

An in-depth guide to AWS

Amazon Web Services IN ACTION

THIRD EDITION

Andreas Wittig
Michael Wittig



MANNING



**MEAP Edition
Manning Early Access Program
Amazon Web Services in Action,
Third Edition
Version 7**

Copyright 2022 Manning Publications

For more information on this and other Manning titles go to
manning.com

welcome

Thank you for buying *AWS in Action* and joining MEAP, the early access program.

When we began work on the first edition in 2015, we had no idea what would follow. It's hard to estimate how many people have used this book to get started with AWS. What we do know is that our book has sold nearly 30,000 copies. On top of that, our book was translated into other languages including Portuguese and Japanese. We are grateful to have made such an outstanding contribution to the AWS community.

AWS has evolved over the years and has released thousands of features. Therefore, it is high time that we update our book. We are currently working hard to incorporate all the new features into the third edition.

We are excited to share the first revised chapters of our book with you. Over the next few months, we will release the remaining chapters, though not always in order. But rest assured, purchasing this MEAP edition of the *third* edition automatically gives you access to the full second edition. So if you need a chapter we have not yet finished updating, you can always read an older version in the second edition.

In addition to reworking the existing chapters, we are also working on a brand-new chapter covering modern architectures based on containers using ECS and Fargate-- we will release this soon.

Our goals with *AWS in Action* are:

- Explaining the fundamental concepts of AWS and giving an overview of essential services.
- Introducing the concept of automating the AWS infrastructure.line interface for ease of use
- Teaching how to build highly available, fault-tolerant, and scalable systems on AWS.
- Sharing our excitement about the technology with engaging examples that you can use in your day-to-day work.

We love the innovations and improvements AWS offers based on a strong customer-centric approach. Our goal is to apply the customer-focused approach to the development of our book as well. Because of this, we are looking forward to your feedback. Please tell us if you find bugs in our code, don't understand what we are trying to teach, or have recommendations for the scope of our book. Manning's [liveBook platform](#) is the perfect place to share your feedback.

Besides that, we invite you to follow our journey of writing the 3rd edition.

- Twitter: @andreaswittig and @hellomichibye
- LinkedIn: <https://bit.ly/3sZm4Q3> and <https://bit.ly/3I0kth2>

Thanks,

— Andreas Wittig
— Michael Wittig

brief contents

PART 1: GETTING STARTED

- 1 What is Amazon web services?*
- 2 A simple example: WordPress in fifteen minutes*

PART 2: BUILDING VIRTUAL INFRASTRUCTURE OF COMPUTERS AND NETWORKING

- 3 Using virtual machines: EC2*
- 4 Programming your infrastructure: The command line, SDKs and CloudFormation*
- 5 Securing your system: IAM, security groups and VPC*
- 6 Automating operational tasks with Lambda*

PART 3: STORING DATA IN THE CLOUD

- 7 Storing your objects: S3*
- 8 Storing on hard drives: EBS and instance store*
- 9 Sharing data volumes between machines: EFS*
- 10 Using a relational database service: RDS*
- 11 Caching data in memory: ElastiCache*
- 12 Programming for the NoSQL database service: DynamoDB*

PART 4: ARCHITECTING ON AWS

- 13 Achieving high availability: Availability zones, auto-scaling and CloudWatch*
- 14 Decoupling your infrastructure: Elastic load balancing and simple queue service*
- 15 Automating deployment*
- 16 Designing for fault tolerance*
- 17 Scaling up and down: Auto-scaling and CloudWatch*
- 18 Building modern architectures for the cloud: ECS and Fargate*



What is Amazon Web Services?

This chapter covers

- Overview of Amazon Web Services
- The benefits of using Amazon Web Services
- What you can do with Amazon Web Services
- Creating and setting up an AWS account

Almost every IT solution gets labeled with the term *cloud computing* or even just *cloud* nowadays. Buzzwords like these may help sales, but they're hard to work with when trying to teach—or learn—how to work with these technologies. So for the sake of clarity, let's start this book by defining some terms.

Cloud computing, or the cloud, is a metaphor for supply and consumption of IT resources. The IT resources in the cloud aren't directly visible to the user; there are layers of abstraction in between. The level of abstraction offered by the cloud varies, from offering virtual machines (VMs) to providing software as a service (SaaS) based on complex distributed systems. Resources are available on demand in enormous quantities, and you pay for what you use.

The official definition from the National Institute of Standards and Technology:

Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (networks, virtual machines, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.

— National Institute of Standards and Technology *The NIST Definition of Cloud Computing*

Offerings are often divided into three types:

- *Public*—A cloud managed by an organization and open to use by the general public.
- *Private*—A cloud that virtualizes and distributes the IT infrastructure for a single organization.
- *Hybrid*—A mixture of a public and a private cloud.

Amazon Web Service (AWS) is a public cloud. By combining your on-premises data center with AWS, you building a hybrid cloud.

Cloud computing services also have several classifications:

- *Infrastructure as a service (IaaS)*—Offers fundamental resources like computing, storage, and networking capabilities, using virtual machines such as Amazon EC2, Google Compute Engine, and Microsoft Azure Virtual Machines.
- *Platform as a service (PaaS)*—Provides platforms to deploy custom applications to the cloud, such as AWS Elastic Beanstalk, Google App Engine, and Heroku.
- *Software as a service (SaaS)*—Combines infrastructure and software running in the cloud, including office applications like Amazon WorkSpaces, Google Apps for Work, and Microsoft 365.

AWS is a cloud computing provider with a wide variety of IaaS, PaaS, and SaaS offerings.

1.1 What is Amazon Web Services (AWS)?

Amazon Web Services (AWS) is a platform of web services that offers solutions for computing, storing, and networking, at different layers of abstraction. For example, you can attach volumes to a virtual machine - low level of abstraction - or store and retrieve data via a REST API - high level of abstraction. Use the services provided by AWS to host websites, run enterprise applications, and mine tremendous amounts of data. *Web services* are accessible via the internet by using typical web protocols (such as HTTP) and used by machines or by humans through a UI. The most prominent services provided by AWS are *EC2*, which offers virtual machines, and *S3*, which offers storage capacity. Services on AWS work well together: you can use them to replicate your existing local network setup, or you can design a new setup from scratch. The pricing model for services is pay-per-use.

As an AWS customer, you can choose among different *data centers*. AWS data centers are distributed worldwide. For example, you can start a virtual machine in Japan in exactly the same way as you would start one in Ireland. This enables you to serve customers worldwide.

The map in figure 1.1 shows AWS's data centers. Access is limited to some of them: some data centers are accessible for U.S. government organizations only, and special conditions apply for the data centers in China. Additional data centers have been announced for Canada, Spain, Switzerland, Israel, UAE, India, Australia, and New Zealand.



* limited access

Figure 1.1 AWS data center locations

Now that we have defined the most important terms, the question is: What can you do with AWS?

1.2 What can you do with AWS?

You can run all sorts of application on AWS by using one or a combination of services. The examples in this section will give you an idea of what you can do.

1.2.1 Hosting a web shop

John is CIO of a medium-sized e-commerce business. He wants to develop a fast and reliable web shop. He initially decided to host the web shop on-premises, and three years ago he rented machines in a data center. A web server handles requests from customers, and a database stores product information and orders. John is evaluating how his company can take advantage of AWS by running the same setup on AWS, as shown in figure [1.2](#).

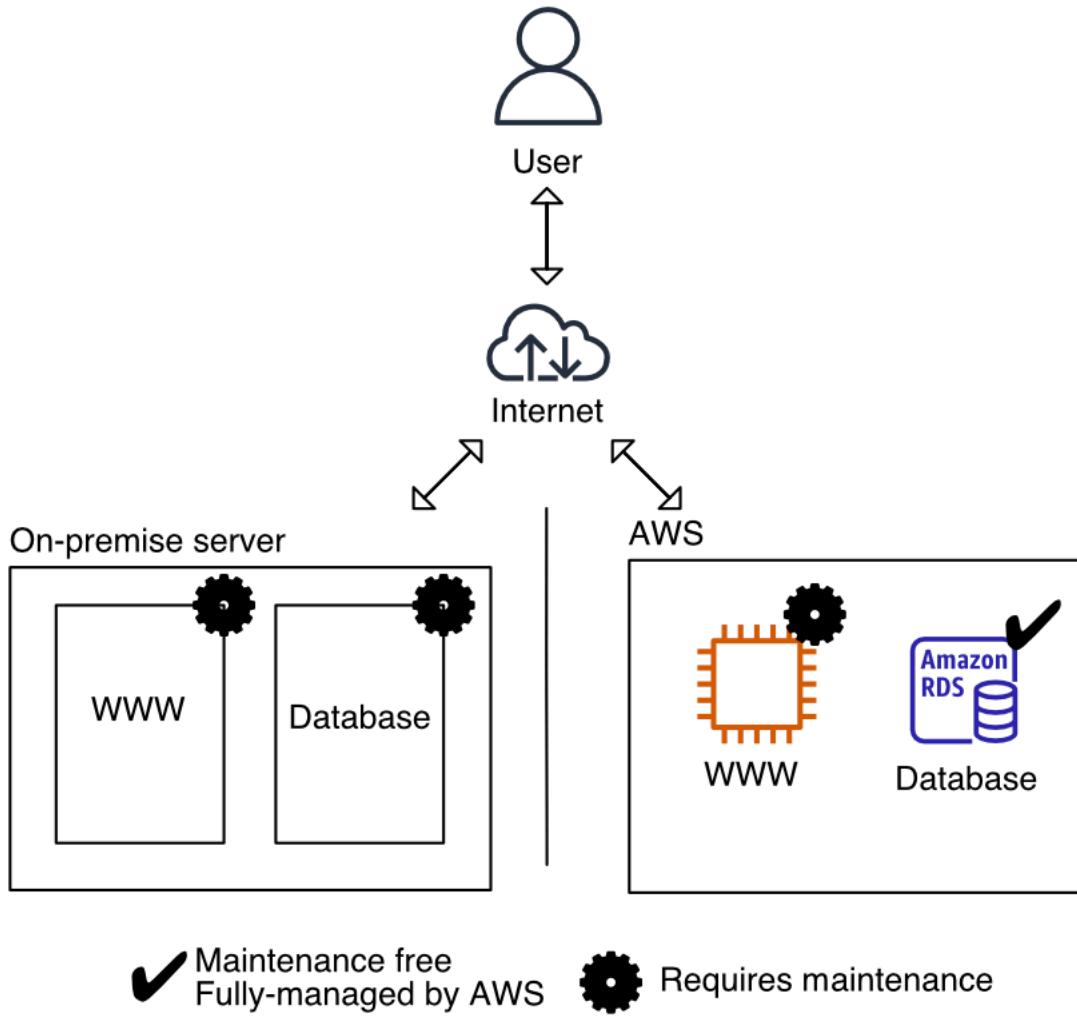


Figure 1.2 Running a web shop on-premises vs. on AWS

John not only wants to lift-and-shift his current on-premises infrastructure to AWS; he wants to get the most out of the advantages the cloud is offering. Additional AWS services allow John to improve his setup.

- The web shop consists of dynamic content (such as products and their prices) and static content (such as the company logo). Splitting these up would reduce the load on the web servers and improve performance by delivering the static content over a content delivery network (CDN).
- Switching to maintenance-free services including a database, an object store, and a DNS system would free John from having to manage these parts of the system, decreasing operational costs and improving quality.
- The application running the web shop can be installed on virtual machines. Using AWS, John can run the same amount of resources he was using on his on-premises machine, but split into multiple smaller virtual machines at no extra cost. If one of these virtual

machines fails, the load balancer will send customer requests to the other virtual machines. This setup improves the web shop's reliability.

Figure 1.3 shows how John enhanced the web shop setup with AWS.

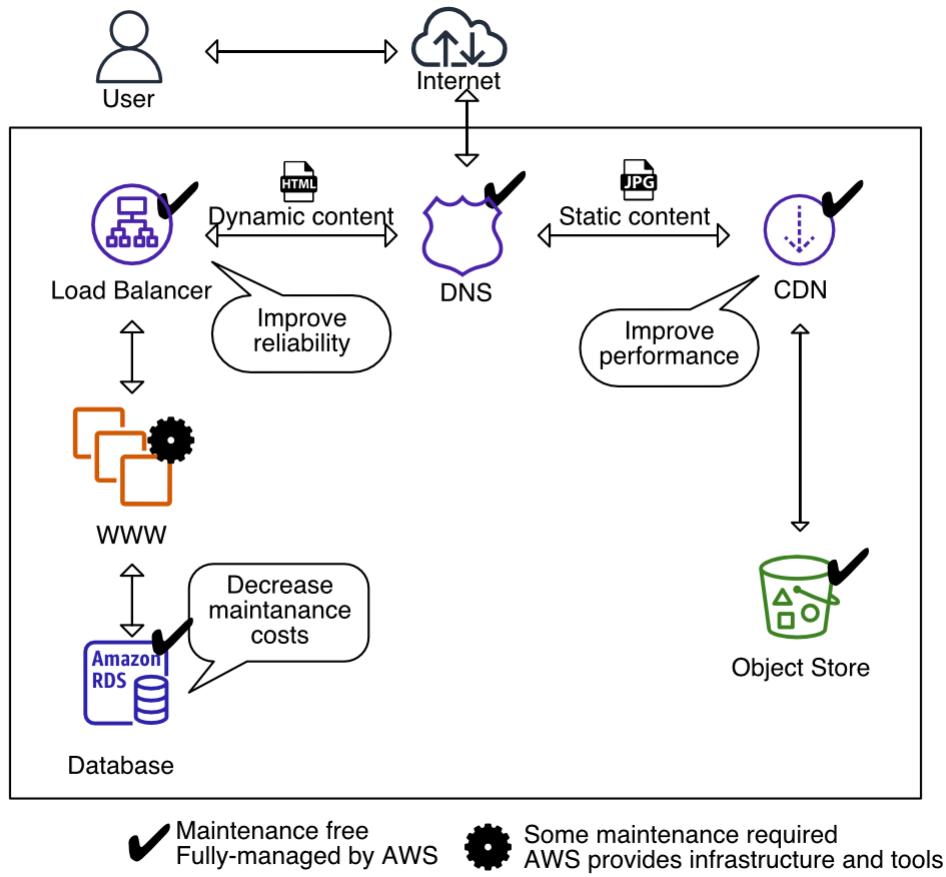


Figure 1.3 Running a web shop on AWS with CDN for better performance, a load balancer for high availability, and a managed database to decrease maintenance costs

John is happy with running his web shop on AWS. By migrating his company's infrastructure to the cloud, he was able to increase the reliability and performance of the web shop.

1.2.2 Running a Java EE application in your private network

Maureen is a senior system architect in a global corporation. She wants to move parts of her company's business applications to AWS when the data-center contract expires in a few months, to reduce costs and gain flexibility. She wants to run enterprise applications (such as Java EE applications) consisting of an application server and an SQL database on AWS. To do so, she defines a virtual network in the cloud and connects it to the corporate network through a Virtual Private Network (VPN) connection. She installs application servers on virtual machines to run the Java EE application. Maureen also wants to store data in an SQL database service (such as Oracle Database Enterprise Edition or Microsoft SQL Server EE).

For security, Maureen uses subnets to separate systems with different security levels from each other. By using access-control lists, she can control ingoing and outgoing traffic for each subnet. For example, the database is only accessible from the JEE server's subnet which helps to protect mission-critical data. Maureen controls traffic to the internet by using Network Address Translation (NAT) and firewall rules as well. Figure 1.4 illustrates Maureen's architecture.

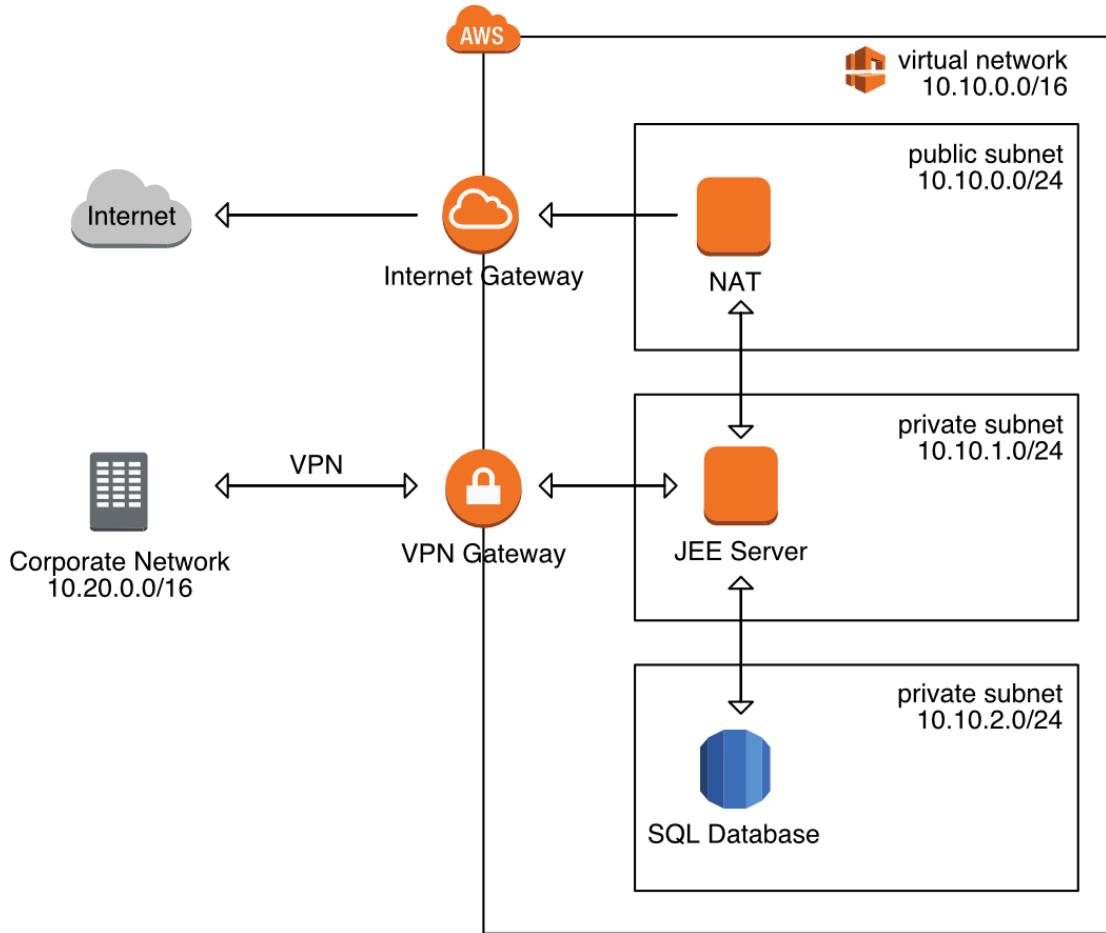


Figure 1.4 Running a Java EE application with enterprise networking on AWS improves flexibility and lowers costs.

Maureen has managed to connect the local data center with a private network running remotely on AWS to enable clients to access the JEE server. To get started, Maureen uses a VPN connection between the local data center and AWS, but she is already thinking about setting up a dedicated network connection to reduce network costs and increase network throughput in the future.

The project was a great success for Maureen. She was able to reduce the time needed to set up an enterprise application from months to hours, as AWS can take care of the virtual machines,

databases, and even the networking infrastructure on demand within a few minutes. Maureen's project also benefits from lower infrastructure costs on AWS, compared to using their own infrastructure on-premises.

1.2.3 Implementing a highly available system

Alexa is a software engineer working for a fast-growing startup. She knows that Murphy's Law applies to IT infrastructure: anything that can go wrong *will* go wrong. Alexa is working hard to build a highly available system to prevent outages from ruining the business. All services on AWS are either highly available or can be used in a highly available way. So, Alexa builds a system like the one shown in figure 1.5 with a high availability architecture. The database service is offered with replication and fail-over handling. In case the primary database instance fails, the standby database is promoted as the new primary database automatically. Alexa uses virtual machines acting as web servers. These virtual machines aren't highly available by default, but Alexa launches multiple virtual machines in different data centers to achieve high availability. A load balancer checks the health of the web servers and forwards requests to healthy machines.

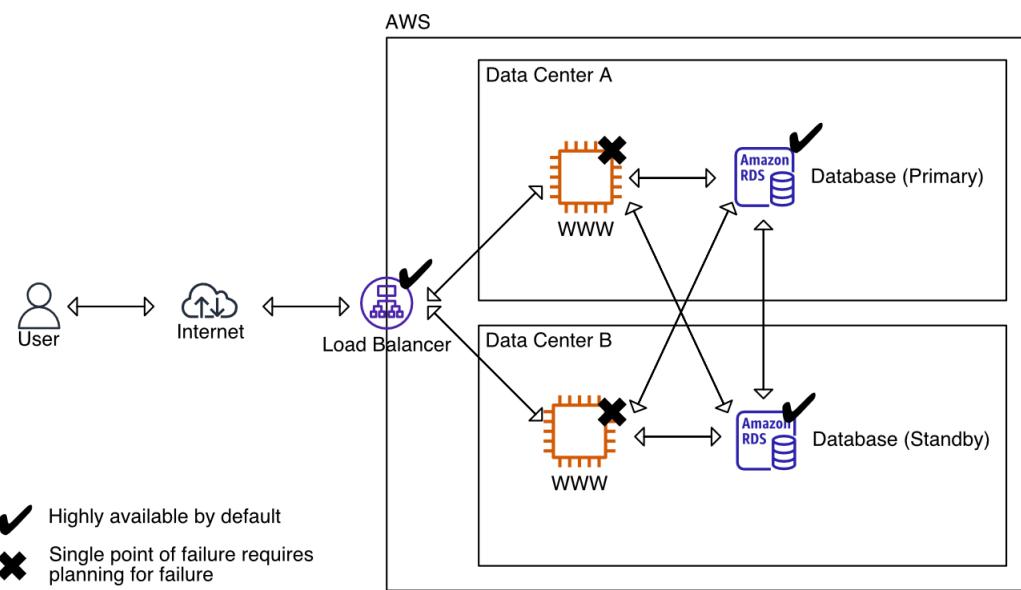


Figure 1.5 Building a highly available system on AWS by using a load balancer, multiple virtual machines, and a database with primary-standby replication

So far, Alexa has protected the startup from major outages. Nevertheless, she and her team are always planning for failure and are constantly improving the resilience of their systems.

1.2.4 Profiting from low costs for batch processing infrastructure

Nick is a data scientist who needs to process massive amounts of measurement data collected from gas turbines. He needs to generate a report containing the maintenance condition of hundreds of turbines daily. Therefore, his team needs a computing infrastructure to analyze the newly arrived data once a day. Batch jobs are run on a schedule and store aggregated results in a database. A business intelligence (BI) tool is used to generate reports based on the data stored in the database.

As the budget for computing infrastructure is very small, Nick and his team have been looking for a cost effective solution to analyze their data. He finds a way to make clever use of AWS's price model:

- *AWS bills virtual machines per second with a minimum of 60 seconds and.* So Nick launches a virtual machine when starting a batch job, and terminates it immediately after the job finished. Doing so allows him to pay for computing infrastructure only when actually using it. This is a big game changer compared to the traditional data center where Nick had to pay a monthly fee for each machine, no matter how much it was used.
- *AWS offers spare capacity in their data centers at substantial discount.* It is not important for Nick to run a batch job at a specific time. He can wait to execute a batch job until there is enough spare capacity available, so AWS offers him a virtual machine with a discount of 50%.

Figure 1.6 illustrates how Nick benefits from the pay-per-use price model for virtual machines.

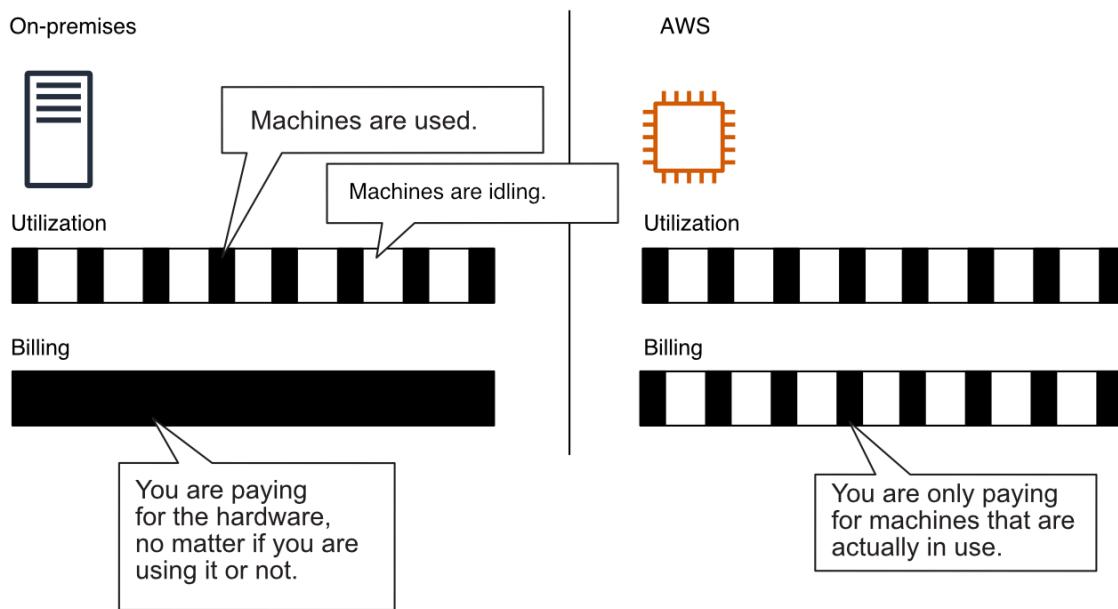


Figure 1.6 Making use of the pay-per-use price model of virtual machines

Nick is happy to have access to a computing infrastructure that allows his team to analyze data at low costs. You now have a broad idea of what you can do with AWS. Generally speaking, you

can host any application on AWS. The next section explains the nine most important benefits AWS has to offer.

1.3 How you can benefit from using AWS

What's the most important advantage of using AWS? Cost savings, you might say. But saving money isn't the only advantage. Let's see how else you can benefit from using AWS by looking at some of its key features.

1.3.1 Innovative and fast-growing platform

AWS is announcing new services, features, and improvements constantly. Go to aws.amazon.com/about-aws/whats-new/ to get an impression of the speed of innovation. We have counted 2080 announcements in 2021. Making use of the innovative technologies provided by AWS helps you to generate valuable solutions for your customers and thus achieve a competitive advantage.

Amazon reported net sales of \$62 billion for 2021. See ir.aboutamazon.com/news-release/news-release-details/2022/Amazon.com-Announces-Fourth-Quarter- if you are interested in the full report. We expect AWS to expand the size and extend of its platform in the upcoming years, for example, by adding additional services and data centers.

1.3.2 Services solve common problems

As you've learned, AWS is a platform of services. Common problems such as load balancing, queuing, sending email, and storing files are solved for you by services. You don't need to reinvent the wheel. It's your job to pick the right services to build complex systems. So let AWS manage those services while you focus on your customers.

