and peculiarities of AWS. This also enabled him to build resilient, large-scale applications from scratch. He has stumbled into all the traps that 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. He uses a serverless-first architecture on AWS to build a scalable and 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 has worked with AWS for the past 5 years and has built production-grade applications in the areas of machine learning, data lakes, container-based services, and web applications. Currently, he mainly focuses on event-driven architecture.