1.3.3 Enabling automation

As AWS is API driven, you can automate everything: write code to create networks, start virtual machine clusters, or deploy a relational database. Automation increases reliability and improves efficiency.

The more dependencies your system has, the more complex it gets. A human can quickly lose perspective, whereas a computer can cope with interconnected systems of any size. You should concentrate on tasks humans are good at—such as describing a system—while the computer figures out how to resolve all those dependencies to create the system. Setting up an environment in the cloud based on your blueprints can be automated with the help of infrastructure as code, covered in chapter 4.

1.3.4 Flexible capacity (scalability)

Flexible capacity reduces overcapacity. You can scale from one virtual machine to thousands of virtual machines. Your storage can grow from gigabytes to petabytes. You no longer need to predict your future capacity needs for the coming months and years to purchase hardware.

If you run a web shop, you have seasonal traffic patterns, as shown in figure 1.7. Think about day versus night, and weekday versus weekend or holiday. Wouldn't it be nice if you could add capacity when traffic grows and remove capacity when traffic shrinks? That's exactly what flexible capacity is about. You can start new virtual machines within minutes and throw them away a few hours after that.

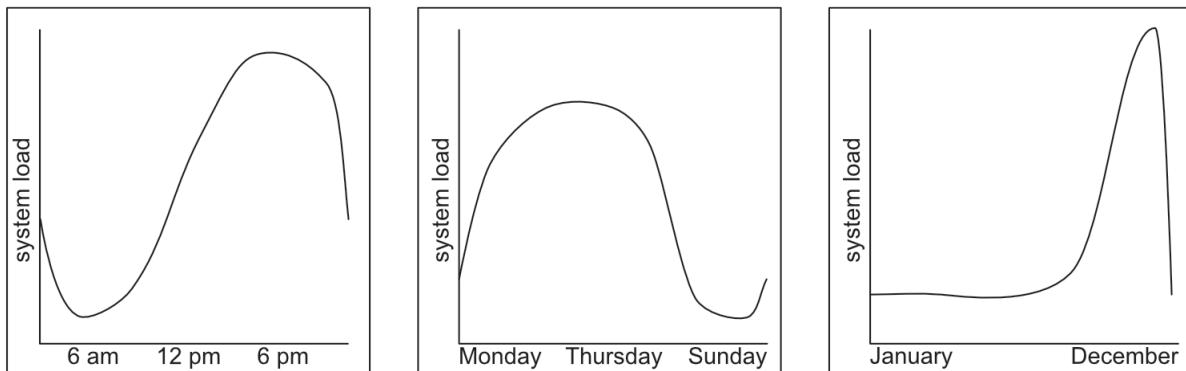


Figure 1.7 Seasonal traffic patterns for a web shop

The cloud has almost no capacity constraints. You no longer need to think about rack space, switches, and power supplies—you can add as many virtual machines as you like. If your data volume grows, you can always add new storage capacity.

Flexible capacity also means you can shut down unused systems. In one of our last projects, the test environment only ran from 7 a.m. to 8 p.m. on weekdays, allowing us to save 60%.

1.3.5 Built for failure (reliability)

Most AWS services are highly available or fault tolerant by default. If you use those services, you get reliability for free. Also, AWS provides tooling allowing you to build systems in a reliable way. It provides everything you need to create your own highly available or even fault-tolerant systems.

1.3.6 Reducing time to market

In AWS, you request a new virtual machine, and a few minutes later that virtual machine is booted and ready to use. The same is true with any other AWS service available. You can use them all on demand.

Your development process will be faster because of the shorter feedback loops. You can

eliminate constraints such as the number of test environments available; if you need another test environment, you can create it for a few hours.

1.3.7 Benefiting from economies of scale

AWS is increasing its global infrastructure constantly. Thus AWS benefits from an economy of scale. As a customer, you will benefit partially from these effects.

AWS reduces prices for their cloud services every now and then. A few examples:

- In January 2019, AWS reduced the price for running containers on Fargate by 20% for vCPU and 65% for memory.
- In November 2020, AWS reduced prices for EBS volumes of type Cold HDD by 40%.
- In November 2021, AWS reduced prices for S3 storage by up to 31% in three storage classes.
- In April 2022, AWS removed additional costs for network traffic between data centers when using AWS PrivateLink, AWS Transit Gateway, and AWS Client VPN.

1.3.8 Global infrastructure

Are you serving customers worldwide? Making use of AWS's global infrastructure has the following advantages: low network latencies between your customers and your infrastructure, being able to comply with regional data protection requirements, and benefiting from different infrastructure prices in different regions. AWS offers data centers in North America, South America, Europe, Africa, Asia, and Australia, so you can deploy your applications worldwide with little extra effort.

1.3.9 Professional partner

When you use AWS services, you can be sure that their quality and security follow the latest standards and certifications. For example:

- *ISO 27001*—A worldwide information security standard certified by an independent and accredited certification body.
- *ISO 9001*—A standardized quality management approach used worldwide and certified by an independent and accredited certification body.
- *PCI DSS Level 1*—A data security standard (DSS) for the payment card industry (PCI) to protect cardholders data.

Go to aws.amazon.com/compliance/ if you want to dive into the details. If you're still not convinced that AWS is a professional partner, you should know that Expedia, Volkswagen, FINRA, Airbnb, Slack, and many more are running serious workloads on AWS.¹

We have discussed a lot of reasons to run your workloads on AWS. But what does AWS cost? You will learn more about the pricing models in the next section.

1.4 How much does it cost?

A bill from AWS is similar to an electric bill. Services are billed based on use. You pay for the time a virtual machine was running, the used storage from the object store, or the number of running load balancers. Services are invoiced on a monthly basis. The pricing for each service is publicly available; if you want to calculate the monthly cost of a planned setup, you can use the AWS Pricing Calculator (calculator.aws/).

1.4.1 Free Tier

You can use some AWS services for free within the first 12 months of your signing up. The idea behind the Free Tier is to enable you to experiment with AWS and get some experience using its services. Here is a taste of what's included in the Free Tier:

- 750 hours (roughly a month) of a small virtual machine running Linux or Windows. This means you can run one virtual machine for a whole month or you can run 750 virtual machines for one hour.
- 750 hours (or roughly a month) of a classic or application load balancer.
- Object store with 5 GB of storage.
- Small relational database with 20 GB of storage, including backup.
- 25 GB of data stored on NoSQL database.

If you exceed the limits of the Free Tier, you start paying for the resources you consume without further notice. You'll receive a bill at the end of the month. We'll show you how to monitor your costs before you begin using AWS.

After your one-year trial period ends, you pay for all resources you use. But some resources are free forever. For example, the first 25 GB of the NoSQL database are free forever.

You get additional benefits, as detailed at aws.amazon.com/free. This book will use the Free Tier as much as possible and will clearly state when additional resources are required that aren't covered by the Free Tier.

1.4.2 Billing example

As mentioned earlier, you can be billed in several ways:

- *Based on time of use*—A virtual machine is billed per second. A load balancer is billed per hour.
- *Based on traffic*—Traffic is measured in gigabytes or in number of requests, for example.
- *Based on storage usage*—Usage can be measured by capacity (for example, 50 GB volume no matter how much you use) or real usage (such as 2.3 GB used).

Remember the web shop example from section 1.2? Figure 1.8 shows the web shop and adds information about how each part is billed.

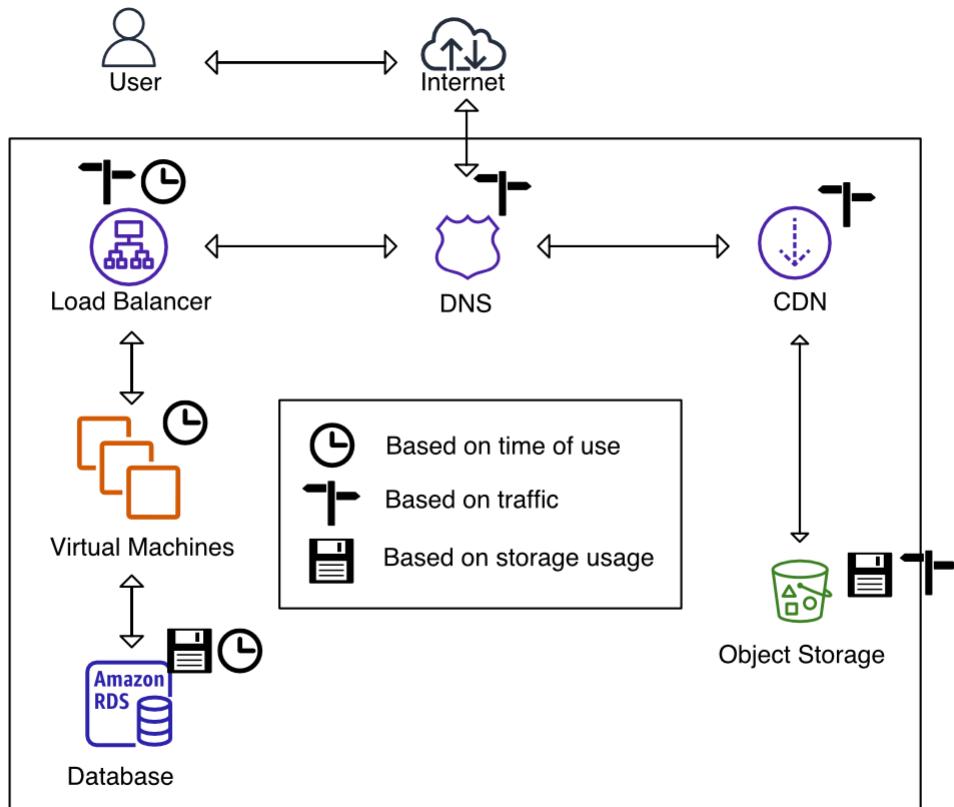


Figure 1.8 Some services are billed based on time of use, others by throughput, or consumed storage.

Let's assume your web shop started successfully in January, and you ran a marketing campaign to increase sales for the next month. Lucky you: you were able to increase the number of visitors to your web shop fivefold in February. As you already know, you have to pay for AWS based on usage. Table 1.1 shows your bill for February. The number of visitors increased from 100,000 to 500,000, and your monthly bill increased from \$112 USD to \$473 USD, which is a 4.2-fold increase. Because your web shop had to handle more traffic, you had to pay more for services, such as the CDN, the web servers, and the database. Other services, like the amount of storage needed for static files, didn't change, so the price stayed the same.

Table 1.1 How an AWS bill changes if the number of web shop visitors increases

Service	January usage	February usage	February charge	Increase
Visits to website	100,000	500,000		
CDN	25 M requests + 25 GB traffic	125 M requests + 125 GB traffic	\$115.00 USD	\$100.00 USD
Static files	50 GB used storage	50 GB used storage	\$1.15 USD	\$0.00 USD
Load balancer	748 hours + 50 GB traffic	748 hours + 250 GB traffic	\$19.07 USD	\$1.83 USD
Web servers	1 virtual machine = 748 hours	4 virtual machines = 2,992 hours	\$200.46 USD	\$150.35 USD
Database (748 hours)	Small virtual machine + 20 GB storage	Large virtual machine + 20 GB storage	\$133.20 USD	\$105.47 USD
DNS	2 M requests	10 M requests	\$4.00 USD	\$3.20 USD
Total cost			\$472.88 USD	\$360.85 USD

With AWS, you can achieve a linear relationship between traffic and costs. And other opportunities await you with this pricing model.

1.4.3 Pay-per-use opportunities

The AWS pay-per-use pricing model creates new opportunities. For example, the barrier for starting a new project is lowered, as you no longer need to invest in infrastructure up front. You can start virtual machines on demand and only pay per second of usage, and you can stop using those virtual machines whenever you like and no longer have to pay for them. You don't need to make an upfront commitment regarding how much storage you'll use.

Another example: a big virtual machine costs exactly as much as two smaller ones with the same capacity. Thus you can divide your systems into smaller parts, because the cost is the same. This makes fault tolerance affordable not only for big companies but also for smaller budgets.

1.5 Comparing alternatives

AWS isn't the only cloud computing provider. Microsoft Azure and Google Cloud Platform (GCP) are major players as well.

The three major cloud providers share a lot in common. They all have:

- A worldwide infrastructure that provides computing, networking, and storage capabilities.
- An IaaS offering that provides virtual machines on-demand: Amazon EC2, Azure Virtual Machines, Google Compute Engine.
- Highly distributed storage systems able to scale storage and I/O capacity without limits: Amazon S3, Azure Blob storage, Google Cloud Storage.
- A pay-as-you-go pricing model.

But what are the differences between the cloud providers?

AWS is the market leader in cloud computing, offering an extensive product portfolio. Even if AWS has expanded into the enterprise sector during recent years, it is still obvious that AWS started with services to solve internet-scale problems. Overall, AWS is building great services based on innovative, mostly open source, technologies. AWS offers complicated but rock-solid ways to restrict access to your cloud infrastructure.

Microsoft Azure provides Microsoft's technology stack in the cloud, recently expanding into web-centric and open source technologies as well. It seems like Microsoft is putting a lot of effort into catching up with Amazon's market share in cloud computing.

The Google Cloud Platform (GCP) is focused on developers looking to build sophisticated distributed systems. Google combines their worldwide infrastructure to offer scalable and fault-tolerant services (such as Google Cloud Load Balancing). The GCP seems more focused on cloud-native applications than on migrating your locally hosted applications to the cloud, in our opinion.

There are no shortcuts to making an informed decision about which cloud provider to choose. Each use case and project is different. The devil is in the details. Also don't forget where you are coming from. (Are you using Microsoft technology heavily? Do you have a big team consisting of system administrators or are you a developer-centric company?) Overall, in our opinion, AWS is the most mature and powerful cloud platform available at the moment.

1.6 Exploring AWS services

In this section you will get an idea of the range of services that AWS offers. We'll also give you a mental model to help you better understand AWS.

Hardware for computing, storing, and networking is the foundation of the AWS cloud. AWS runs services on this hardware, as shown in figure [1.9](#). The API acts as an interface between AWS services and your applications.

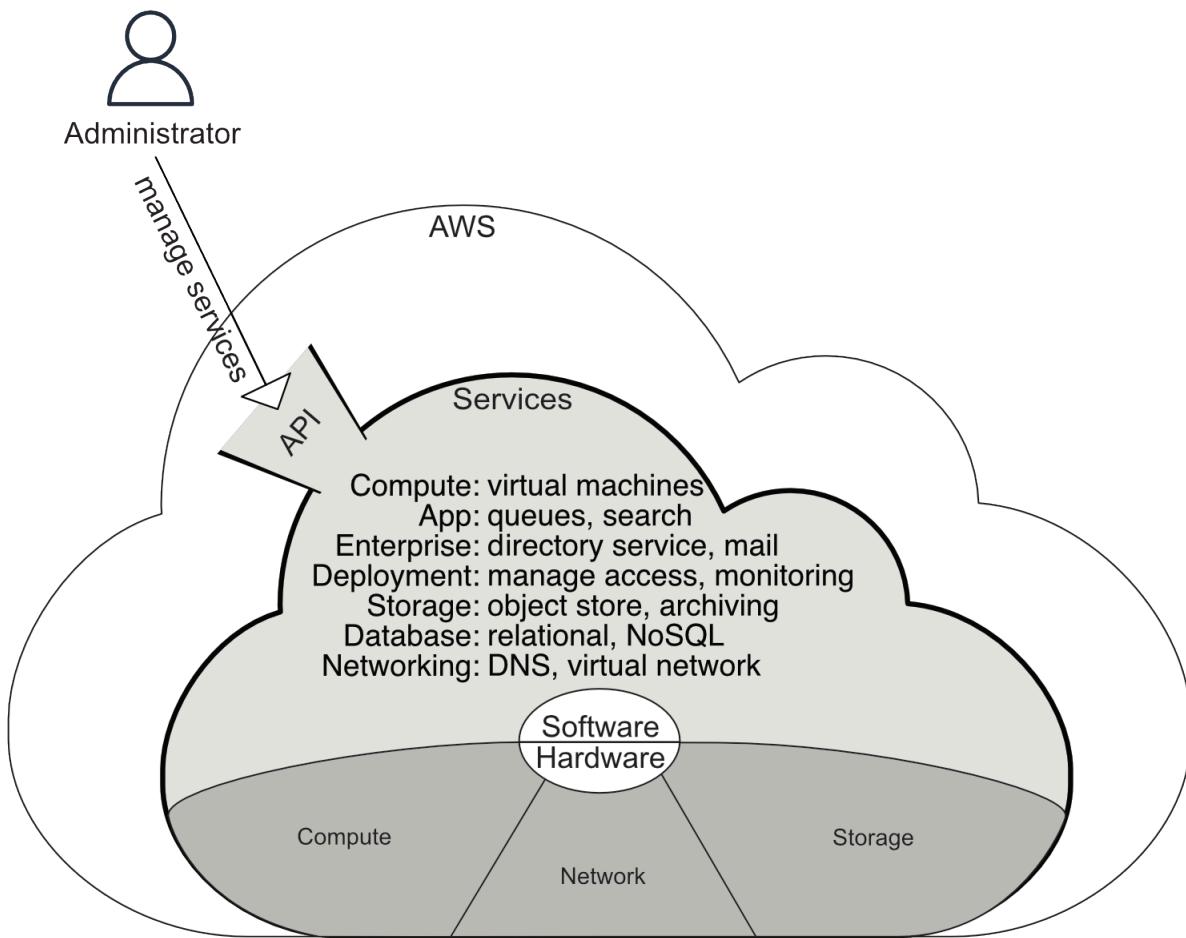


Figure 1.9 The AWS cloud is composed of hardware and software services accessible via an API.

You can manage services by sending requests to the API manually via a web-based GUI like the Management Console, a command-line interface (CLI), or programmatically via an SDK. Virtual machines have a special feature: you can connect to virtual machines through SSH, for example, and gain administrator access. This means you can install any software you like on a virtual machine. Other services, like the NoSQL database service, offer their features through an API and hide everything that's going on behind the scenes. Figure 1.10 shows an administrator installing a custom PHP web application on a virtual machine and managing dependent services such as a NoSQL database used by the application.

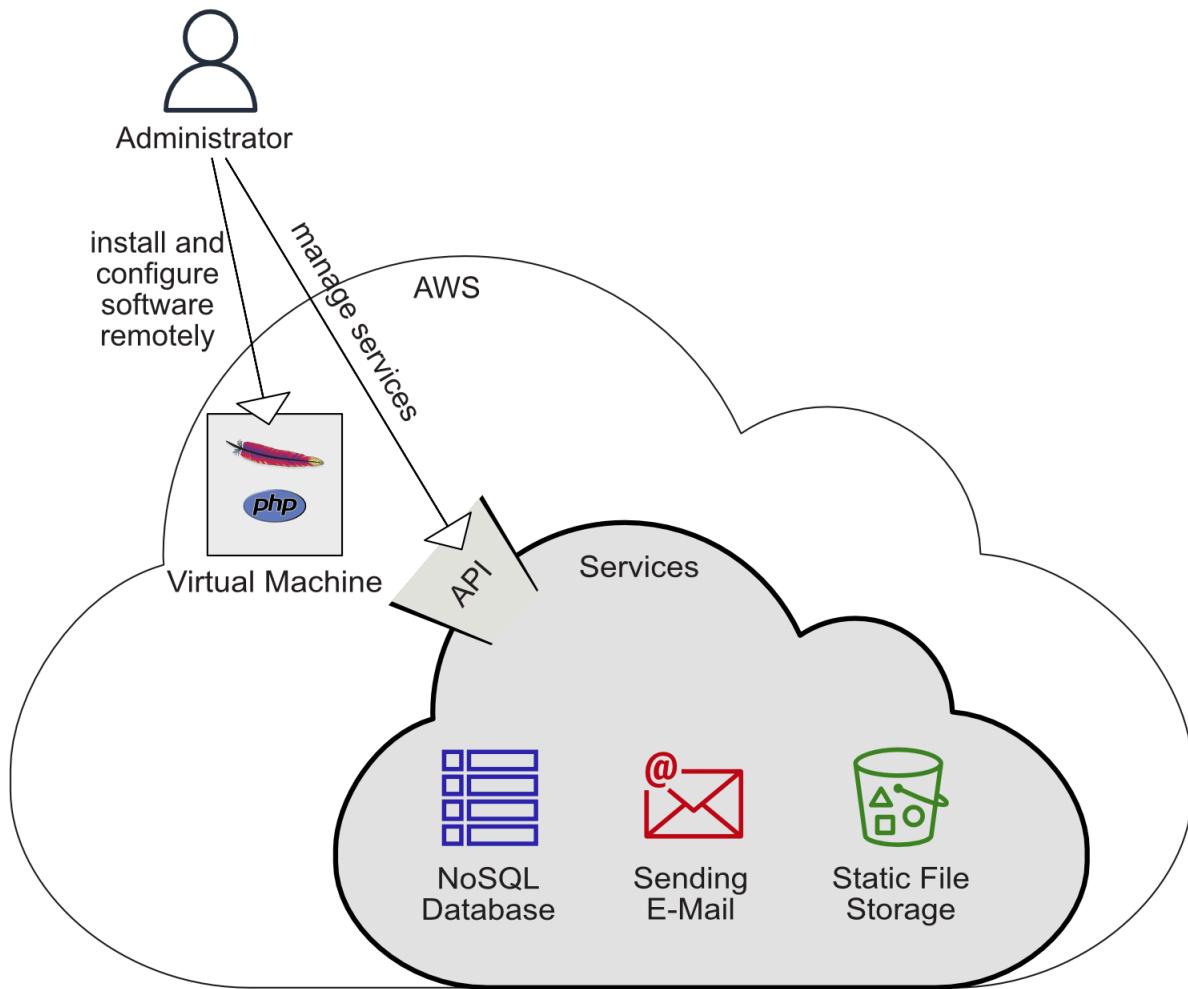


Figure 1.10 Managing a custom application running on a virtual machine and cloud-native services

Users send HTTP requests to a virtual machine. This virtual machine is running a web server along with a custom PHP web application. The web application needs to talk to AWS services in order to answer HTTP requests from users. For example, the application might need to query data from a NoSQL database, store static files, and send email. Communication between the web application and AWS services is handled by the API, as figure [1.11](#) shows.

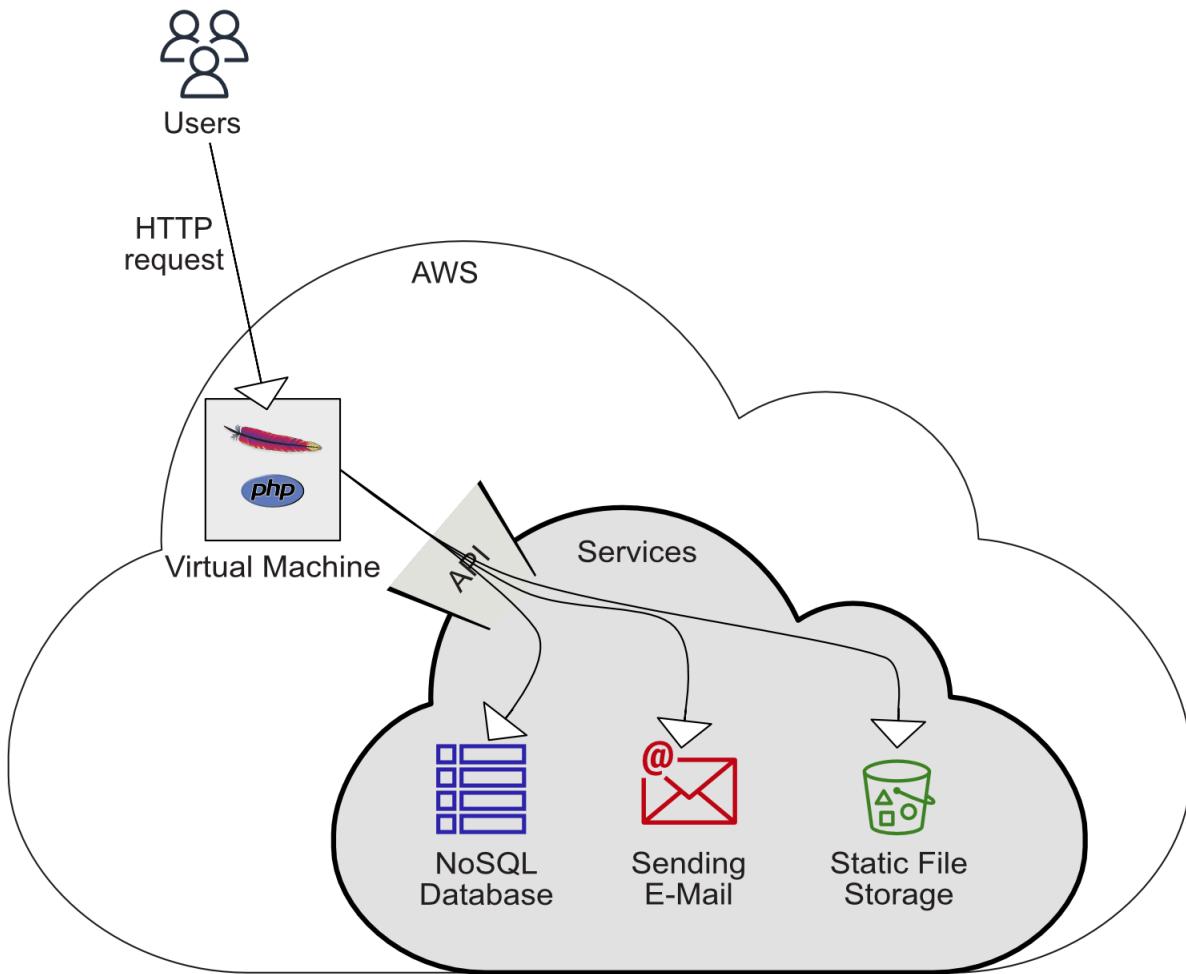


Figure 1.11 Handling an HTTP request with a custom web application using additional AWS services

The number of services available can be scary at the outset. When logging into AWS's web interface you are presented with an overview listing around 200 services. On top of that, new services are announced constantly during the year and at the big conference in Las Vegas, AWS re:Invent.

AWS offers services in the following categories:

• Analytics	• Application Integration	• AR & VR
• AWS Cost Management	• Blockchain	• Business Applications
• Compute	• Containers	• Customer Enablement
• Database	• Developer Tools	• End User Computing
• Front-end Web & Mobile	• Game Development	• Internet of Things
• Machine Learning	• Management & Governance	• Media Services
• Migration & Transfer	• Networking & Content Delivery	• Quantum Technologies
• Robotics	• Satellite	• Security, Identity, & Compliance
• Storage		

It is impossible to cover the many services offered by AWS in one book. Therefore, we are focusing on the services that will best help you get started quickly, as well as the most widely used services. The following services are covered in detail in our book:

- *EC2*—Virtual machines
- *ECS and Fargate*—Running and managing containers
- *Lambda*—Executing functions
- *S3*—Object store
- *Glacier*—Archiving data
- *EBS*—Block storage for virtual machines
- *EFS*—Network filesystem
- *RDS*—SQL databases
- *DynamoDB*—NoSQL database
- *ElastiCache*—In-memory key-value store
- *VPC*—Virtual network

- *ELB*—Load balancers
- *Simple Queue Service*—Distributed queues
- *CodeDeploy*—Automate code deployments
- *CloudWatch*—Monitoring and logging
- *CloudFormation*—Automating your infrastructure
- *IAM*—Restricting access to your cloud resources

We are missing at least three important topics that would fill their own books: continuous delivery, machine learning, and analytics. "Let us know when you are interested in reading one of these unwritten books."

But how do you interact with an AWS service? The next section explains how to use the web interface, the CLI, and SDKs to manage and access AWS resources.

1.7 Interacting with AWS

When you interact with AWS to configure or use services, you make calls to the API. The API is the entry point to AWS, as figure 1.12 demonstrates.

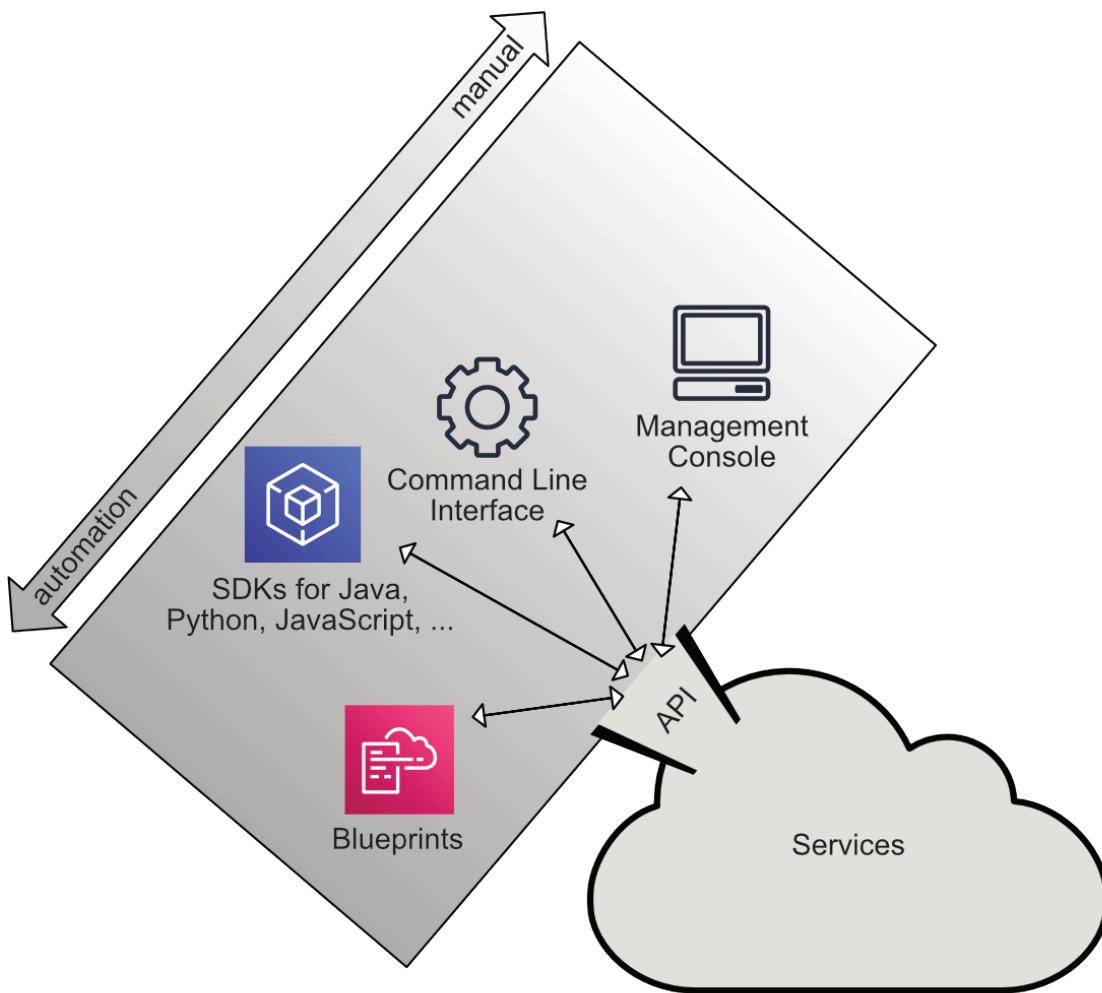


Figure 1.12 Different ways to access the AWS API, allowing you to manage and access AWS services

Next, we'll give you an overview of the tools available for communicating with AWS's APIs: the Management Console, the command-line interface, the SDKs, and infrastructure blueprints. We will compare the different tools, and you will learn how to use all of them while working your way through the book.

1.7.1 Management Console

The AWS Management Console allows you to manage and access AWS services through a graphical user interface (GUI), which works with modern browsers on desktop computers, laptops, and tablets. See figure [1.13](#).

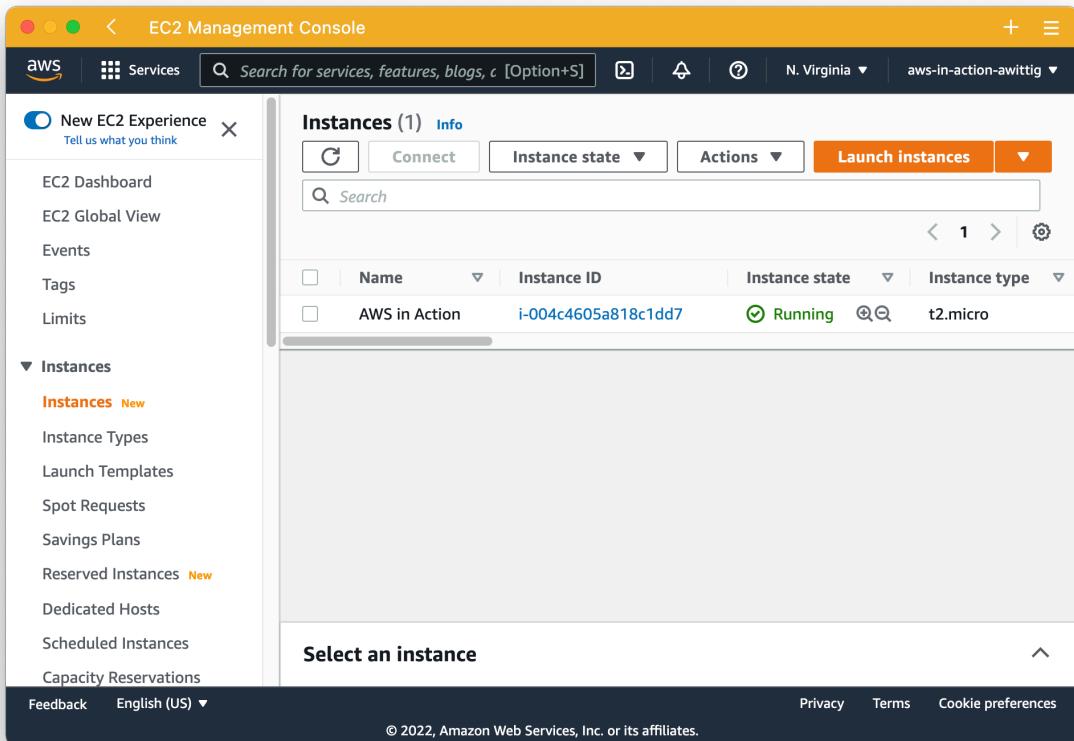
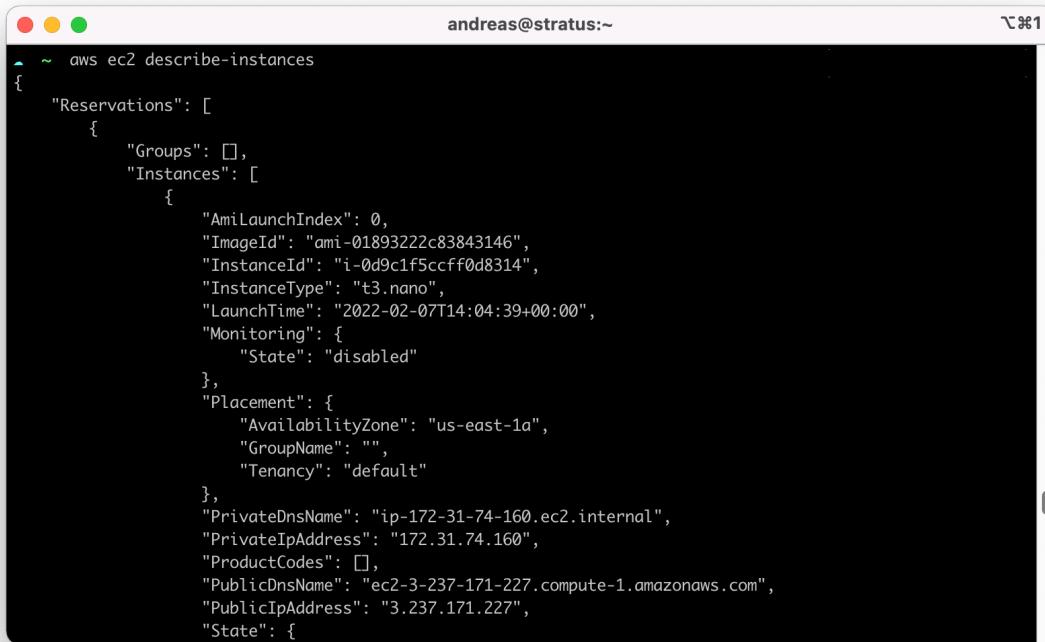


Figure 1.13 The AWS Management Console offers a GUI to manage and access AWS services.

When getting started or experimenting with AWS, the Management Console is the best place to start. It helps you to gain an overview of the different services quickly. The Management Console is also a good way to set up a cloud infrastructure for development and testing.

1.7.2 Command-line interface

The command-line interface (CLI) allows you to manage and access AWS services within your terminal. Because you can use your terminal to automate or semi-automate recurring tasks, CLI is a valuable tool. You can use the terminal to create new cloud infrastructures based on blueprints, upload files to the object store, or get the details of your infrastructure's networking configuration regularly. Figure 1.14 shows the CLI in action.



```
andreas@stratus:~ % aws ec2 describe-instances
{
  "Reservations": [
    {
      "Groups": [],
      "Instances": [
        {
          "AmiLaunchIndex": 0,
          "ImageId": "ami-01893222c83843146",
          "InstanceId": "i-0d9c1f5ccff0d8314",
          "InstanceType": "t3.nano",
          "LaunchTime": "2022-02-07T14:04:39+00:00",
          "Monitoring": {
            "State": "disabled"
          },
          "Placement": {
            "AvailabilityZone": "us-east-1a",
            "GroupName": "",
            "Tenancy": "default"
          },
          "PrivateDnsName": "ip-172-31-74-160.ec2.internal",
          "PrivateIpAddress": "172.31.74.160",
          "ProductCodes": [],
          "PublicDnsName": "ec2-3-237-171-227.compute-1.amazonaws.com",
          "PublicIpAddress": "3.237.171.227",
          "State": {
            "Code": 16,
            "Name": "running"
          }
        }
      ]
    }
  ]
}
```

Figure 1.14 The CLI allows you to manage and access AWS services from your terminal.

If you want to automate parts of your infrastructure with the help of a continuous integration server, like Jenkins, the CLI is the right tool for the job. The CLI offers a convenient way to access the API and combine multiple calls into a script.

You can even begin to automate your infrastructure with scripts by chaining multiple CLI calls together. The CLI is available for Windows, Mac, and Linux, and there is also a PowerShell version available.

1.7.3 SDKs

Use your favorite programming language to interact with the AWS API. AWS offers SDKs for the following platforms and languages:

• JavaScript	• Python	• PHP
• .NET	• Ruby	• Java
• Go	• Node.js	• C++

SDKs are typically used to integrate AWS services into applications. If you’re doing software development and want to integrate an AWS service like a NoSQL database or a push-notification service, an SDK is the right choice for the job. Some services, such as queues and topics, must be used with an SDK.

1.7.4 Blueprints

A *blueprint* is a description of your system containing all resources and their dependencies. An Infrastructure as Code tool compares your blueprint with the current system, and calculates the steps to create, update, or delete your cloud infrastructure. Figure 1.15 shows how a blueprint is transferred into a running system.

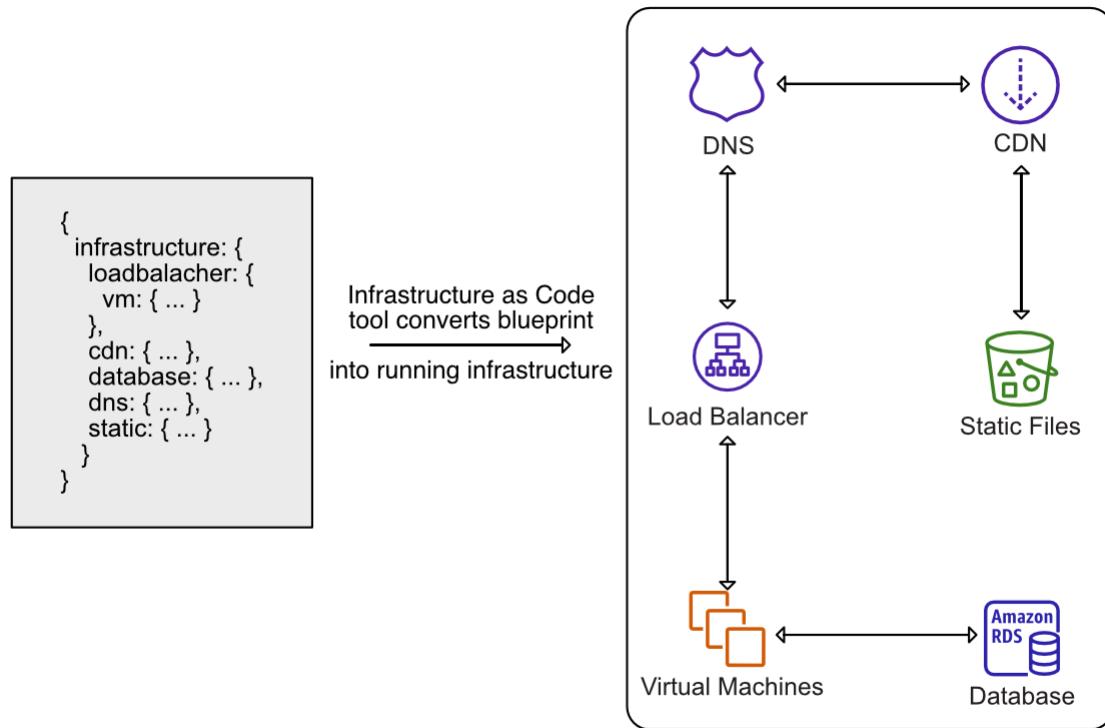


Figure 1.15 Infrastructure automation with blueprints

Consider using blueprints if you have to control many or complex environments. Blueprints will help you to automate the configuration of your infrastructure in the cloud. You can use them to set up a network and launch virtual machines, for example.

Automating your infrastructure is also possible by writing your own source code with the help of the CLI or the SDKs. But doing so requires you to resolve dependencies, make sure you are able to update different versions of your infrastructure, and handle errors yourself. As you will see in

chapter 4, using a blueprint and an Infrastructure-as-Code tool solves these challenges for you. It's time to get started creating your AWS account and exploring AWS practice after all that theory.

1.8 Creating an AWS account

Before you can start using AWS, you need to create an account. Your account is a basket for all your cloud resources. You can attach multiple users to an account if multiple humans need access to it; by default, your account will have one AWS account root user. To create an account, you need the following:

- A telephone number to validate your identity
- A credit card to pay your bills

NOTE

Using an old account?

It is possible to use your existing AWS account while working through this book. In this case, your usage might not be covered by the Free Tier. So you might have to pay for the use.

Also, in case you created your existing AWS account before Dec. 4, 2013, please create a new one, as there are some legacy issues that might cause trouble when following our examples.

NOTE

Multiple AWS accounts?

It is fine to create more than one AWS account. AWS even encourages to do so, to isolate different workloads.

1.8.1 Signing up

The sign-up process consists of five steps:

1. Providing login credentials
2. Providing contact information
3. Providing payment details
4. Verifying your identity
5. Choosing a support plan

Point your favorite web browser to aws.amazon.com, and click the *Create an AWS account* button.

1. PROVIDING LOGIN CREDENTIALS

Creating an AWS account starts with defining a unique AWS account name, as shown in figure 1.16. The AWS account name has to be globally unique among all AWS customers. Try `aws-in-action-$yourname` and replace `$yourname` with your name. Beside the account name, you have to specify an email address and a password used to authenticate the root user of your AWS account.

We advise you to choose a strong password to prevent misuse of your account. *Use a password consisting of at least 20 characters.* Protecting your AWS account from unwanted access is crucial to avoid data breaches, data loss, or unwanted resource usage on your behalf.

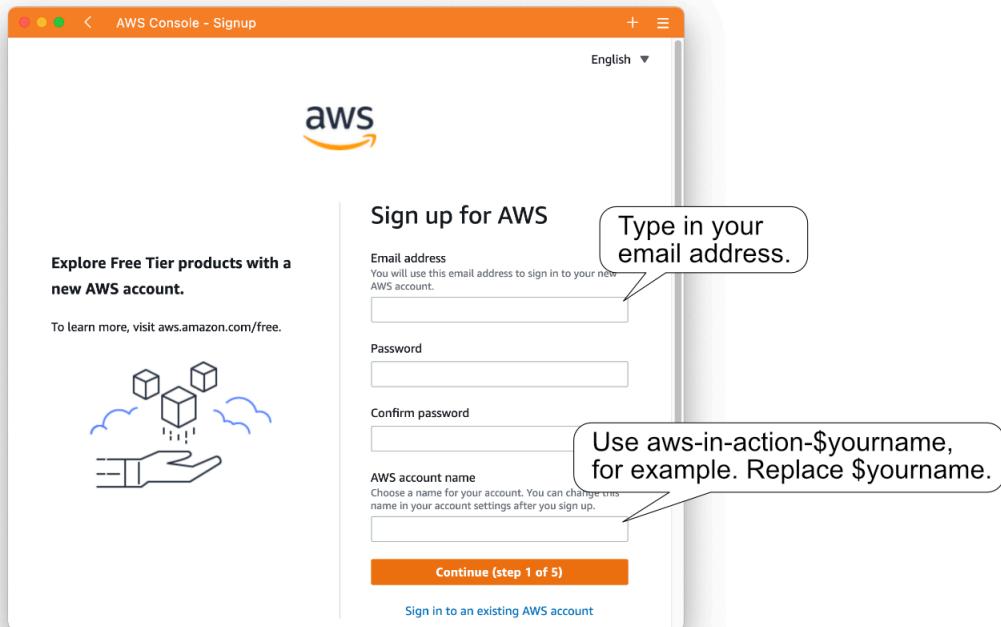


Figure 1.16 First step of creating an AWS account: account name and password

2.PROVIDING CONTACT INFORMATION

The next step, as shown in figure 1.17, is adding your contact information. Fill in all the required fields, and continue.

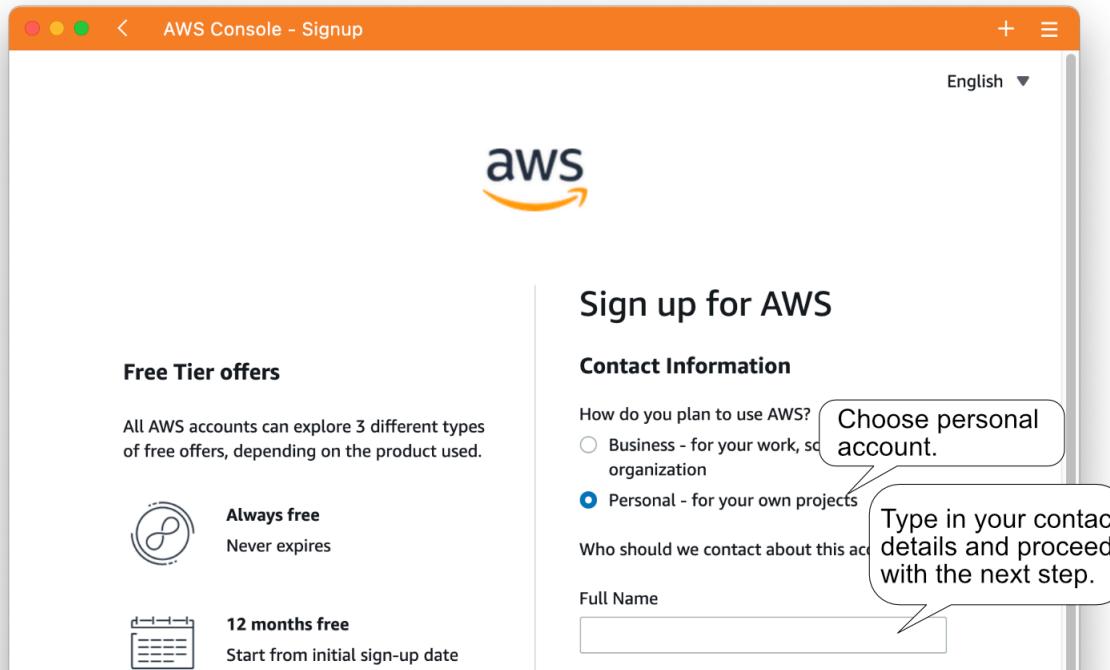


Figure 1.17 Second step of creating an AWS account: contact details

3. PROVIDING PAYMENT DETAILS

Next, the screen shown in figure 1.18 asks for your payment information. Provide your credit card information. There's an option to change the currency setting from USD to AUD, BRL, CAD, CHF, CNY, DKK, EUR, GBP, HKD, JPY, KRW, NOK, NZD, SEK, SGD, or ZAR later on if that's more convenient for you. If you choose this option, the amount in USD is converted into your local currency at the end of the month.

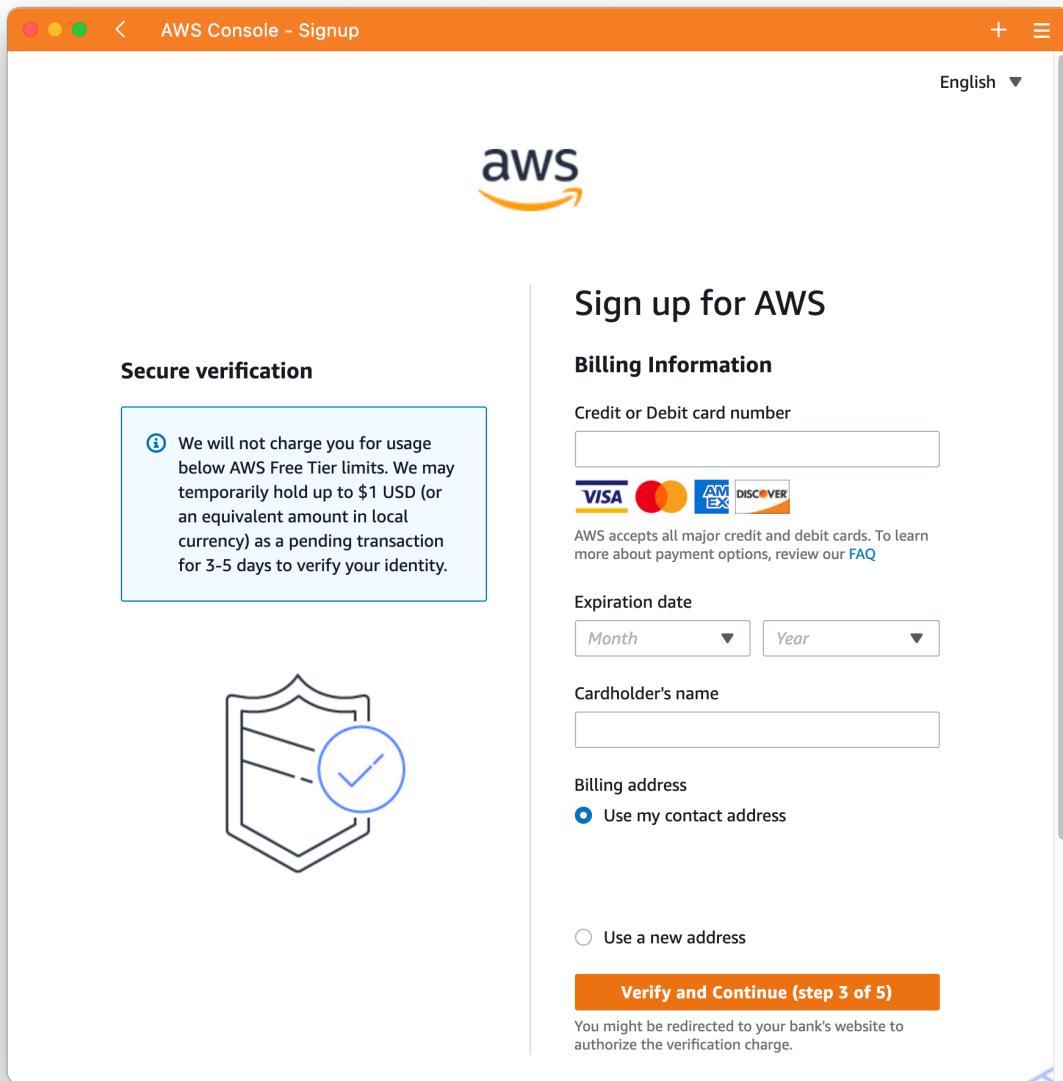


Figure 1.18 Third step of creating an AWS account: payment details

4. VERIFYING YOUR IDENTITY

The next step is to verify your identity. Figure 1.19 shows the first step of the process.

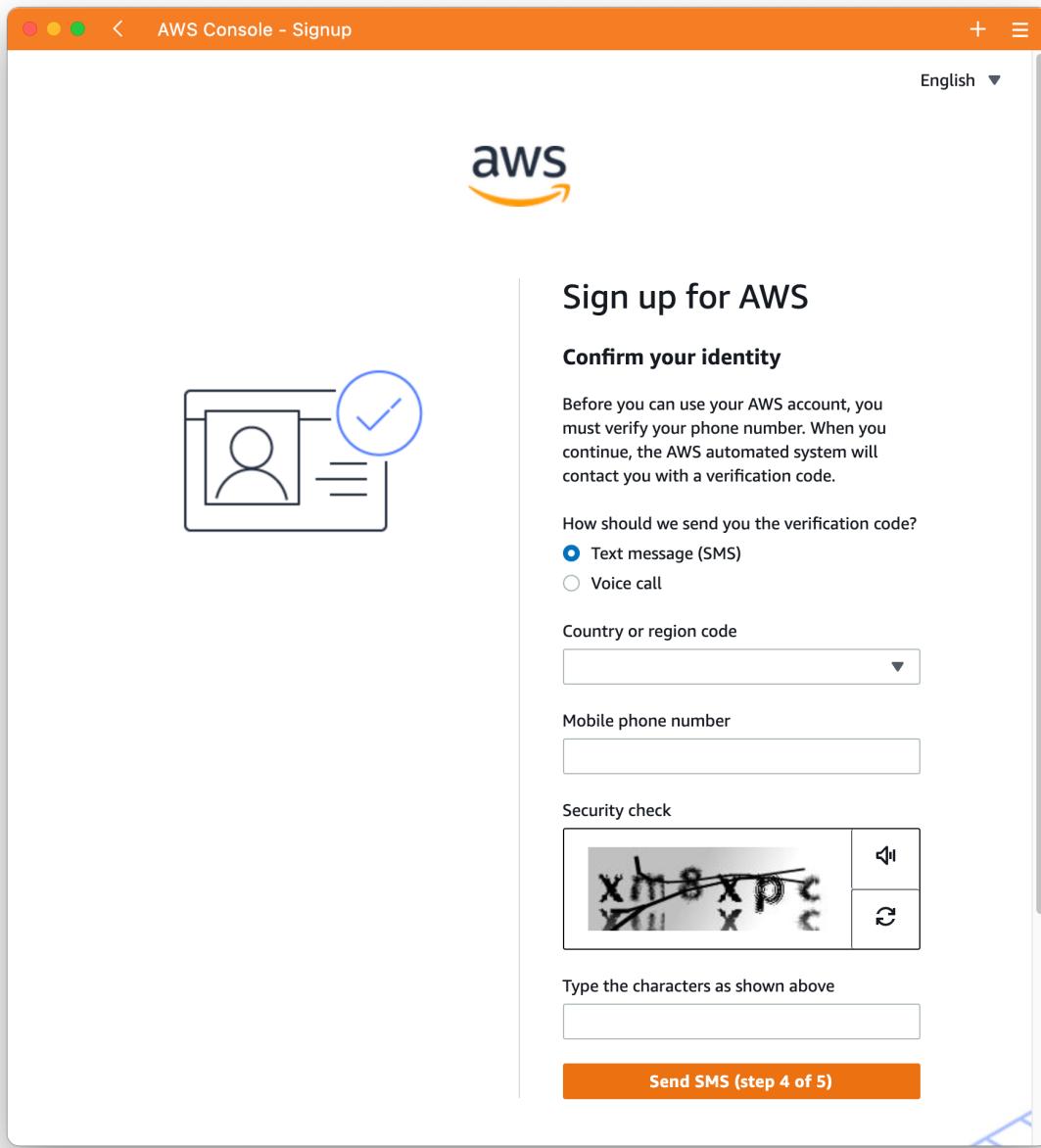


Figure 1.19 Fourth step of creating an AWS account: verify identity

After you complete the first part of the form, you'll receive a text message or call from AWS. All you need to do, is to type in the verification code.

5. CHOOSING A SUPPORT PLAN

The last step is to choose a support plan; see figure 1.20. For now, select the *Basic* plan, which is free. When running a production workload an AWS, we recommend at least a *Developer* plan to be able to ask questions about upcoming issues.

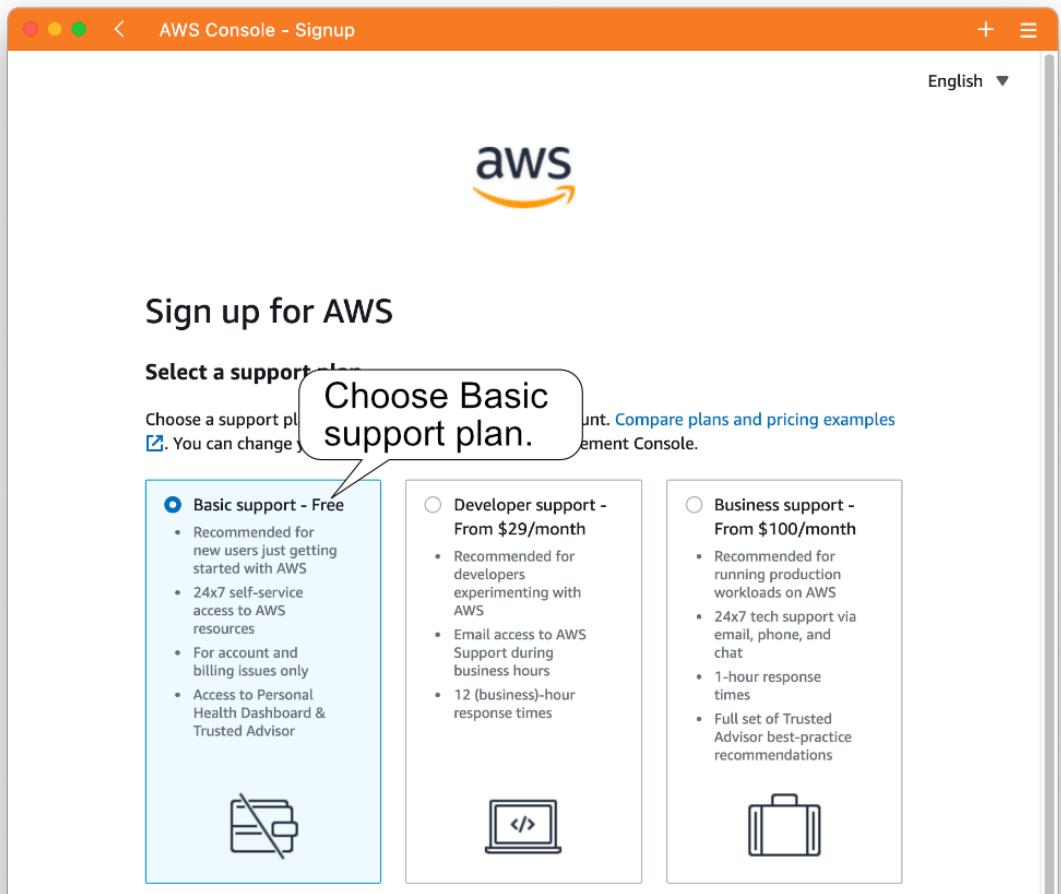


Figure 1.20 Fifth step of creating an AWS account: choose support plan

High five! You're done. Click *Go to the AWS Management Console* as shown in figure 1.21 to sign into your AWS account for the first time.

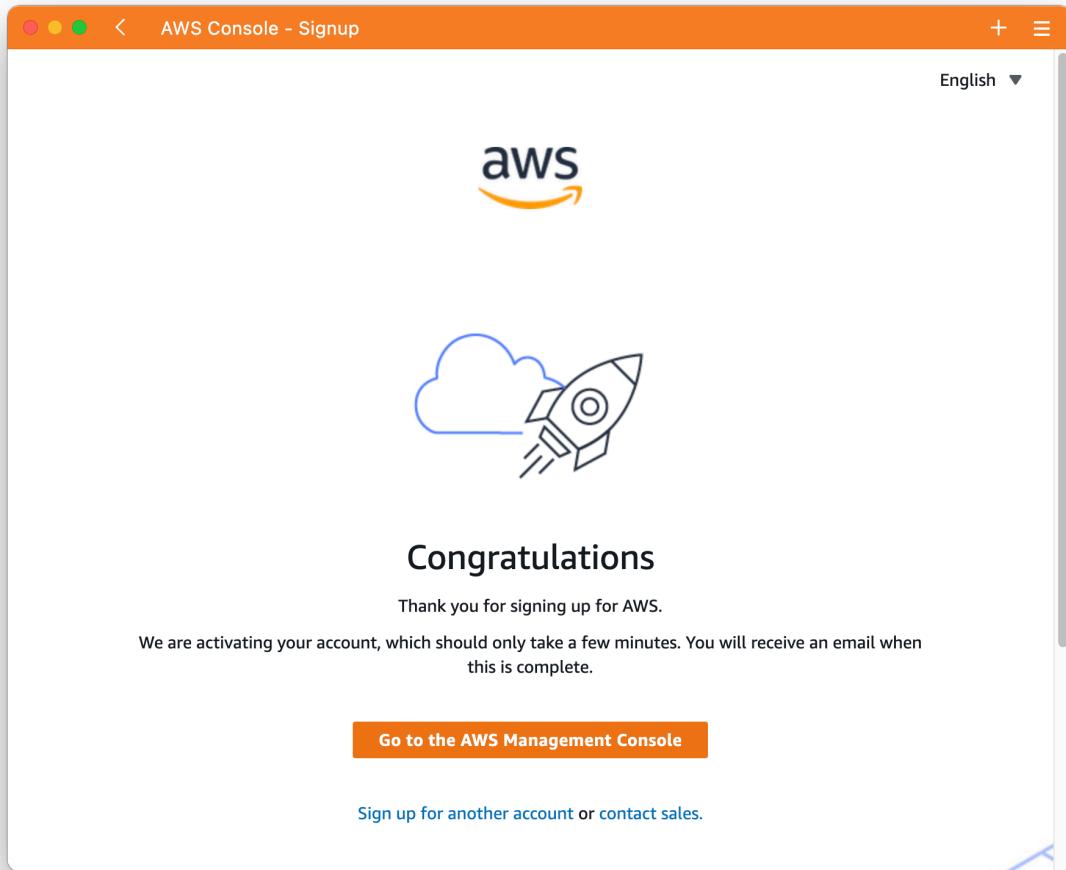


Figure 1.21 Fifth step of creating an AWS account: Success!

1.8.2 Signing In

You now have an AWS account and are ready to sign in to the AWS Management Console. As mentioned earlier, the Management Console is a web-based tool you can use to control AWS resources; it makes most of the functionality of the AWS API available to you. Figure 1.22 shows the sign-in form at console.aws.amazon.com. Choose *Root user* and enter your email address, click Next, and then enter your password to sign in.

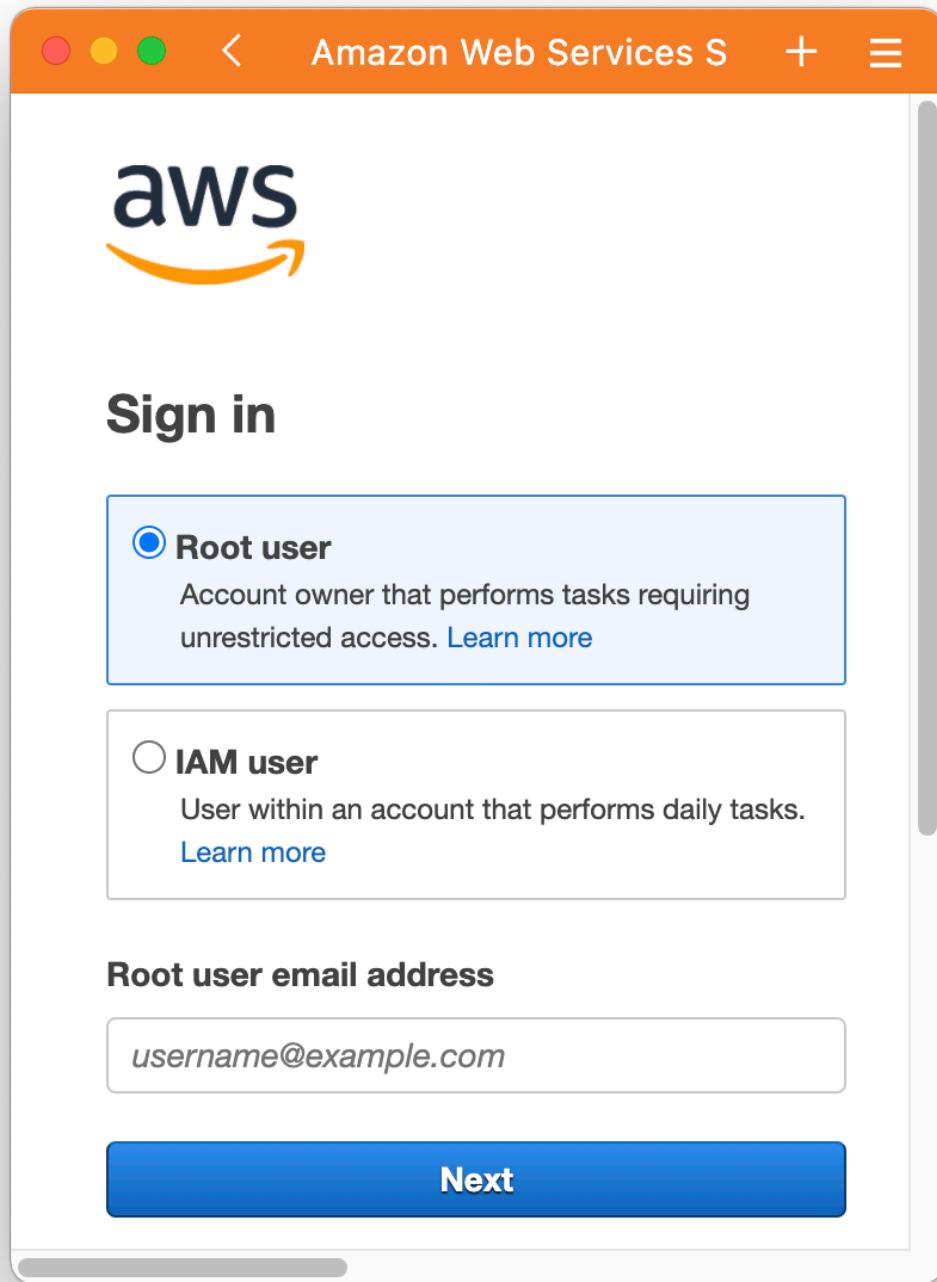


Figure 1.22 Sign in with the AWS account root user

After you have signed in successfully, you are forwarded to the start page of the Management Console, as shown in figure [1.23](#).

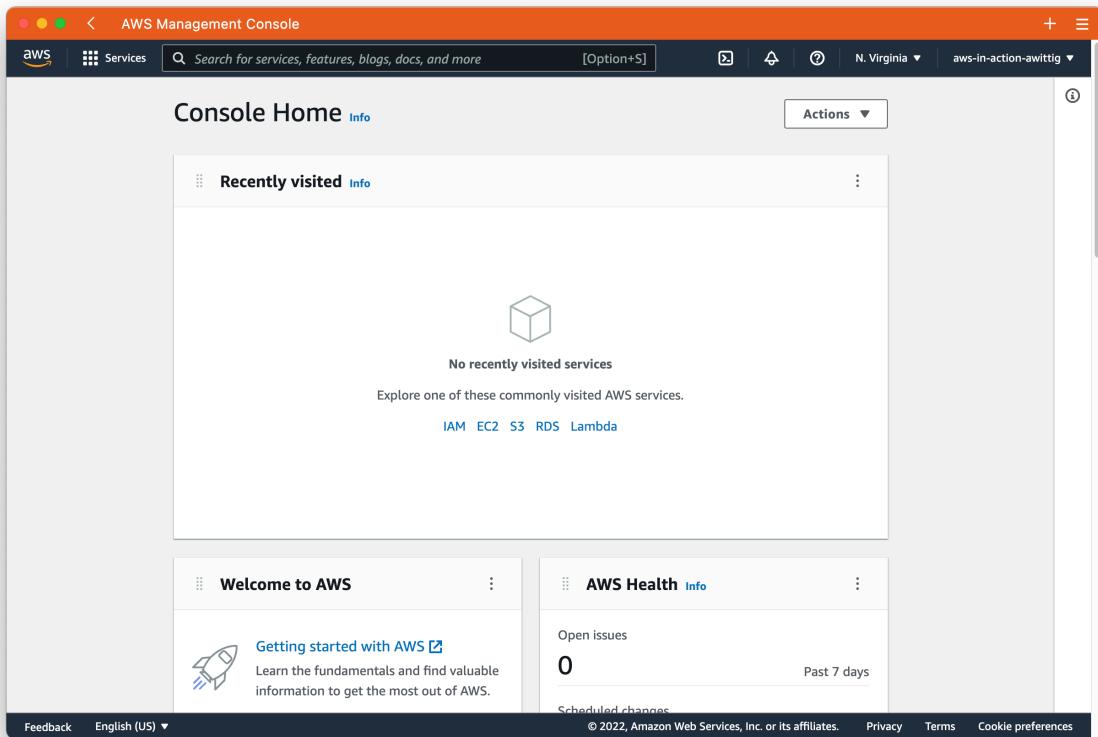


Figure 1.23 The dashboard after logging in for the first time

The most important part is the navigation bar at the top, shown in figure [1.24](#). It consists of seven sections:

- *AWS*—The dashboard of the Management Console shows an overview of your AWS account.
- *Services*—Provides access to all AWS services.
- *Search*—Allows you to search for services, features, and more.
- *Terminal*—Spin up a terminal with access to your cloud resources in the browser.
- *Notifications*—View alerts and notifications by AWS, for example planned downtimes or outages.
- *Help*—Get support by the community, experts, or AWS support.
- *Region*—Select the region you want to manage.
- *Account*—Manage your AWS account.

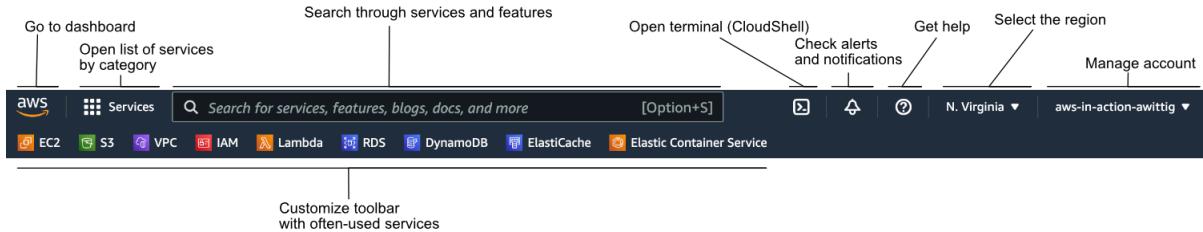


Figure 1.24 Creating a cost budget to monitor incurred and forecasted costs

Next, you'll make sure to avoid unwanted AWS costs.

1.9 Create a budget alert to keep track of your AWS bill

At first, the pay-per-use pricing model of AWS might feel unfamiliar to you, as it is not 100% foreseeable what your bill will look like at the end of the month. Most of the examples in this book are covered by the Free Tier, so AWS won't charge you anything. Exceptions are clearly marked. To provide you with the peace of mind needed to learn about AWS in a comfortable environment, you will create a budget next. The budget will notify you via email in case your AWS bill will exceed \$5 USD so that you can react quickly.

Before creating a budget, configure a Free Tier usage alert.

1. Open the billing preferences of your AWS account at console.aws.amazon.com/billing/home#/preferences.
2. Enable *Receive Free Tier Usage Alerts* and type in your email address as shown in figure [1.25](#).
3. Press the *Save preferences* button.

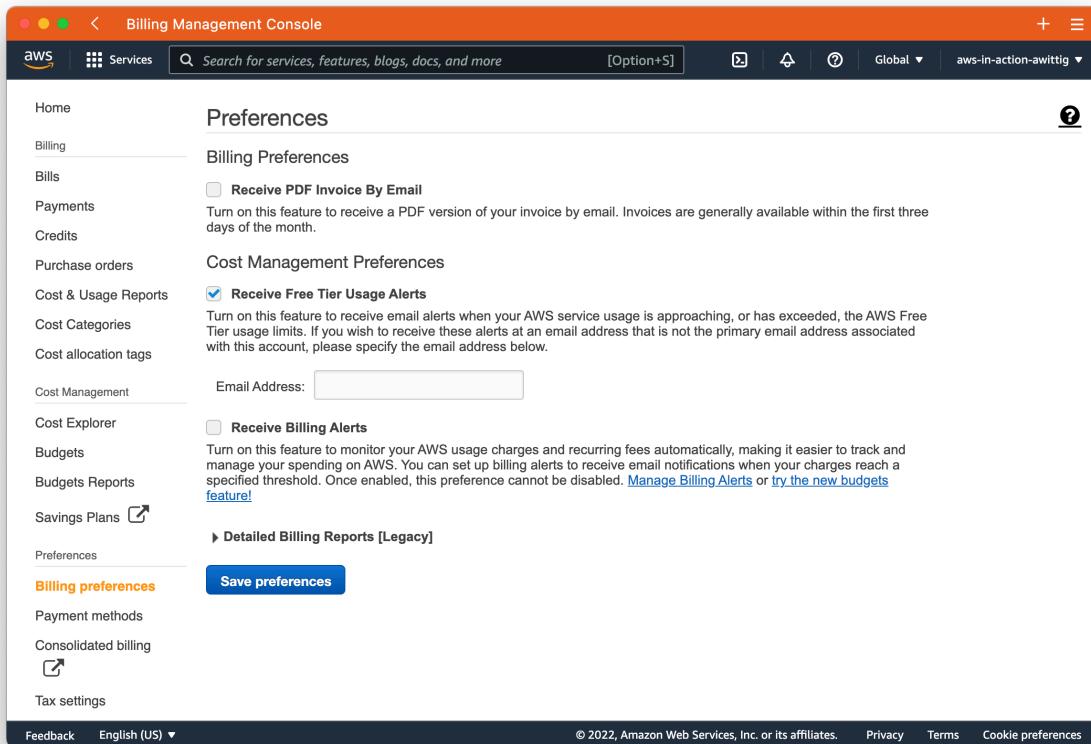


Figure 1.25 Creating a Free Tier usage alert to avoid unwanted costs.

Next, you will create a budget which monitors costs incurred and forecasts costs to the end of the month.

1. Search and open *Budgets* in the Management Console's navigation bar.
2. Click *Create budget*.
3. Select *Cost budget* as shown in figure [1.26](#).
4. Click *Next*.

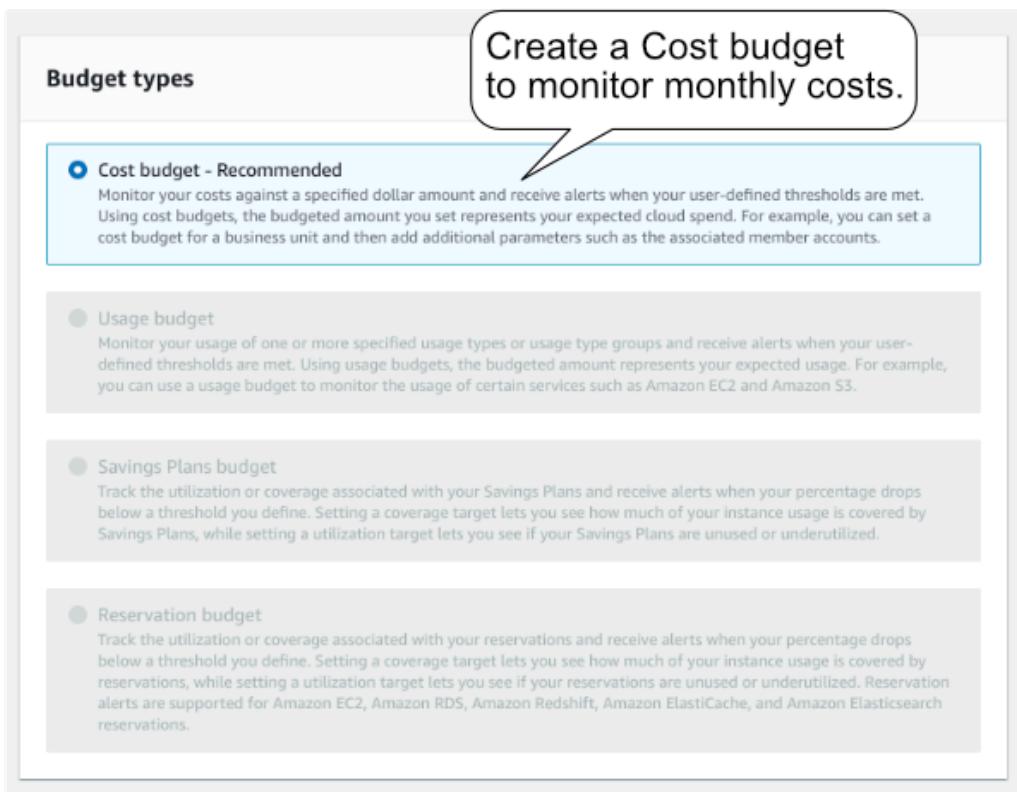


Figure 1.26 Creating a cost budget to monitor incurred and forecasted costs

Next, configure the cost budget as illustrated in figure [1.27](#).

1. Choose period *Monthly*.
2. Select *Recurring budget*.
3. Choose the current month and year as start month.
4. Select *Fixed* to use the same budget for every month of the year.
5. Type in *5* to set the budget to \$5 USD per month.
6. Click *Next*.

Billing Management Console

Billing Console > Budgets > Create budget

Step 1 Choose budget type

Set your budget Info

Because Cost Explorer isn't enabled for this account, you won't be able to view or filter your historical data when creating a budget. After creating a budget, Cost Explorer will automatically be enabled and it can take up to 24 hours to populate all of your spend data. [Learn more](#)

Step 2 Set your budget

Step 3 Configure alerts (Disabled)

Step 4 - Optional Attach actions (Disabled)

Step 5 Review

Set budget amount

Period Daily budgets do not support enabling forecasted alerts, or daily budget planning.

1 **1**

Budget effective date

2 **2** Recurring budget Recurring budgets renew on the first day of every monthly billing period.
 Expiring budget Expiring monthly budgets stop renewing at the end of the selected expiration month.

Start month

3 **3**

Choose how to budget

4 **4** Fixed Create a budget that tracks against a single monthly budgeted amount.
 Monthly budget planning Specify your budgeted amount for each budget period.

Enter your budgeted amount (\$)
Last month's cost: .

5 **5**

Feedback English (US) ▾ © 2022, Amazon Web Services, Inc. or its affiliates. Privacy Terms Cookie preferences

Figure 1.27 Creating a cost budget to monitor incurred and forecasted costs

After defining the budget, it is time to create alerts that will notify you via email.

1. Click *Add alert threshold*.
2. Type in 100 % of budgeted amount.
3. Select trigger 'Actual' to get notified when the incurred costs increase the budget.
4. Type in your email address.
5. Click *Add alert threshold*.
6. Type in 100 % of budgeted amount.
7. Select trigger 'Forecasted' to get notified when the forecasted costs increase the budget.
8. Type in your email address.
9. Click *Next* to proceed.
10. Review the budget and click *Create budget*.

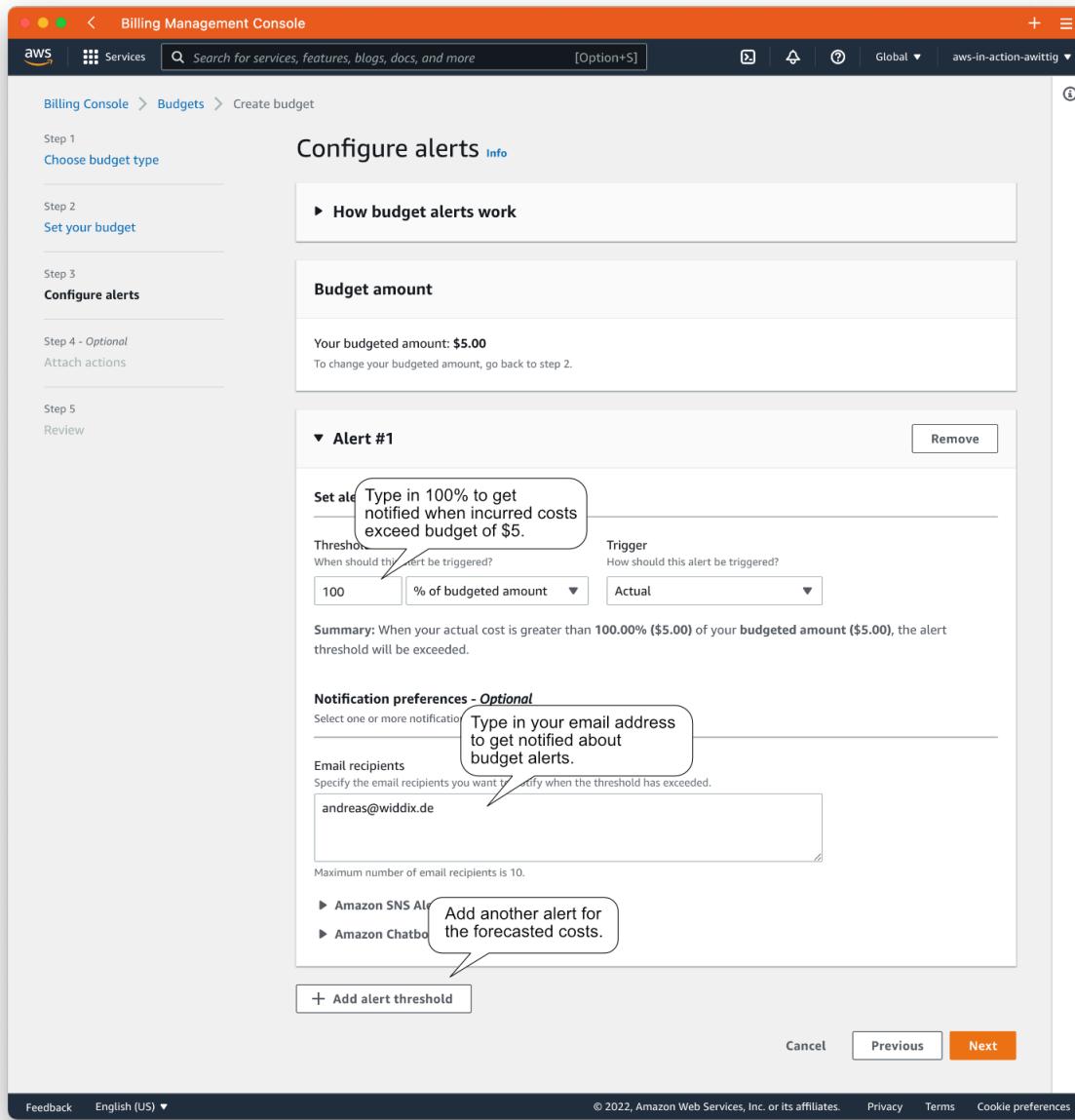


Figure 1.28 Define alarms to get notified when incurred or forecasted costs exceed your monthly budget

That's it, you are ready to get started learning all the services and approaches we cover in the rest of this book. When you forget to shutdown or delete resources when following our examples, AWS will notify in case your monthly bill will exceed \$5 USD.

1.10 Summary

- Cloud computing, or the cloud, is a metaphor for supply and consumption of IT resources.
- Amazon Web Services (AWS) offers Infrastructure as a service (IaaS), Platform as a service (PaaS), and Software as a service (SaaS).
- Amazon Web Services (AWS) is a platform of web services for computing, storing, and networking that work well together.
- Cost savings aren't the only benefit of using AWS. You'll also profit from an innovative and fast-growing platform with flexible capacity, fault-tolerant services, and a worldwide infrastructure.
- Almost any use case can be implemented on AWS, whether it's a widely used web application or a specialized enterprise application with an advanced networking setup.
- Interact with AWS in many different ways. Control the different services by using the web-based user interface, use code to manage AWS programmatically from the command line or SDKs, or use blueprints to set up, modify, or delete your infrastructure on AWS.
- Pay-per-use is the pricing model for AWS services. Computing power, storage, and networking services are billed similarly to electricity.
- To create an AWS account all you need is a telephone number and a credit card.
- Creating budget alerts allows you to keep track of your AWS bill and get notified whenever you exceed the Free Tier.

A simple example: WordPress in fifteen minutes



This chapter covers

- Creating a cloud infrastructure for Wordpress
- Exploring a cloud infrastructure consisting of a load balancer, virtual machines, a database, and a network file system
- Estimating costs of a cloud infrastructure
- Shutting down a cloud infrastructure

Having looked at why AWS is such a great choice to run web applications in the cloud, in this chapter, you'll explore migrating a simple web application to AWS by setting up a sample cloud infrastructure within fifteen minutes. Over the course of the book we will revisit the Wordpress example to understand the concepts in more detail. For example, we will take a look at the relational database in chapter 10 and learn how to add and remove virtual machines based on the current load in chapter 17.

NOTE

The example in this chapter is totally covered by the Free Tier (see section 1.4.1 for details). As long as you don't run this example longer than a few days, you won't pay anything for it. Keep in mind that this applies only if you created a fresh AWS account for this book and there is nothing else going on in your AWS account. Try to complete the chapter within a few days, because you'll clean up your account at the end of the chapter.

Imagine you work for a mid-sized company that runs a blog to attract new software and operations engineers. WordPress is used as the content management system. Around 1,000 people visit the blog daily. You are paying \$250 USD per month for the on-premises infrastructure. This seems expensive to you, particularly because at the moment the blog is suffering from several outages per month.

To leave a good impression on potential candidates, the infrastructure should be highly available, which is defined as an uptime of 99.99%. Therefore, you are evaluating new options to operate WordPress reliably. AWS seems to be a good fit. As a proof-of-concept, you want to evaluate whether a migration is possible. To do so, you need to do the following:

- Set up a highly available infrastructure for WordPress.
- Estimate monthly costs of the infrastructure.
- Come to a decision and delete the infrastructure afterward.

WordPress is written in PHP and uses a MySQL database to store data. Besides that, data like user uploads is stored on disk. Apache is used as the web server to serve the pages. With this information in mind, it's time to map your requirements to AWS services.

2.1 Creating your infrastructure

You'll use five different AWS services to copy the old infrastructure to AWS:

- *Elastic Load Balancing (ELB)*—AWS offers a load balancer as a service. The load balancer distributes traffic to a bunch of virtual machines, and is highly available by default. Requests are routed to virtual machines as long as their health check succeeds. You'll use the Application Load Balancer (ALB) which operates on Layer 7 (HTTP and HTTPS).
- *Elastic Compute Cloud (EC2)*—The EC2 service provides virtual machines. You'll use a Linux machine with an optimized distribution called Amazon Linux to install Apache, PHP, and WordPress. You aren't limited to Amazon Linux; you could also choose Ubuntu, Debian, Red Hat, or Windows. Virtual machines can fail, so you need at least two of them. The load balancer will distribute the traffic between them. In case a virtual machine fails, the load balancer will stop sending traffic to the failed VM, and the remaining VM will need to handle all requests until the failed VM is replaced.
- *Relational Database Service (RDS) for MySQL*—WordPress relies on the popular MySQL database. AWS provides MySQL with its RDS. You choose the database size (storage, CPU, RAM), and RDS takes over operating tasks like creating backups and installing patches and updates. RDS can also provide a highly available MySQL database by replication.
- *Elastic File System (EFS)*—WordPress itself consists of PHP and other application files. User uploads, for example images added to an article, are stored as files as well. By using a network file system, your virtual machines can access these files. EFS provides a scalable, highly available, and durable network filesystem using the NFSv4.1 protocol.
- *Security groups*—Control incoming and outgoing traffic to your virtual machine, your database, or your load balancer with a firewall. For example, use a security group allowing incoming HTTP traffic from the internet to port 80 of the load balancer. Or restrict network access to your database on port 3306 to the virtual machines running your web servers.

Figure 2.1 shows all the parts of the infrastructure in action. Sounds like a lot of stuff to set up, so let's get started!

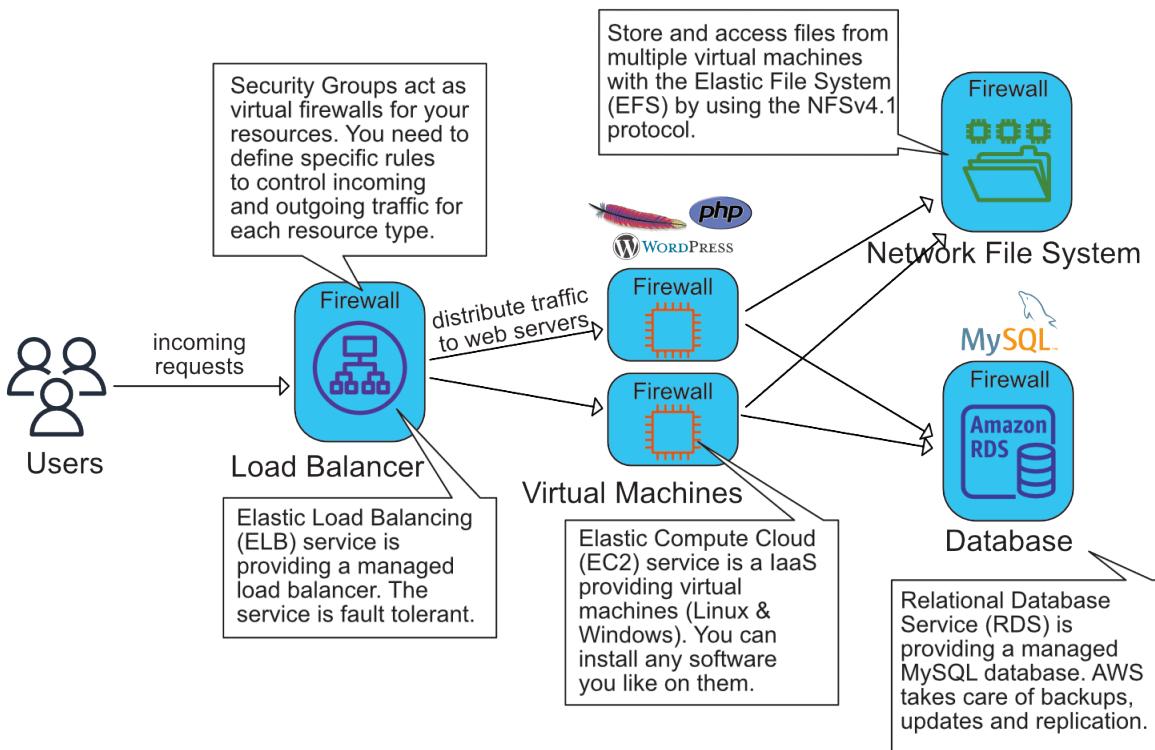


Figure 2.1 The company's blogging infrastructure consists of two load-balanced web servers running WordPress, a network filesystem, and a MySQL database server.

If you expect pages of instructions, you'll be happy to know that you can create all that with just a few clicks. Doing so is possible by using a service called AWS CloudFormation that you will learn about in detail in chapter 4. AWS CloudFormation will do all of the following automatically in the background:

1. Create a load balancer (ELB).
2. Create a MySQL database (RDS).
3. Create a network filesystem (EFS).
4. Create and attach firewall rules (security groups).
5. Create two virtual machines running web servers:
 - A. Create two virtual machines (EC2).
 - B. Mount the network filesystem (EFS).
 - C. Install Apache and PHP.
 - D. Download and extract the 4.8 release of WordPress.
 - E. Configure WordPress to use the created MySQL database (RDS).
 - F. Start the Apache web server.

To create the infrastructure for your proof-of-concept, open the AWS Management Console at console.aws.amazon.com. Click *Services* in the navigation bar, and select *CloudFormation*. You can use the search function to find CloudFormation more easily.

NOTE**Default region for examples**

All examples in this book use N. Virginia (also called us-east-1) as the default region. Exceptions are indicated. Please make sure you switch to the region N. Virginia before starting to work on an example. When using the AWS Management Console, you can check and switch the region on the right side of the main navigation bar at the top.

Click *Create stack* to start the four-step wizard, as shown in figure 2.2. Choose *Template is ready* and enter the Amazon S3 URL s3.amazonaws.com/awsinaction-code3/chapter02/template.yaml to use the template prepared for this chapter. Proceed with the next step of the wizard.

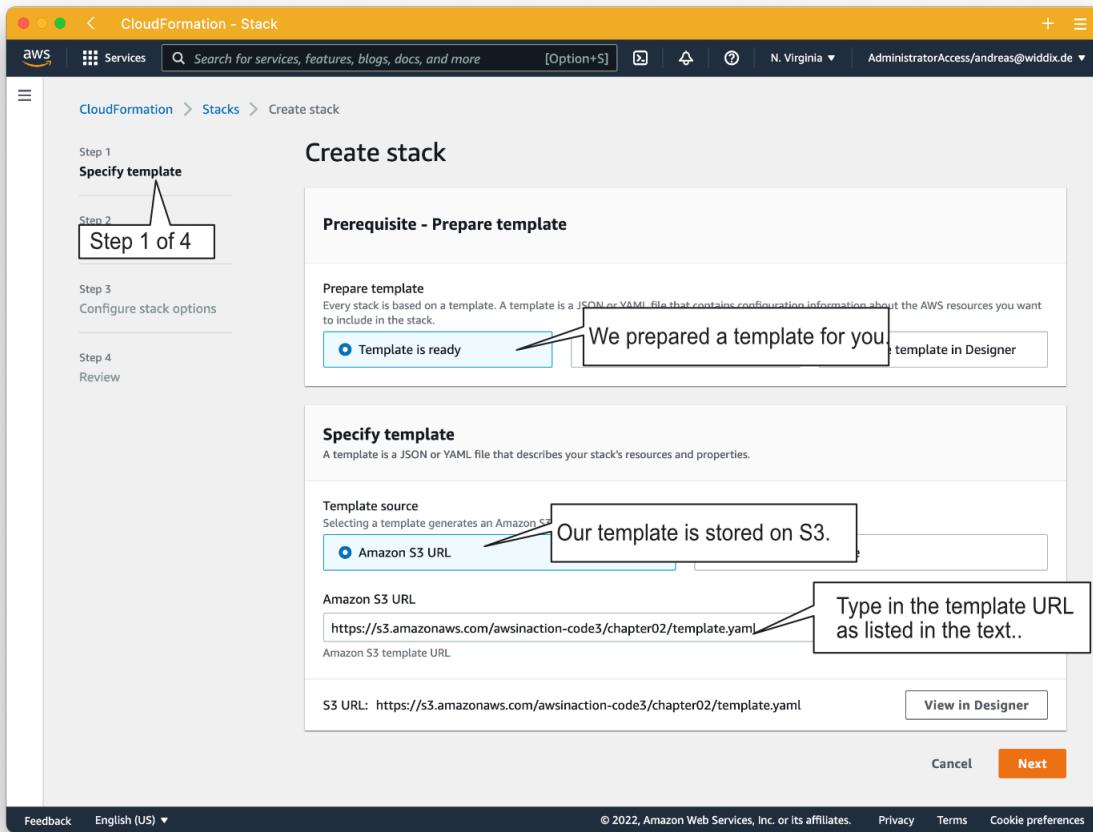


Figure 2.2 Creating the stack for the proof-of-concept: step 1 of 4

Specify `wordpress` as the Stack name and set the `WordpressAdminPassword` to a password of your choice that you are not using somewhere else in the Parameters section as shown in figure 2.3.

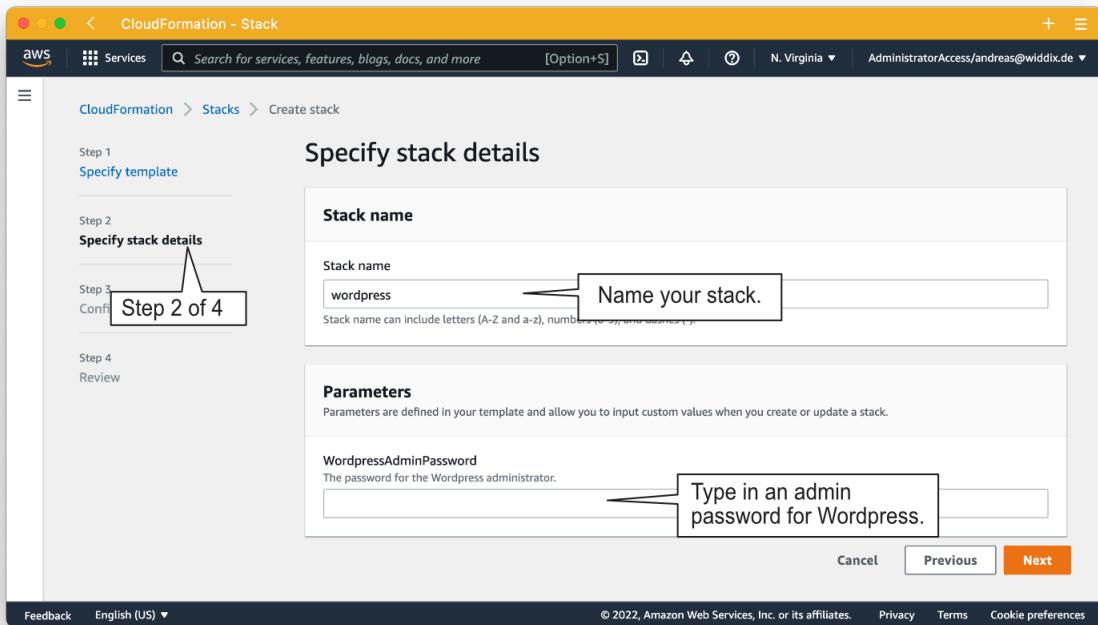


Figure 2.3 Creating the stack for the proof-of-concept: step 2 of 4

The next step is to specify tags for your infrastructure, as illustrated by figure 2.4. A *tag* consists of a key and a value, and can be used to add metadata to all parts of your infrastructure. You can use tags to differentiate between testing and production resources, add a cost center to easily track costs in your organization, or mark resources that belong to a certain application if you host multiple applications in the same AWS account.

Figure 2.4 shows how to configure the tag. In this example, you'll use a tag to mark all resources that belong to the `wordpress` system. This will help you to easily find all the parts of your infrastructure later. Use a custom tag consisting of the key `system` and the value `wordpress`. Afterward, press the Next button to proceed to the next step.

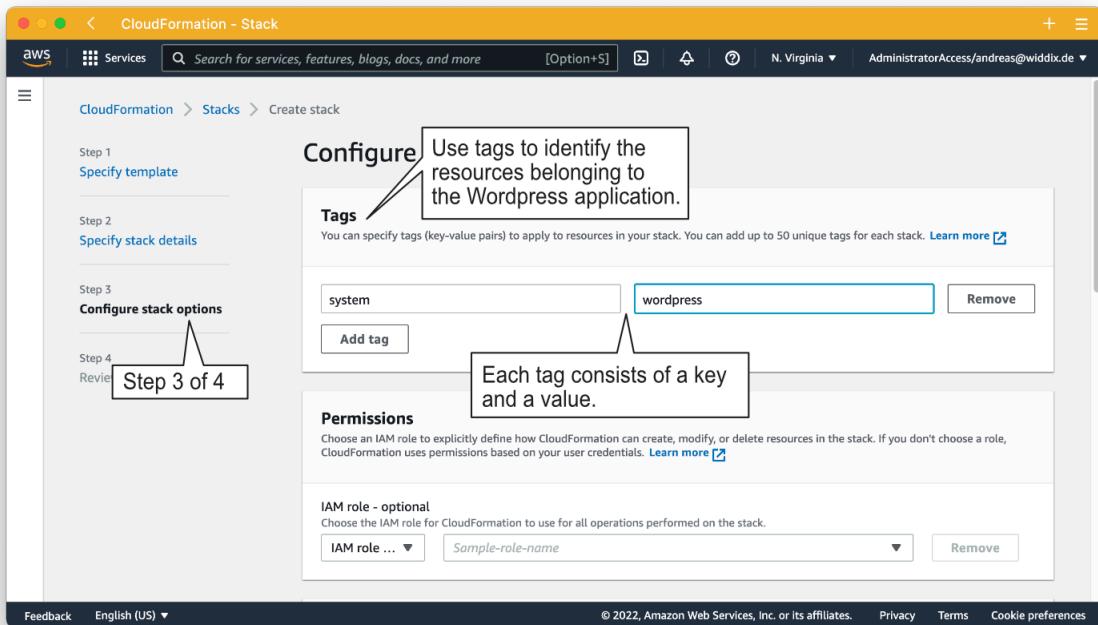


Figure 2.4 Creating the stack for the proof-of-concept: step 3 of 4

You can define your own tags as long as the key name is shorter than 128 characters and the value has fewer than 256 characters.

SIDEBAR

Additional CloudFormation Stack options

It is possible to define specific permissions used to manage resources, as well as to set up notifications and other advanced options. You won't need these options for 99% of the use cases, so we don't cover them in our book. Have a look at the CloudFormation User Guide (docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/cfn-console-add-tags.html) if you're interested in the details.

Figure 2.5 illustrates that all you need to do is to acknowledge that CloudFormation will create IAM resources and click *Create stack*.

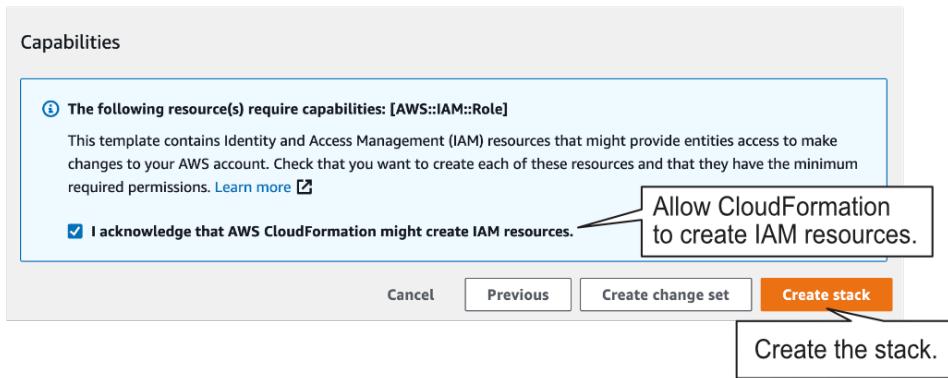


Figure 2.5 Creating the stack for the proof-of-concept: step 4 of 4

Your infrastructure will now be created. Figure 2.6 shows that `wordpress` is in state `CREATE_IN_PROGRESS`. It's a good time to take a break; come back in fifteen minutes, and you'll be surprised.

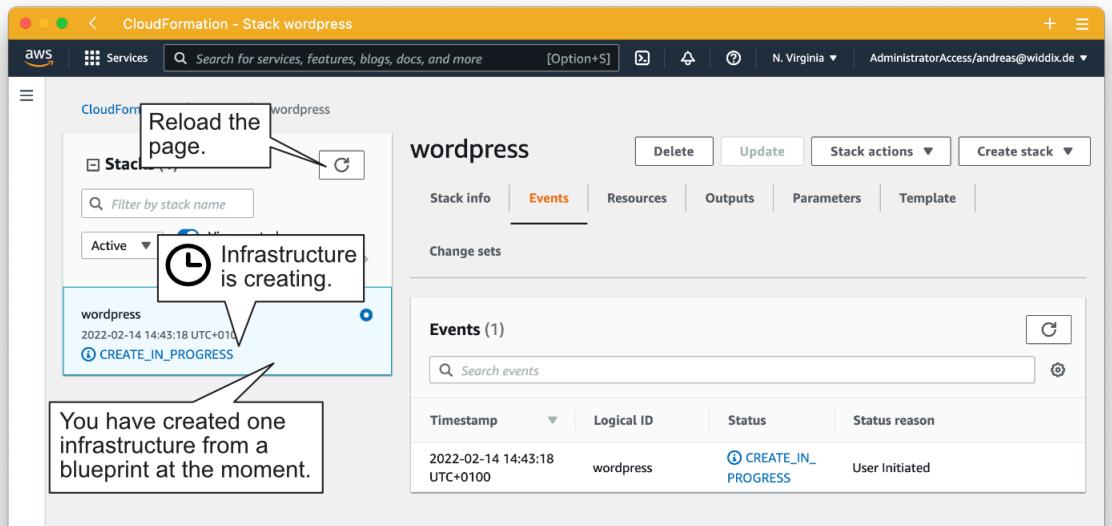


Figure 2.6 CloudFormation is creating the resources needed for WordPress.

After all the needed resources have been created, the status will change to `CREATE_COMPLETE`. Be patient and hit the refresh button from time to time if your status still shows as `CREATE_IN_PROGRESS`.

Select the check box at the beginning of the row containing your `wordpress` stack. Switch to the *Outputs* tab, as shown in figure 2.7. There you'll find the URL to your WordPress installation; click the link to open it in your browser.

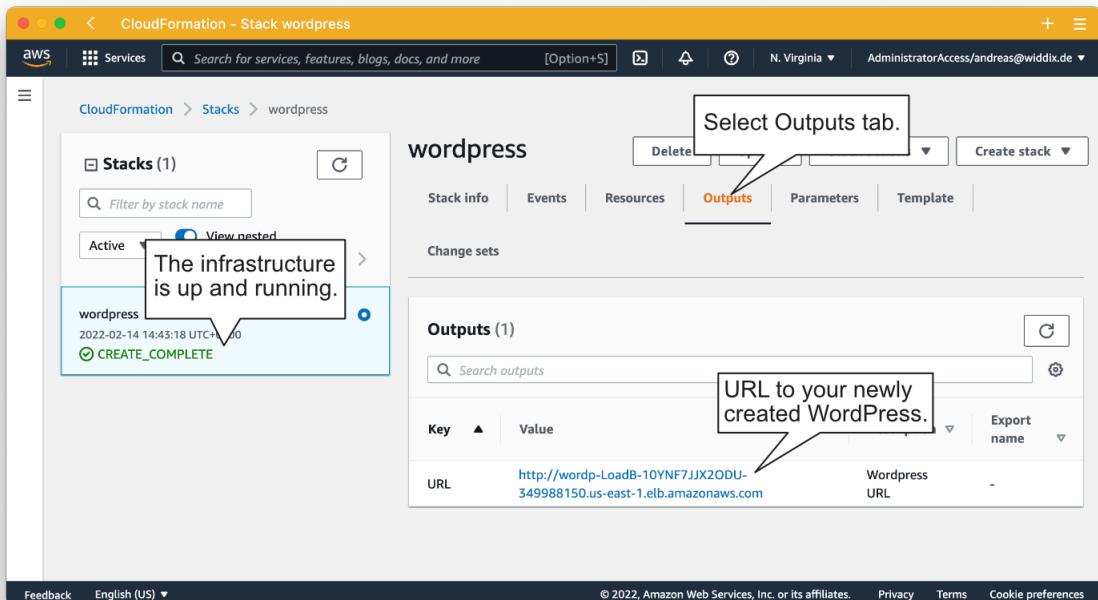


Figure 2.7 Blogging infrastructure has been created successfully.

You may ask yourself, how does this work? The answer is *automation*.

SIDEBAR

Automation references

One of the key concepts of AWS is automation. You can automate everything. In the background, your cloud infrastructure was created based on a blueprint. You'll learn more about blueprints and the concept of programming your infrastructure in chapter 4. You'll learn to automate the deployment of software in chapter 15.

You'll explore the blogging infrastructure in the next section to get a better understanding of the services you're using.

2.2 Exploring your infrastructure

Now that you've created your blogging infrastructure, let's take a closer look at it. Your infrastructure consists of the following:

- Web servers running on virtual machines
- Load balancer
- MySQL database
- Network filesystem

2.2.1 Virtual machines

First, use the navigation bar to open the EC2 service as shown in figure 2.8. Next, select *Instances* from the sub navigation. A list showing two virtual machines named `wordpress` shows up. When you select one of those instances, details about the virtual machine appear below.

Name	Instance ID	Instance state	Instance type	Status check	Alarm status
<input checked="" type="checkbox"/> wordpress	i-0afb2cc61eaf8c3df	Running	t2.micro	2/2 checks passed	No alarms
<input type="checkbox"/> wordpress	i-01e9a24916cc6dd8d	Running	t2.micro	2/2 checks passed	No alarms

Figure 2.8 The virtual machines are running a web server to deliver Wordpress

You're now looking at the details of your virtual machine, also called an EC2 instance. Some interesting details are as follows:

- *Instance ID*—The ID of the virtual machine.
- *Instance type* the size of the virtual machine (CPU and memory).
- *IPv4 Public IP*—The IP address that is reachable over the internet.
- *AMI ID*—Remember that you used the Amazon Linux OS. If you click the AMI ID, you'll see the version number of the OS, among other things.

Select the *Monitoring* tab to see how your virtual machine is utilized. This tab is essential if you really want to know how your infrastructure is doing. AWS collects some metrics and shows

them here. For example, if the CPU is utilized more than 80%, it might be a good time to add another virtual machine to prevent increasing response times. You will learn more about monitoring virtual machines in section 3.2.

2.2.2 Load balancer

Next, have a look at the load balancer, which are part of the EC2 service as well. So there is no need to switch to a different service in the Management Console. Just click *Load Balancer* in the sub navigation on the lefthand side.

Select your load balancer from the list to show more details. Your *internet-facing* load balancer is accessible from the internet via an automatically generated DNS name.

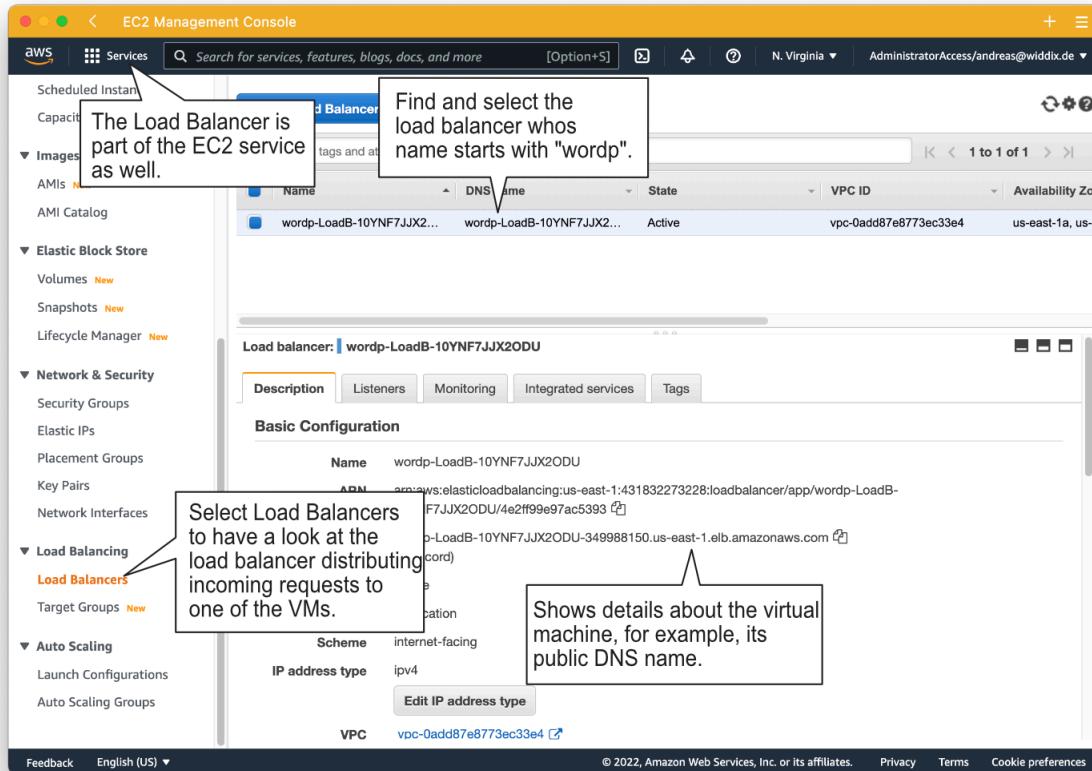


Figure 2.9 Get to know the load balancer

The load balancer forwards an incoming request to one of your virtual machines. A target group is used to define the targets for a load balancer. You'll find your target group after switching to *Target Groups* through the sub navigation of the EC2 service as shown in figure 2.10.

The load balancer performs health checks to ensure requests are routed to healthy targets only. Two virtual machines are listed as targets for the target group. As you can see in the figure, the status of both virtual machines is healthy.

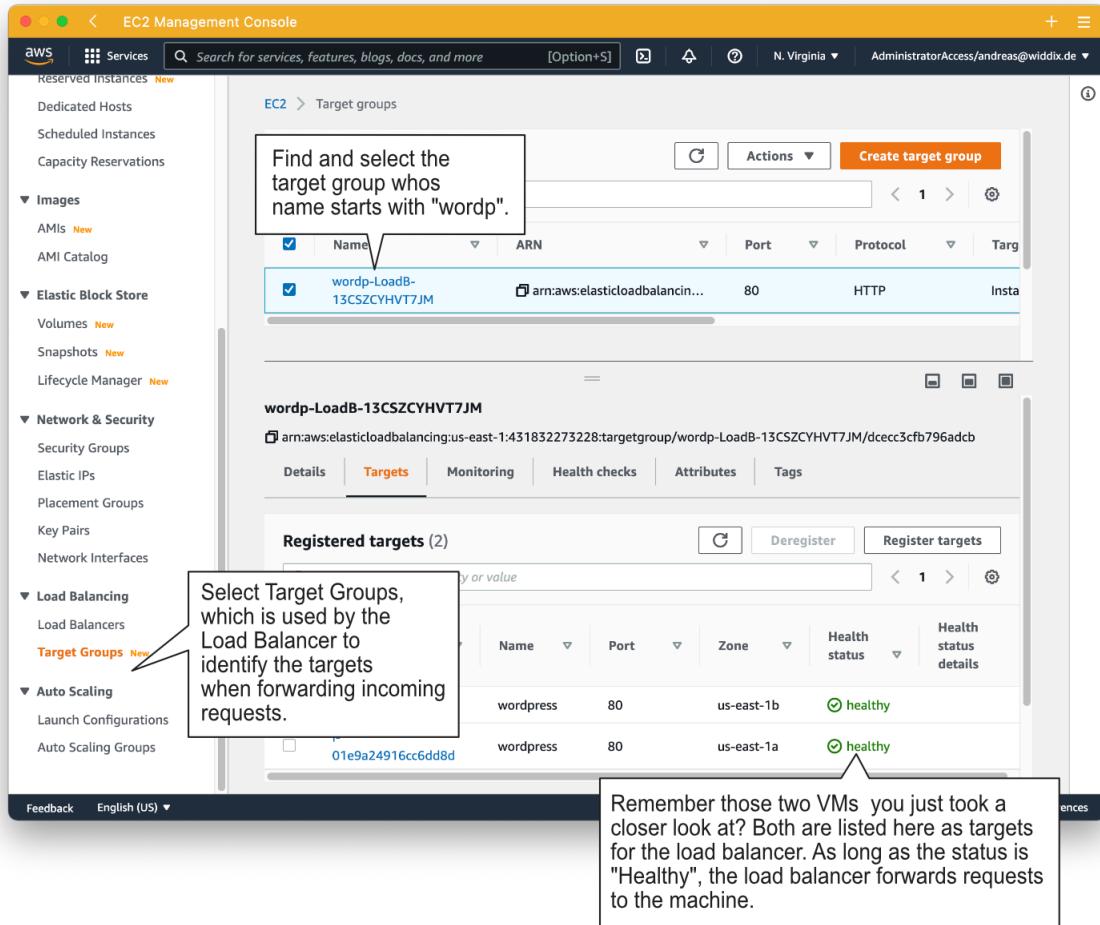


Figure 2.10 Details of target group belonging to the load balancer

As before, there is a Monitoring tab where you can find interesting metrics that you should watch in production. If the traffic pattern changes suddenly, this indicates a potential problem with your system. You'll also find metrics indicating the number of HTTP errors, which will help you to monitor and debug your system.

2.2.3 MySQL database

The MySQL database is an important part of your infrastructure; you'll look at it next. To do so, open the Relational Database Service (RDS) via the main navigation. Afterwards, select *Databases* from the sub navigation. Select the database using engine *MySQL community* as illustrated in figure [2.11](#).

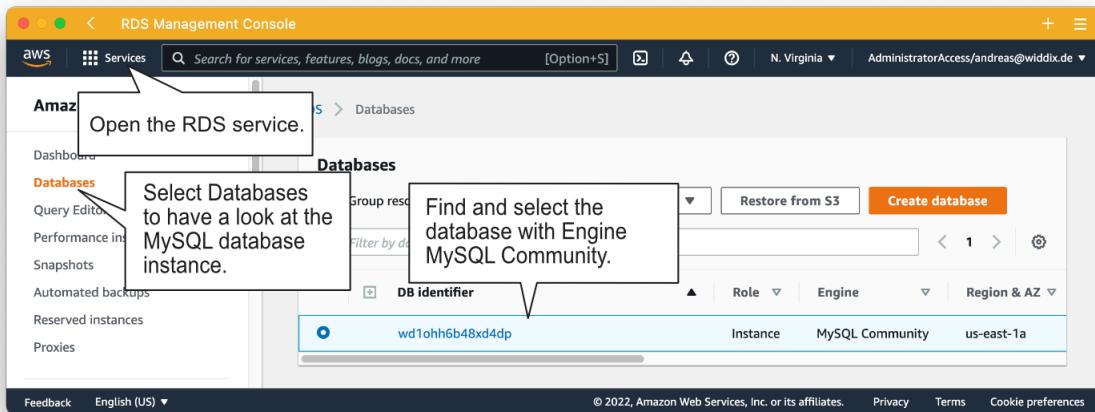


Figure 2.11 Finding the MySQL database

WordPress requires a MySQL database, so you have launched a database instance with the MySQL engine as noted in figure 2.12. Your blog receives a low amount of traffic, so the database doesn't need to be very powerful. A small instance class with a single virtual CPU and 1 GB memory is sufficient. Instead of using SSD storage, you are using magnetic disks, which is cheaper and sufficient for a web application with around 1,000 visitors per day.

As you'll see in chapter 8, other database engines, such as PostgreSQL or Oracle Database, are available as well as more powerful instance classes, offering up to 96 cores with 768 GB memory, for example.

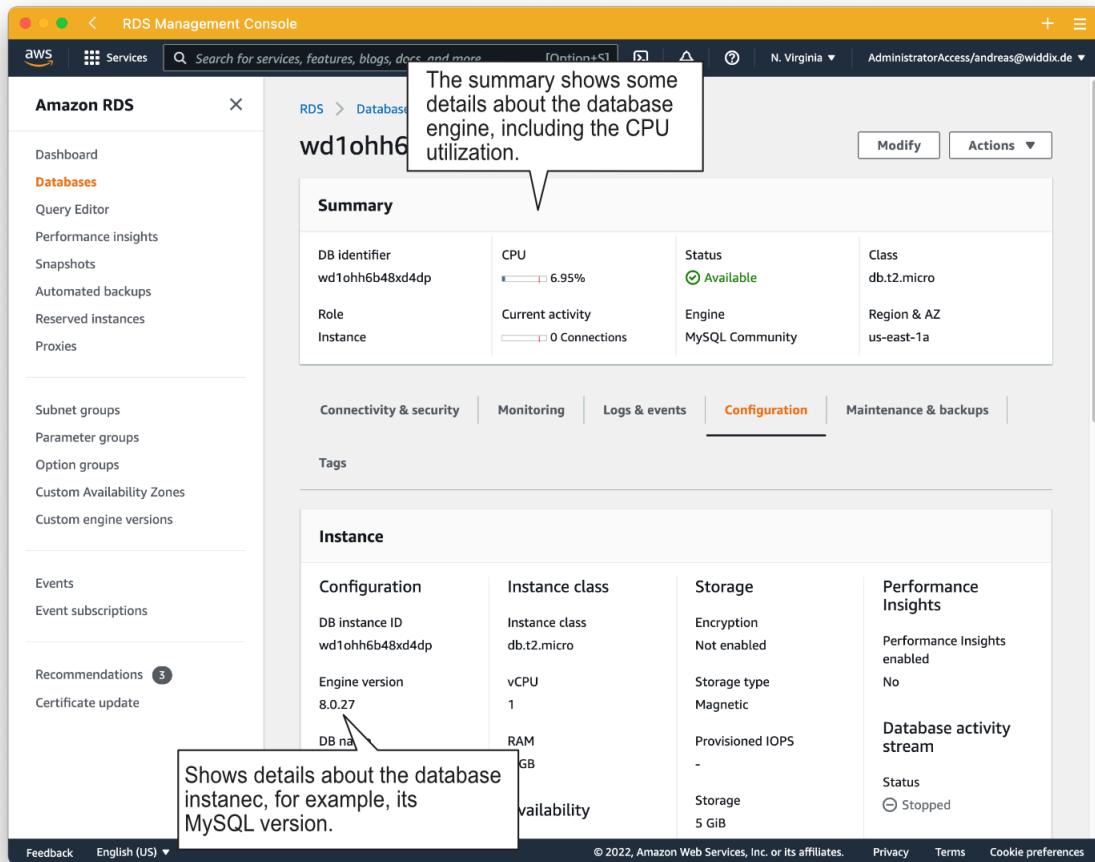


Figure 2.12 Details of the MySQL database storing data for the blogging infrastructure

Common web applications use a database to store and query data. That is true for WordPress as well. The Content Management System (CMS) stores blog posts, comments, and more within a MySQL database.

But WordPress also stores data outside the database on disk. For example, if an author uploads an image for their blog post, the file is stored on disk. The same is true when you are installing plug-ins and themes as an administrator.

2.2.4 Network filesystem

The Elastic File System (EFS) is used to store files and access them from multiple virtual machines. EFS is a storage service accessible through the NFS protocol. To keep things simple, all files that belong to WordPress are stored on EFS so they can be accessed from all virtual machines. This includes PHP, HTML, CSS, and PNG.

Open the EFS service via the main navigation as shown in figure 2.13. Next, select the file system whose name starts with `wordpress`.

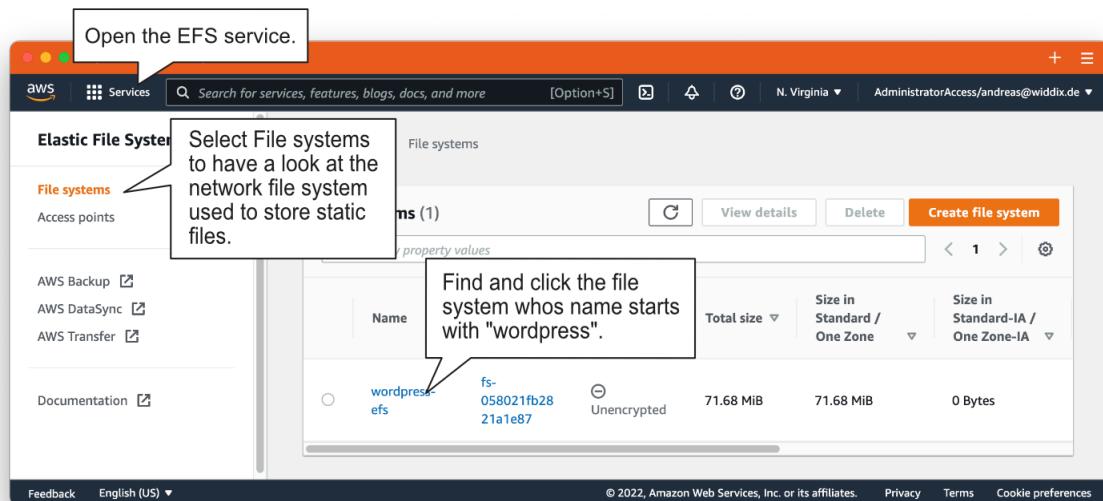


Figure 2.13 NFS used to store the WordPress application and user uploads

Figure 2.14 shows the details of the file system. For example, the throughput mode *bursting* is used, to allow high I/O throughput for small periods of time during the day at low costs.

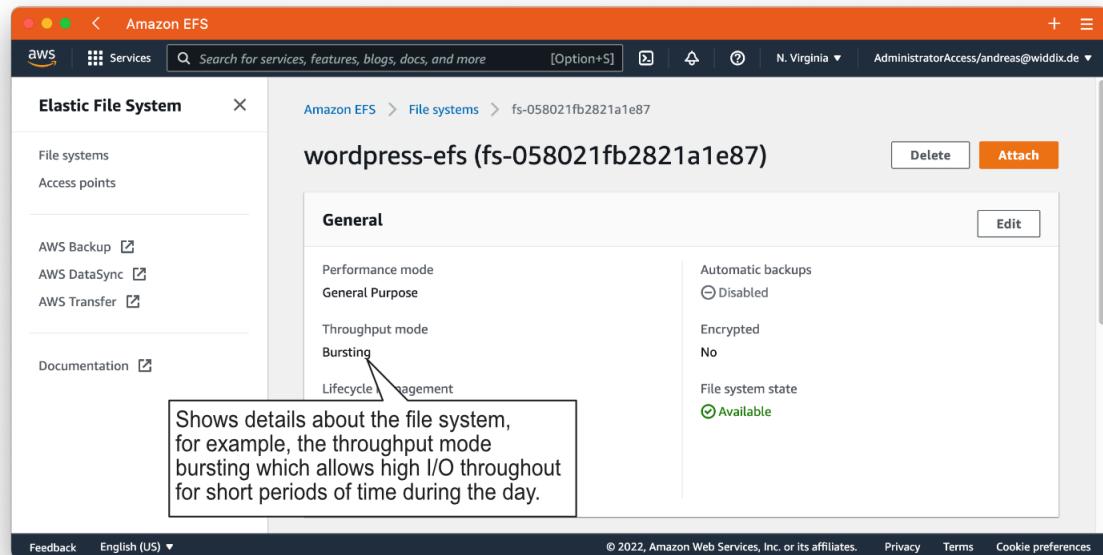


Figure 2.14 The details of the EFS

To mount the Elastic File System from a virtual machine, mount targets are needed. You should use two mount targets for fault tolerance. The network filesystem is accessible using a DNS name for the virtual machines.

Now it's time to evaluate costs. You'll analyze the costs of your blogging infrastructure in the next section.

2.3 How much does it cost?

Part of evaluating AWS is estimating costs. We recommend using the AWS Pricing Calculator to do so. We created a cost estimation including the load balancer, the virtual machines, the database, and the network file system. Go to <calculator.aws/#/estimate?id=e4dc37223e61b801719902cdb9b4421ea6790ffc> to check out the results which is also shown in figure 2.15. We estimate costs of about \$75 for hosting Wordpress on AWS in a highly available manner, which means all components are distributed among at least two different data centers. Don't worry, the example in this chapter is covered by the Free Tier. Our cost estimation does not consider the Free Tier, as it only applies for the first 12 months.

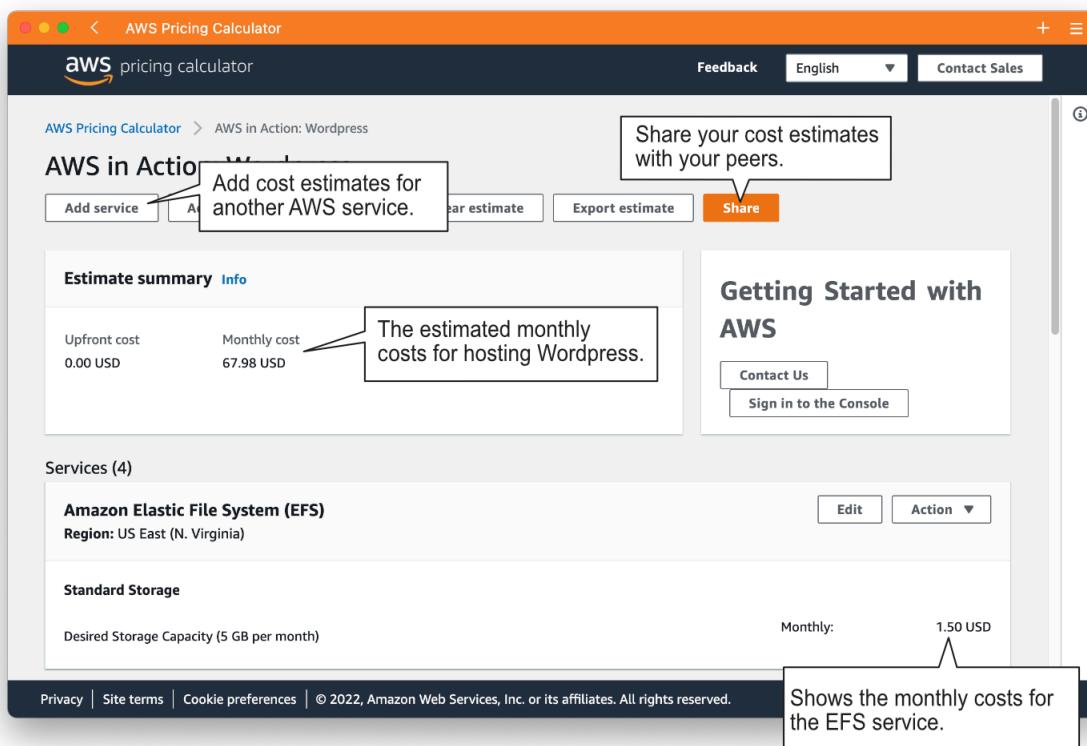


Figure 2.15 Blogging infrastructure cost calculation

Table 2.1 summarizes the results from the AWS Pricing Calculator.

Table 2.1 More detailed cost calculation for blogging infrastructure

AWS service	Infrastructure	Pricing	Monthly cost
EC2	Virtual machines	$2 * 730 \text{ hours} * \$0.0116 (\text{t2.micro})$	\$16.94
EC2	Storage	$2 * 8 \text{ GB} * \$0.10 \text{ per month}$	\$1.60
Application Load Balancer	Load balancer	$730 \text{ hours} * \$0.0225 (\text{load balancer hour}) 200 \text{ GB per month}$	\$18.03
Application Load Balancer	Outgoing traffic	$100 \text{ GB} * \$0.00 (\text{first 100 GB}) 100 \text{ GB} * \$0.09 (\text{up to 10 TB})$	\$9.00
RDS	MySQL database instance (primary + standby)	$732.5 \text{ hours} * \$0.017 * 2$	\$24.82
RDS	Storage	$5 \text{ GB} * \$0.115 * 2$	\$2.30
EFS	Storage	$5 \text{ GB} * \$0.3$	\$1.50
			\$74.19

Keep in mind that this is only an estimate. You’re billed based on actual use at the end of the month. Everything is on-demand and usually billed by seconds or gigabyte of usage. But what factors might influence how much you actually use this infrastructure?

- *Traffic processed by the load balancer*—Expect costs to go down in December and in the summer when people are on vacation and not looking at your blog.
- *Storage needed for the database*—If your company increases the amount of content in your blog, the database will grow, so the cost of storage will increase.
- *Storage needed on the NFS*—User uploads, plug-ins, and themes increase the amount of storage needed on the NFS, which will also increase costs.
- *Number of virtual machines needed*—Virtual machines are billed by seconds of usage. If two virtual machines aren’t enough to handle all the traffic during the day, you may need a third machine. In that case, you’ll consume more seconds of virtual machines.

Estimating the cost of your infrastructure is a complicated task, but that is also true if your infrastructure doesn’t run in AWS. The benefit of using AWS is that it’s flexible. If your estimated number of virtual machines is too high, you can get rid of a machine and stop paying for it. You will learn more about the pricing model of the different AWS services during the course of this book.

You have completed the proof-of-concept for migrating your company’s blog to AWS. It’s time to shut down the infrastructure and complete your migration evaluation.

2.4 Deleting your infrastructure

Your evaluation has confirmed that you can migrate the infrastructure needed for the company’s blog to AWS from a technical standpoint. You have estimated that a load balancer, virtual machines, MySQL database, as well as a NFS capable of serving 1,000 people visiting the blog per day will cost you around \$75 USD per month on AWS. That is all you need to come to a decision.

Because the infrastructure does not contain any important data and you have finished your evaluation, you can delete all the resources and stop paying for them.

Go to the CloudFormation service in the Management Console, and take the following steps as shown in figure 2.16:

1. Select your *wordpress* stack.
2. Click the *Delete* button.

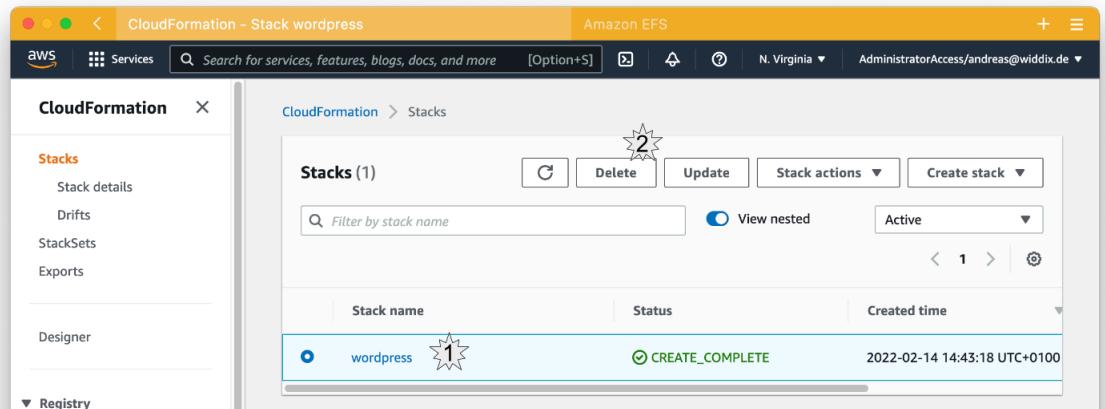


Figure 2.16 Delete your blogging infrastructure.

After you confirm the deletion of the infrastructure, as shown in figure 2.17, it takes a few minutes for AWS to delete all of the infrastructure's dependencies.

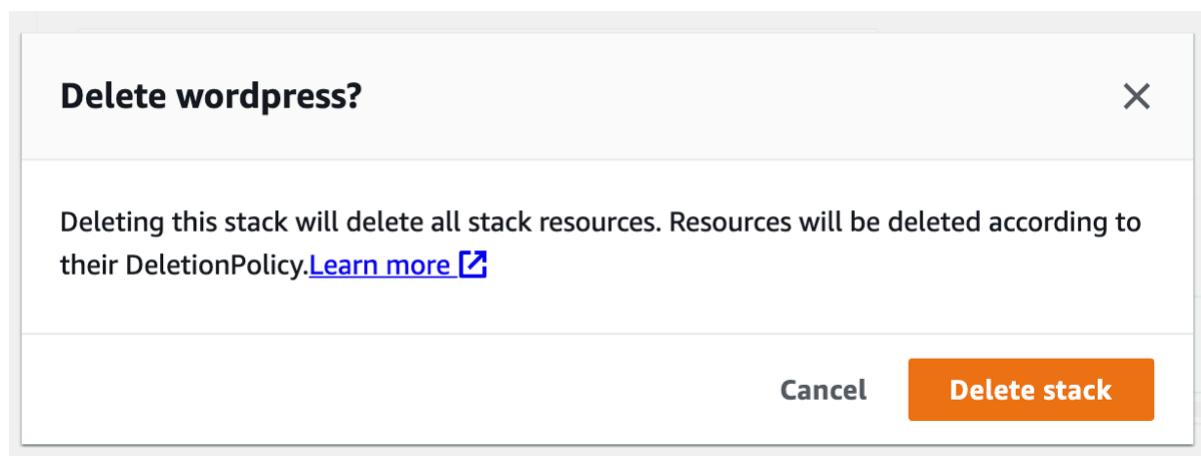


Figure 2.17 Confirming deletion of your blogging infrastructure

This is an efficient way to manage your infrastructure. Just as the infrastructure's creation was automated, its deletion is also. You can create and delete infrastructure on-demand whenever you like. You only pay for infrastructure when you create and run it.

2.5 Summary

- Creating a cloud infrastructure for Wordpress and any other application can be fully automated.
- AWS CloudFormation is a tool provided by AWS for free. It allows you to automate managing your cloud infrastructure.
- The infrastructure for a web application like Wordpress can be created at any time on-demand, without any up-front commitment for how long you'll use it.
- You pay for your infrastructure based on usage. For example, you are paying for a virtual machine per second of usage.
- The infrastructure required to run Wordpress consists of several parts, such as virtual machines, load balancers, databases, and network file systems.
- The whole infrastructure can be deleted with one click. The process is powered by automation.

Using virtual machines: EC2



This chapter covers

- Launching a virtual machine with Linux
- Controlling a virtual machine remotely via Session Manager and SSH
- Monitoring and debugging a virtual machine
- Saving costs for virtual machines

It's impressive what you can achieve with the computing power of the smartphone in your pocket or the laptop in your bag. But if your task requires massive computing power or high network traffic, or needs to run reliably 24/7, a virtual machine is a better fit. With a virtual machine, you get access to a slice of a physical machine located in a data center. On AWS, virtual machines are offered by the service called Elastic Compute Cloud (EC2).

SIDE BAR

Not all examples are covered by Free Tier

The examples in this chapter are not all covered by the Free Tier. A special warning message appears when an example incurs costs. As for the other examples, as long as you don't run them longer than a few days, you won't pay anything for them. Keep in mind that this applies only if you created a fresh AWS account for this book and nothing else is going on in your AWS account. Try to complete the chapter within a few days; you'll clean up your account at the end.

In this chapter you will learn how to launch and manage a virtual machine on AWS. Also, we will show you how to connect to a virtual machine to install or configure applications. On top of

that, you will learn how to monitor a virtual machine. Last but not least, we will introduce the different pricing options of the Elastic Compute Cloud (EC2) making sure you get the most computing power for your money.

3.1 Exploring a virtual machine

A virtual machine (VM) runs on a physical machine isolated from other virtual machines by the hypervisor; it consists of CPUs, memory, networking interfaces, and storage. The physical machine is called the *host machine*, and the VMs running on it are called *guests*. A *hypervisor* is responsible for isolating the guests from each other and for scheduling requests to the hardware, by providing a virtual hardware platform to the guest system. Figure [Figure 3.1](#) shows these layers of virtualization.

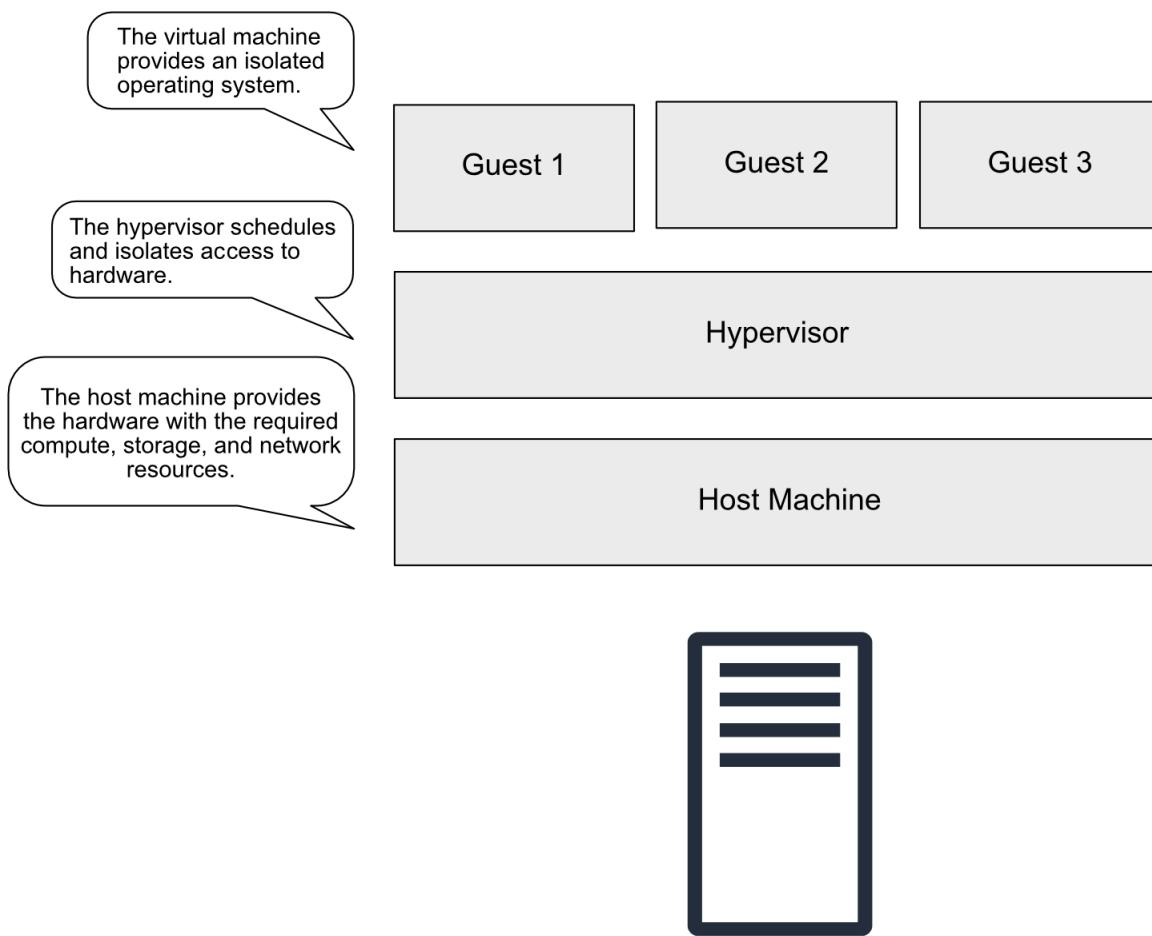


Figure 3.1 Layers of virtualization

Typical use cases for a virtual machine are as follows:

- Hosting a web application such as WordPress
- Operating an enterprise application, such as an ERP application
- Transforming or analyzing data, such as encoding video files

3.1.1 Launching a virtual machine

In the following example, you will launch a virtual machine to run a tool called *linkchecker* that checks a website for broken links. Checking for links resulting in 404 Not Found errors improves the usability and SEO score of your website. You could run the *linkchecker* on your local machine as well, but an EC2 instance in Amazon's data center offers more compute and networking capacities.

It takes only a few clicks to launch a virtual machine, which AWS calls an EC2 instance:

1. Open the AWS Management Console at console.aws.amazon.com.
2. Make sure you're in the N. Virginia (US East) region (see figure [Figure 3.2](#)), because we optimized our examples for this region.
3. Click on *Services* and search for *EC2*.
4. Click *Launch instance* to start the wizard for launching a virtual machine as shown in figure [Figure 3.2](#).

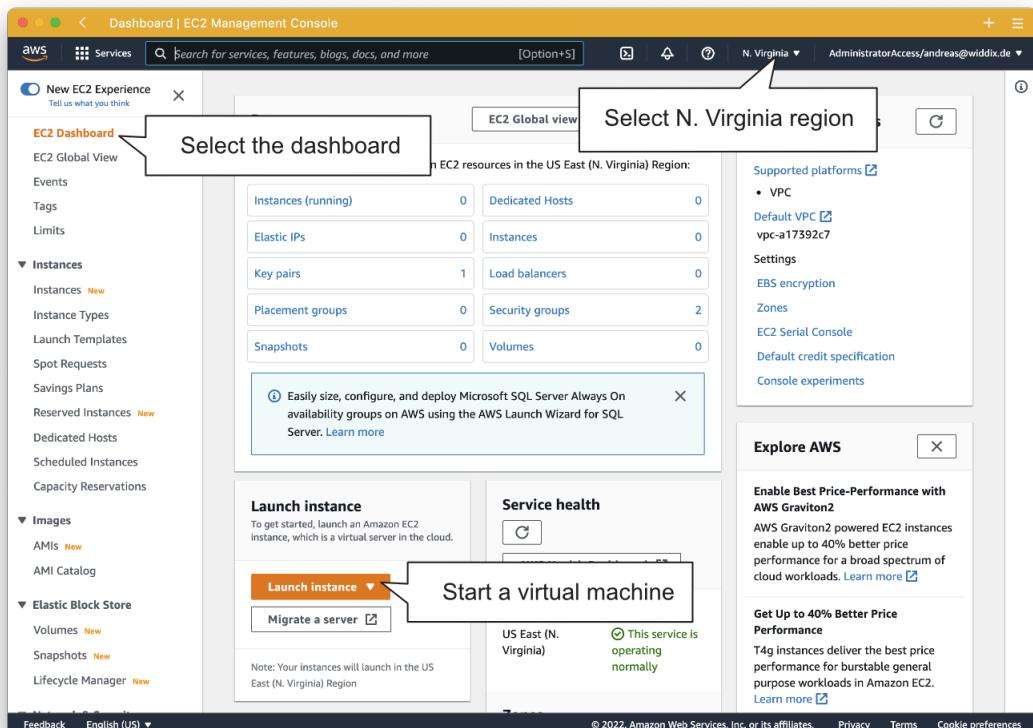


Figure 3.2 The EC2 Dashboard gives you an overview of all parts of the service.

A form will show up, guiding you through all the details needed to create a virtual machine:

1. Naming the virtual machine
2. Selecting the operating system (OS)
3. Choosing the size of your virtual machine
4. Configuring details
5. Adding storage
6. Configuring a firewall
7. Granting the virtual machine permissions to access other AWS services

NAMING THE VIRTUAL MACHINE

To make it easy to find your virtual machine later, it is recommended to assign a name to it. That's especially important, when other people have access to the same AWS account. Figure [Figure 3.2](#) shows the details.

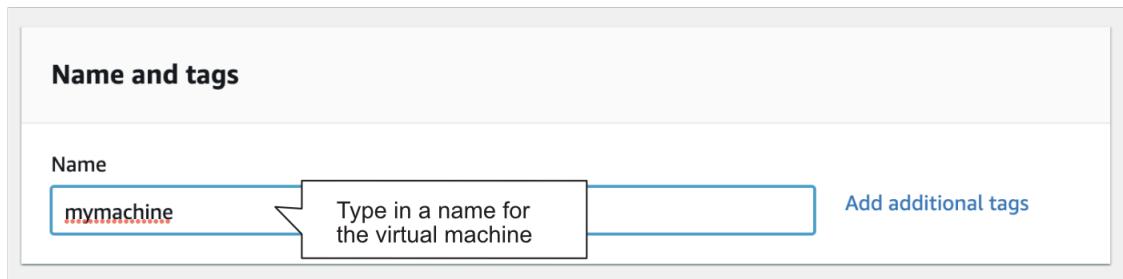


Figure 3.3 Naming the virtual machine

SELECTING THE OPERATING SYSTEM

Next, you need to choose an operating system (OS). In AWS, the OS comes bundled with preinstalled software for your virtual machine; this bundle is called an *Amazon Machine Image (AMI)*. Select *Amazon Linux 2 AMI (HVM)*, as shown in figure [Figure 3.4](#).

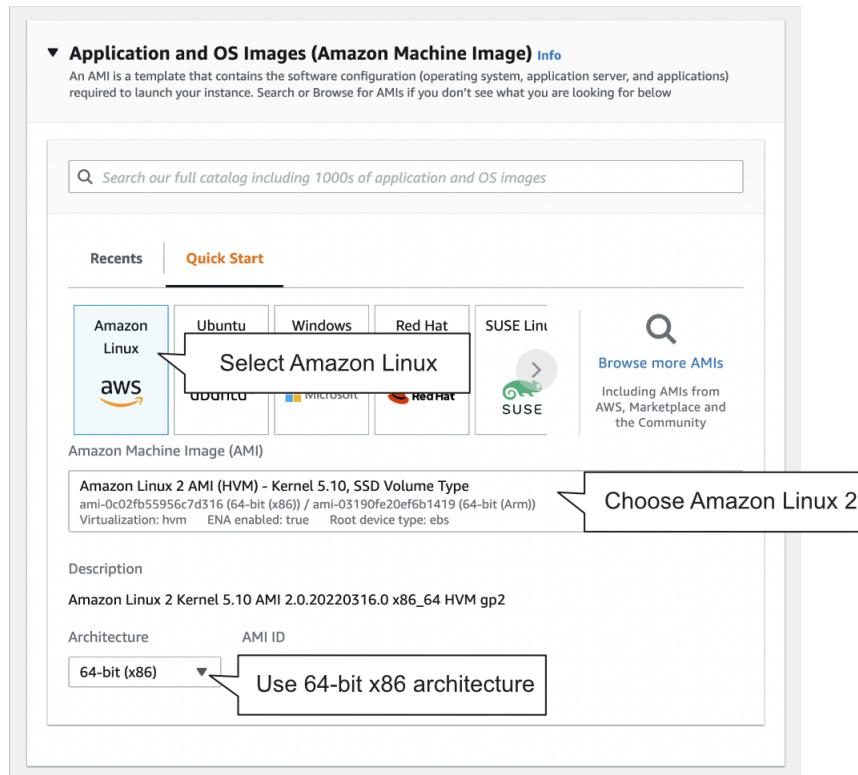


Figure 3.4 Choose the OS for your virtual machine.

The AMI is the basis your virtual machine starts from. AMIs are offered by AWS, third-party providers, and by the community. AWS offers the Amazon Linux AMI, which is based on Red Hat Enterprise Linux and optimized for use with EC2. You'll also find popular Linux distributions and AMIs with Microsoft Windows Server, and you can find more AMIs with preinstalled third-party software in the AWS Marketplace.

When choosing an AMI, start by thinking about the requirements of the application you want to run on the VM. Your knowledge and experience with a specific operating system is another important factor when deciding which AMI to start with. It's also important that you trust the AMI's publisher. We prefer working with Amazon Linux, as it is maintained and optimized by AWS.

SIDE BAR**Virtual appliances on AWS**

As shown in figure [Figure 3.5](#) virtual appliance is an image of a virtual machine containing an OS and preconfigured software. Virtual appliances are used when the hypervisor starts a new VM. Because a virtual appliance contains a fixed state, every time you start a VM based on a virtual appliance, you'll get exactly the same result. You can reproduce virtual appliances as often as needed, so you can use them to eliminate the cost of installing and configuring complex stacks of software. Virtual appliances are used by virtualization tools from VMware, Microsoft, and Oracle, and for infrastructure-as-a-service (IaaS) offerings in the cloud.

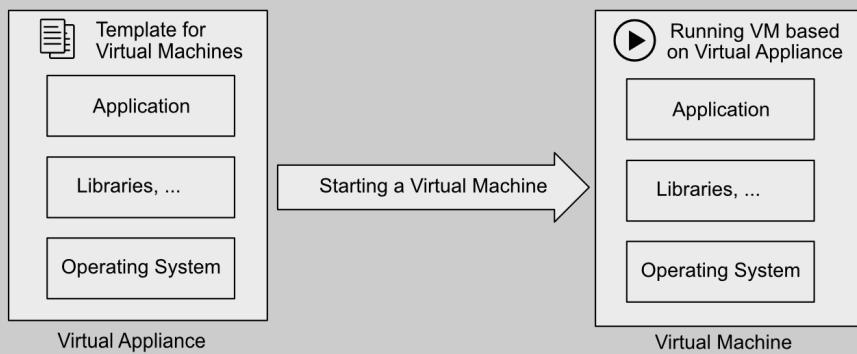


Figure 3.5 A virtual appliance contains a template for a virtual machine

The AMI is a special type of virtual appliance for use with the EC2 service. An AMI technically consists of a read-only filesystem including the OS, additional software, and configuration. So you can also use AMIs for deploying software on AWS. But, the AMI does not include the kernel of the OS. The kernel is loaded from an Amazon Kernel Image (AKI).

Nowadays, AWS uses a virtualization called Nitro. Nitro combines a KVM-based hypervisor with customized hardware (ASICs) aiming to provide a performance that is indistinguishable from bare metal machines.

The current generations of VMs on AWS use hardware-assisted virtualization. The technology is called Hardware Virtual Machine (HVM). A virtual machine run by an AMI based on HVM uses a fully virtualized set of hardware, and can take advantage of extensions that provide fast access to the underlying hardware.

CHOOSING THE SIZE OF YOUR VIRTUAL MACHINE

It's now time to choose the computing power needed for your virtual machine as shown in figure [Figure 3.6](#). AWS classifies computing power into instance types. An instance type primarily describes the number of virtual CPUs and the amount of memory.

Table [3.1](#) shows examples of instance types for different use cases. The prices represent the actual prices in USD for a Linux VM in the US East (N. Virginia) region, as recorded April 5, 2022.

Table 3.1 Examples of instance families and instance types

Instance type	Virtual CPUs	Memory	Description	Typical use case	Monthly cost (USD)
t2.nano	1	0.5 GiB	Small and cheap instance type, with moderate baseline performance and the ability to burst CPU performance above the baseline	Testing and development environments, and applications with very low traffic	\$4
m6i.large	2	8 GiB	Has a balanced ratio of CPU, memory, and networking performance	All kinds of applications, such as medium databases, web servers, and enterprise applications	\$69
r6i.large	2	16 GiB	Optimized for memory-intensive applications with extra memory	In-memory caches and enterprise application servers	\$90

There are instance families optimized for different kinds of use cases.

- *T family*—Cheap, moderate baseline performance with the ability to burst to higher performance for short periods of time
- *M family*—General purpose, with a balanced ration of CPU and memory
- *C family*—Computing optimized, high CPU performance
- *R family*—Memory optimized, with more memory than CPU power compared to M family
- *X family*—Extensive capacity with a focus on memory, up to 1952 GB memory and 128 virtual cores
- *D family*—Storage optimized, offering huge HDD capacity
- *I family*—Storage optimized, offering huge SSD capacity
- *P, G, and CG family*—Accelerated computing based on GPUs (graphics processing units)
- *F family*—Accelerated computing based on FPGAs (field programmable gate arrays)

Additional instance families are available for niche workloads like high-performance computing, in-memory databases, MacOS workloads, and more. See aws.amazon.com/ec2/instance-types/ for a full list of instance types and families.

Our experience indicates that you'll overestimate the resource requirements for your applications. So, we recommend that you try to start your application with a smaller instance

type than you think you need at first. You can change the instance family and type later if needed.

SIDEBAR

Instance types and families

The names for different instance types are all structured in the same way. The instance family groups instance types with similar characteristics. AWS releases new instance types and families from time to time; the different versions are called generations. The instance size defines the capacity of CPU, memory, storage, and networking.

The instance type `t2.micro` tells you the following:

- The instance family is called t. It groups small, cheap virtual machines with low baseline CPU performance but the ability to burst significantly over baseline CPU performance for a short time.
- You're using generation 2 of this instance family.
- The size is micro, indicating that the EC2 instance is very small.

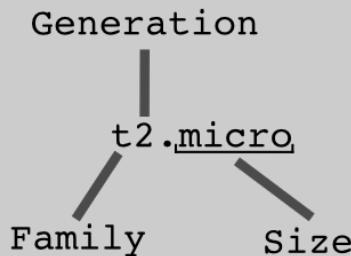


Figure 3.6 Disassembling the `t2.micro` instance type

Computer hardware is getting faster and more specialized, so AWS is constantly introducing new instance types and families. Some of them are improvements of existing instance families, and others are focused on specific workloads. For example, the instance family `R6i` was introduced in November 2021. It provides instances for memory-intensive workloads and replaces the `R5` instance types.

One of the smallest and cheapest VMs will be enough for your first experiments. Choose the instance type `t2.micro` -as shown in figure [Figure 3.7](#)- which is eligible for the Free Tier.

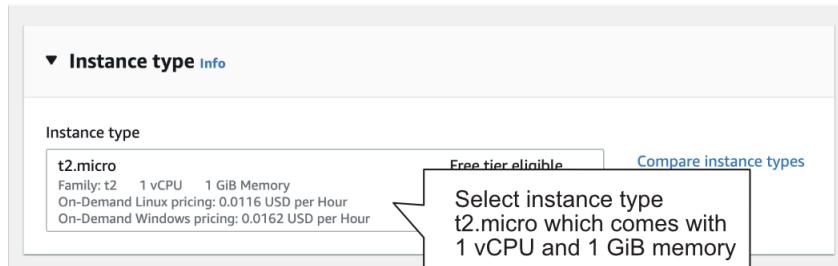


Figure 3.7 Choosing the size of your virtual machine

You might have heard about Apple switching from Intel processors to ARM processors already. The reason for this is that custom-built ARM processors achieve higher performance with lower energy consumption. This is of course not only exciting for laptops, but also for servers in the data center. AWS offers machines based on custom-built ARM processors called *Graviton* as well. As a customer, you will notice similar performance at lower costs. However, you need to make sure that the software you want to run is compiled for the ARM64 architecture. We migrated workloads from EC2 instances with Intel processors to virtual machines with ARM processors a few times already, typically within 1 to 4 hours.

We would have liked to use Graviton instances for the third edition. But unfortunately, these were not yet part of the Free Tier at that time.

We highly recommend to check out the Graviton instance types offered by AWS:

- T4g the burstable and cost-effective instance types.
- M6g, M6gd the general purpose instance types.
- C6g, C6gd, C6gn, C7g the compute optimized instance types.
- R6g, R6gd, X2gd the memory optimized instance types.
- I_m4gn, I_s4gen the instance types providing built-in storage.
- G5g a special instance type optimized for Android game streaming.

CONFIGURING THE KEY PAIR FOR LOGIN

As an administrator of a Linux machine, you used username and password or username and a public/private key pair to authenticate yourself in the past. By default, AWS uses username and a key pair for authentication. That's why the next section of the wizard asks you about defining a key pair for the EC2 instance you are going to launch. We try to avoid this approach, because it only works for a single user, and it is not possible to change the key pair from the outside after launching an EC2 instance.

Therefore, we recommend a different approach to authenticate to an EC2 instance that we will introduce shortly. There is no need to configure the key pair now. So please select *Proceed without a key pair* as shown in figure [Figure 3.8](#).

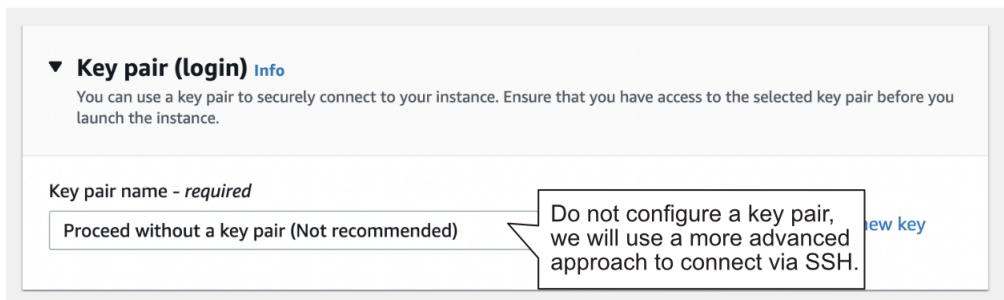


Figure 3.8 Proceed without a key pair

DEFINING NETWORK AND FIREWALL SETTINGS

In the next section it is up to you, to configure the network and firewall settings for the EC2 instance. The defaults are fine for now. You will learn more about networking on AWS in chapter 5. The only thing you should change, is to deselect the *Allow SSH traffic* option. As promised before, you will learn about a new approach to connect to EC2 instances which does not require inbound SSH connectivity. With the configuration shown in figure [Figure 3.9](#) the firewall does not allow any incoming connections at all.

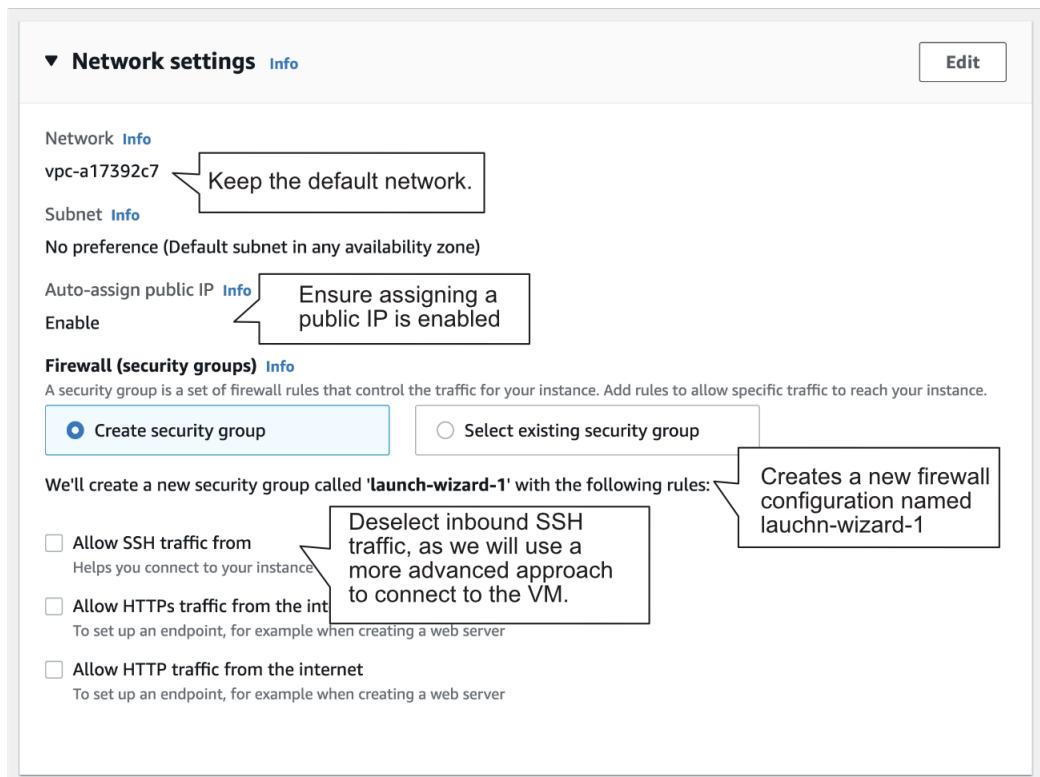


Figure 3.9 Configuring the network and firewall settings for the EC2 instance

ATTACHING STORAGE

Next, attach some storage to your virtual machine used for the root file system. It is fine to keep the defaults and attach a volume with 8 GB of type gp2 which consists of network-attached SSDs as illustrated in figure [Figure 3.10](#).

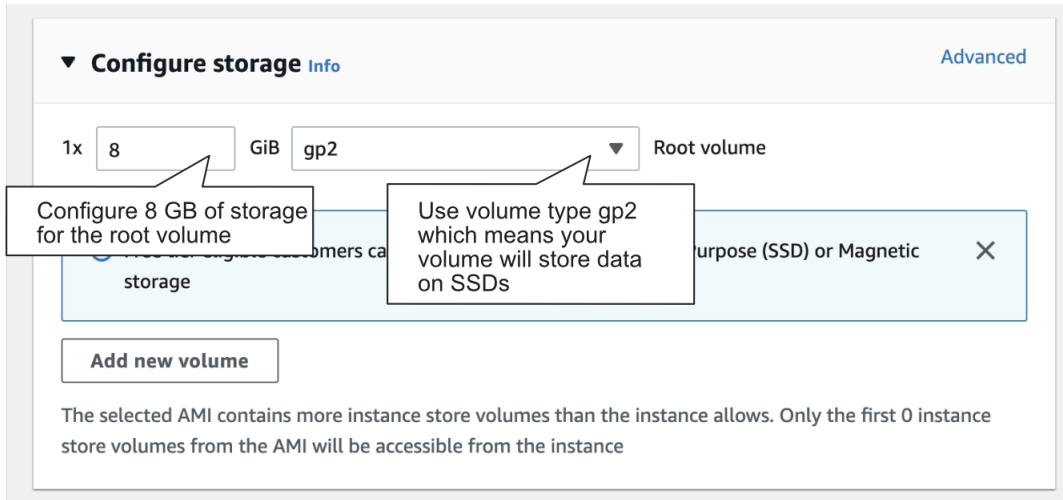


Figure 3.10 Adding storage to the virtual machine

SETTING ADVANCED DETAILS

Last but not least, you need to configure an advanced detail for the EC2 instance you are going to launch: an IAM role. You will learn more about IAM in chapter 5. For now, all you need to know, is that an IAM role grants processes running on the virtual machine access to other AWS services. This is needed, as you will use AWS services called Systems Manager and EC2 Instance Connect to establish an SSH connection with your virtual machine later.

CREATING AN IAM ROLE

Before you proceed configuring your EC2 instance, you need to create an IAM role. To do so open console.aws.amazon.com/iam/ in a new tab in your browser.

1. Select *Roles* from the sub navigation.
2. Press the *Create role* button.
3. Select trusted entity type *AWS service* and *EC2* as shown in figure [Figure 3.11](#).
4. Proceed with the next step.

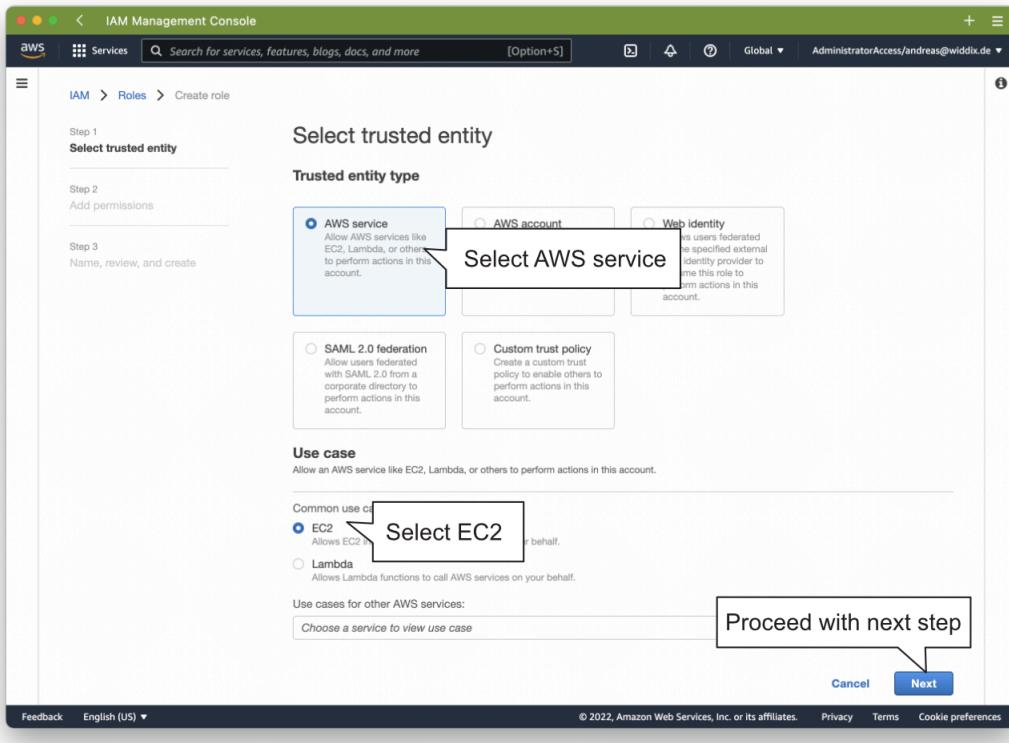


Figure 3.11 Creating an IAM role: Select trusted entity

Add permissions to the IAM role by filtering and selecting the policies named `AmazonSSMManagedInstanceCore` as demonstrated in figure [Figure 3.12](#). Afterwards, proceed with the next step.

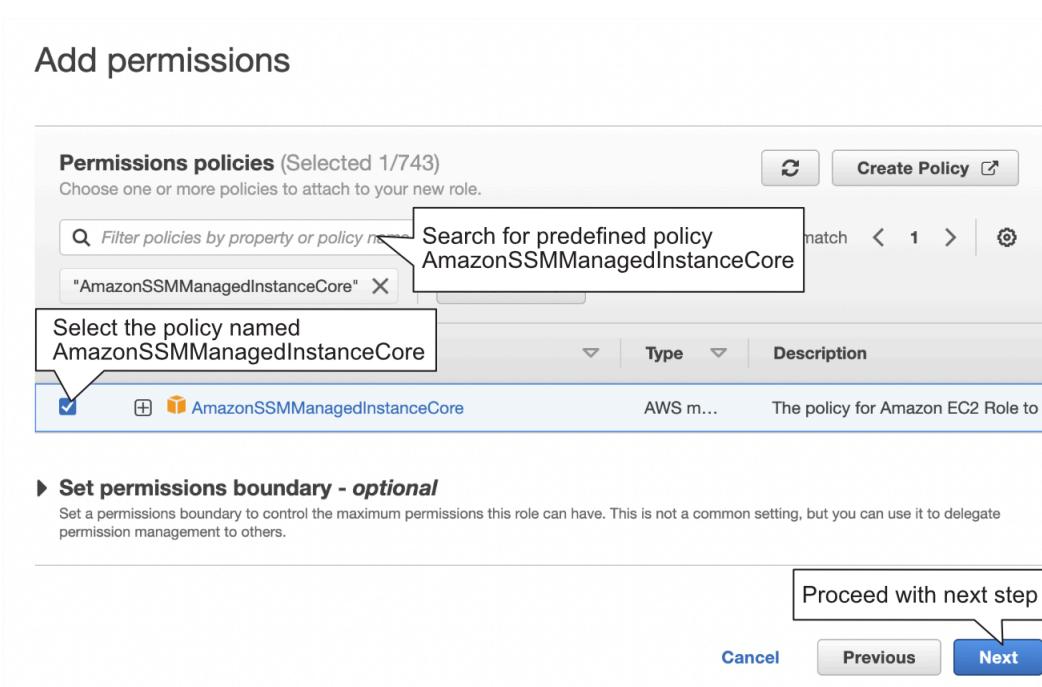


Figure 3.12 Creating an IAM role: Add permissions

To create the IAM role type in the name `ec2-ssm-core` -please use the same name, later chapters depend on it- and a description as shown in figure [Figure 3.13](#). After doing so, press the *Create role* button at the bottom of the page.

Name, review, and create

Role details

Role name
Enter a meaningful name to identify this role
ec2-ssm-core Type in ec2-ssm-core as the name of the IAM role
Maximum 128 characters.

Description
Add a short explanation for this policy.
Allows EC2 instances to interact with SSM.
Optional, add a description to explain the IAM role

Figure 3.13 Creating an IAM role: Role details

Switch back to the *EC2* browser tab. There is a lot to configure in the *advanced details* section. Keep the default for everything expect, the IAM instance profile. Click the reload button next to the dropdown list and select the IAM instance profile named `ec2-ssm-core` afterwards.

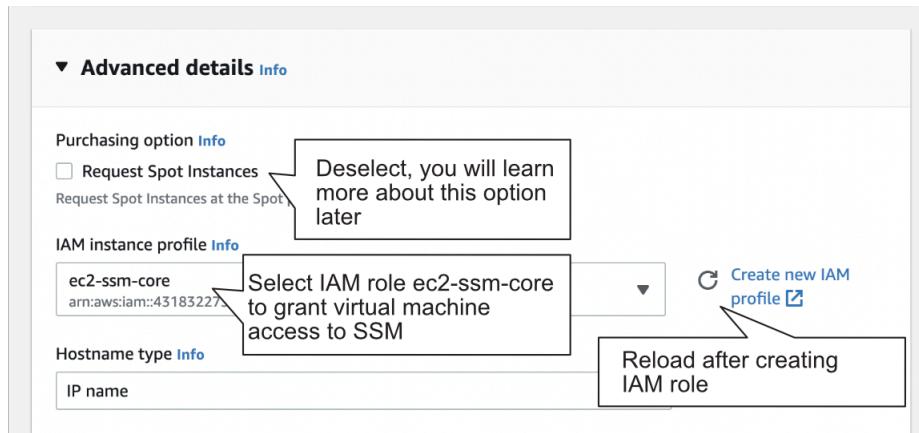


Figure 3.14 Configuring the IAM role for the EC2 instance

LAUNCHING THE EC2 INSTANCE

You are ready to launch your virtual machine. To do so, just press the *Launch instance* button as illustrated in figure [Figure 3.15](#).

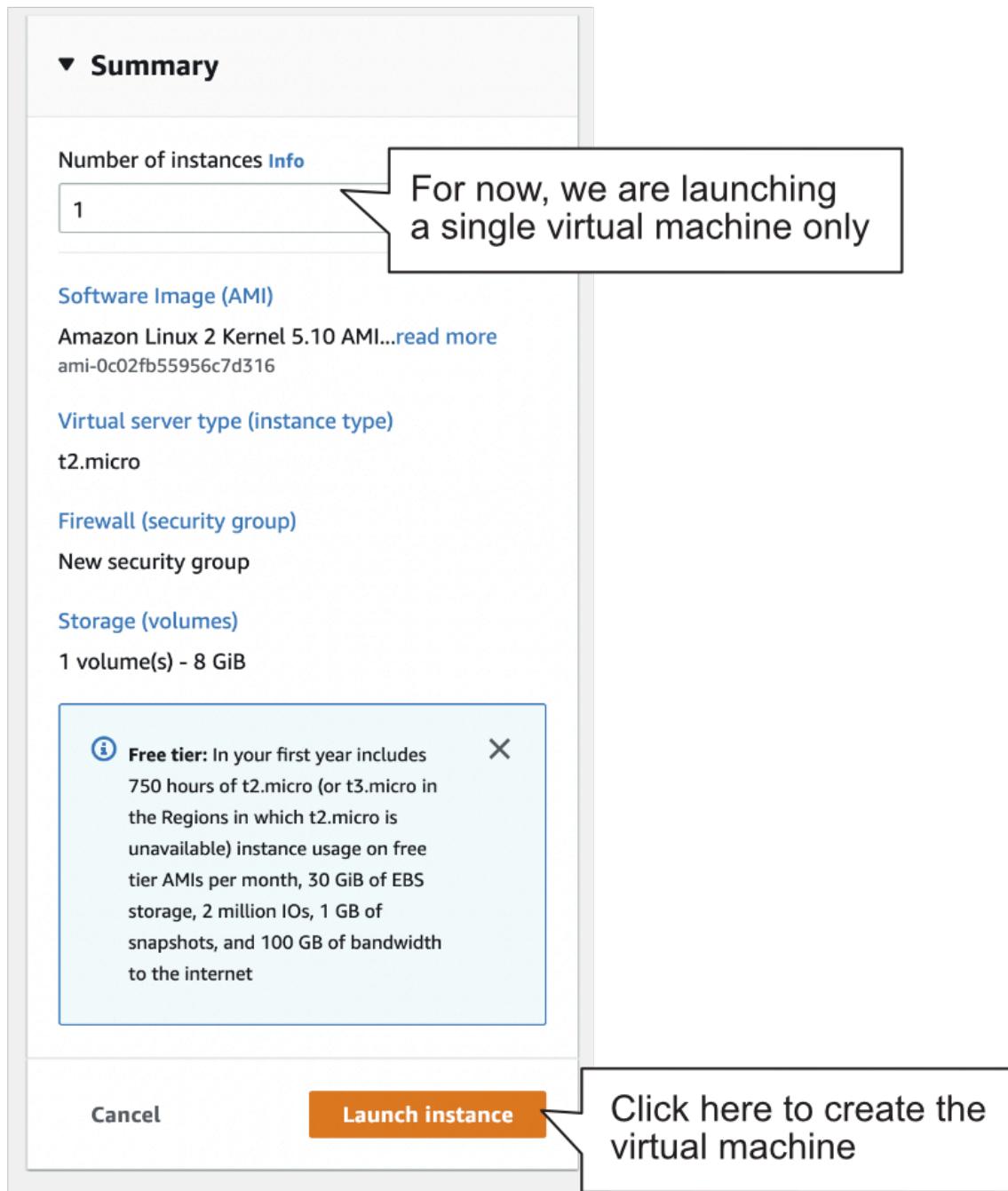


Figure 3.15 Launching the EC2 instance

Mark this day on your calendar. You've just launched your first EC2 instance. Many more will follow!

3.1.2 Connecting to your virtual machine

To be able to do something with the running virtual machines, you have to log in next. After connecting to your E2 instance you will install and run the tool `linkchecker` to check a website for broken links. Of course, this is just an example. As you get administrator access to the virtual machine, you have full control and are able to install application and configure the operating system as needed.

You will learn how to connect to an EC2 instance by using the AWS Systems Manager Session Manager as well as EC2 Instance Connect. The advantages of this approach are:

- You do not need to configure key pairs upfront but use temporary key pairs instead
- No need to allow inbound SSH or RDP connectivity which limits the attack surface.

Using AWS Systems Manager Session Manager comes with the following requirements:

- Works with virtual machines based on Amazon Linux 2, CentOS, Oracle Linux, Red Hat Enterprise Linux, SUSE Linux Enterprise Server, macOS, and Windows Server.
- Requires the `SSM` agent which is pre-installed on Amazon Linux 2, macOS >10.14, SUSE Linux Enterprise Server 12/15, Ubuntu Server >16.04, Windows Server 2008-2012/2016/2019/2022.
- Requires an IAM role granting permissions to the AWS Systems Manager service, see previous section.

The following instructions guide you through the steps necessary to connect to the EC2 instance you launched in the previous section which fulfills all these requirements already:

1. Choose *Instances* from the sub navigation of the EC2 service, in case you are not looking at the list of EC2 instances already as shown in figure [Figure 3.16](#).
2. Select the instance named `mymachine`.
3. Wait until the instance reaches status *Running*.
4. Click the *Connect* button.

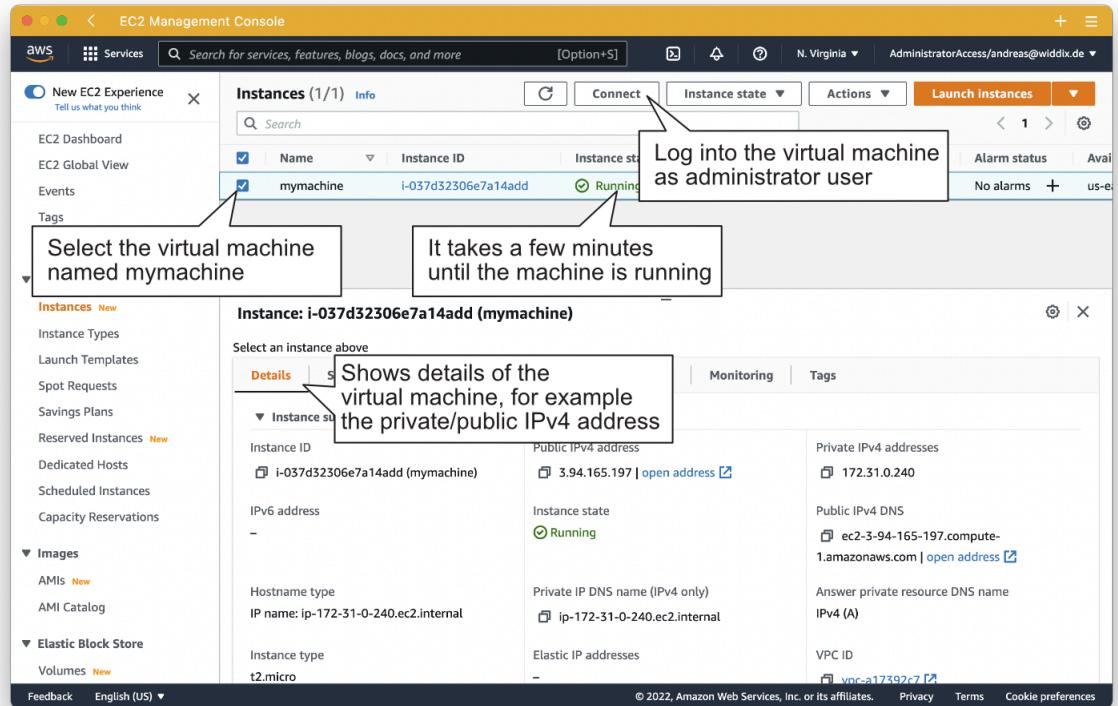


Figure 3.16 Listing all EC2 instances running in the region N. Virginia

5. Select the *Session Manager* tab.
6. Press the *Connect* button as shown in figure [Figure 3.17](#).

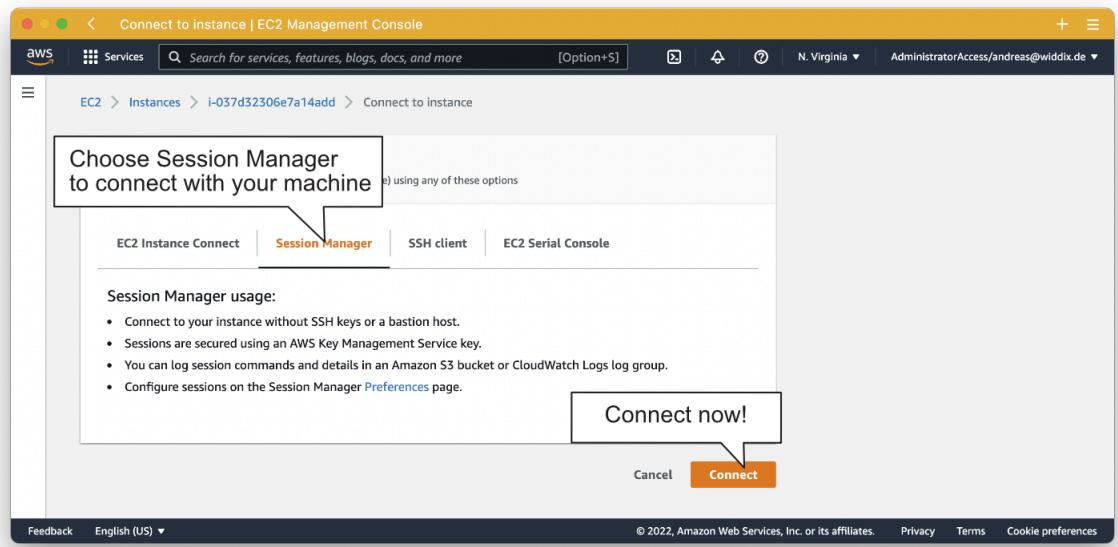


Figure 3.17 Connecting to an EC2 instance via Session Manager

After a few seconds, a terminal appears in your browser window as shown in figure [Figure 3.18](#).



Figure 3.18 The terminal connected with your EC2 instance is waiting for input

SIDE BAR

SSH and SCP

Accessing an EC2 instance directly from the browser is great, but sometimes you might prefer a good old SSH connection from your local machine. For example, to copy files with the help of SCP. Doing so is possible as well. Interested in the details see cloudonaut.io/connect-to-your-ec2-instance-using-ssh-the-modern-way/ and docs.aws.amazon.com/systems-manager/latest/userguide/session-manager-working-with-install-plugin.html.

You are ready to enter the first commands into the terminal of your virtual machine.

3.1.3 Installing and running software manually

Back to our example, you launched a virtual machine to run *linkchecker* to find broken links on a website. First, you need to install *linkchecker*. Also, the tool requires a Python runtime environment.

In general, Amazon Linux 2 comes with the package manager `yum` which allows you to install additional software. Besides that, Amazon Linux 2 comes with an extras library, covering additional software packages.

Run the following command to install Python 3.8. Press `y` when prompted to acknowledge the changes.

```
$ sudo amazon-linux-extras install python3.8
```

Next, execute the following command to install a tool called `linkchecker` that allows you to

find broken links on a website.

```
$ pip3 install linkchecker
```

Now you're ready to check for links pointing to websites that no longer exist. To do so, choose a website and run the following command. The `-r` option limits the recursion level that the tool will crawl through:

```
$ ~/.local/bin/linkchecker -r 2 https://
```

The output of checking the links looks something like this:

```
[...]
URL      `/images/2022/02/terminal.png'
Name     'Connect to your EC2 instance using SSH the modern way'
Parent URL https://clondonaut.io, line 379, col 1165
Real URL https://clondonaut.io/images/2022/02/terminal.png
Check time 2.959 seconds
Size      0B
Result    Error: 404 Not Found
10 threads active,      5 links queued,    72 links in  87 URLs checked, runtime 16 seconds
 1 thread active,      0 links queued,    86 links in  87 URLs checked, runtime 21 seconds

Statistics:
Downloaded: 66.01KB.
Content types: 26 image, 29 text, 0 video, 0 audio, 3 application, 0 mail and 29 other.
URL lengths: min=21, max=160, avg=56.

That's it. 87 links in 87 URLs checked. 0 warnings found. 1 error found.
Stopped checking at 2022-04-04 09:02:55+000 (22 seconds)
[...]
```

Depending on the number of web pages, the crawler may need some time to check all of them for broken links. At the end, it lists the broken links and gives you the chance to find and fix them.

3.2 Monitoring and debugging a virtual machine

If you need to find the reason for an error or why your application isn't behaving as you expect, it's important to have access to tools that can help with monitoring and debugging. AWS provides tools that let you monitor and debug your virtual machines. One approach is to examine the virtual machine's logs.

3.2.1 Showing logs from a virtual machine

If you need to find out what your virtual machine was doing during and after startup, there is a simple solution. AWS allows you to see the EC2 instance's logs with the help of the Management Console (the web interface you use to start and stop virtual machines). Follow these steps to open your VM's logs:

1. Go to EC2 in the AWS Management Console: console.aws.amazon.com/ec2/.
2. Open the list of all virtual machines choosing *Instances* from the sub navigation.

3. Select the running virtual machine by clicking the row in the table.
4. In the Actions menu, choose *Monitor and troubleshoot* and *Get system log*.
5. A screen showing the system logs from your VM that would normally be displayed on a physical monitor during startup as shown in figure [Figure 3.19](#) appears.

The screenshot shows the AWS EC2 Management Console with the URL [Get system log | EC2 Management Console](#). The instance ID is i-07f9230234cb135a0. The log content is as follows:

```

[ 18.567271] cloud-init[3082]: State: Running, pid: 3243
[ 21.485825] cloud-init[3082]: No packages needed for security; 0 packages available
[ 21.492918] cloud-init[3082]: No packages marked for update
[ 22.061580] cloud-init[3277]: Cloud-init v. 19.3-45.amzn2 running 'modules:final' at Mon, 04 Apr 2022 08:52:07 +0000. Up 22.00 se
[ 22.078824] cloud-init[3277]: ci-info: no authorized ssh keys fingerprints found for user ec2-user.
ci-info: no authorized ssh keys fingerprints found for user ec2-user.
<14>Apr 4 08:52:08 ec2:
<14>Apr 4 08:52:08 ec2: #####
<14>Apr 4 08:52:08 ec2: -----BEGIN SSH HOST KEY FINGERPRINTS-----
<14>Apr 4 08:52:08 ec2: 256 SHA256:mB7uvB31XLKvX8i9np0zxczwZmLnXuFkPmA comment (ECDSA)
<14>Apr 4 08:52:08 ec2: 256 SHA256:CY4oRhw8DP+cyyIFxJmkhSnkIPzTqjWn+TQHd no comment (ED25519)
<14>Apr 4 08:52:08 ec2: 2048 SHA256:3yElzkhNP8c0yU8jGHpZtzk4INlVaqR1lvQJA+M9Q no comment (RSA)
<14>Apr 4 08:52:08 ec2: -----END SSH HOST KEY FINGERPRINTS-----
<14>Apr 4 08:52:08 ec2: #####
-----BEGIN SSH HOST KEY KEYS-----
ecdsa-sha2-nistp256 AAAEAE2VzNHNLXNoTT1bm1zdHdYbGNTYAAAIBmlzdHdYbGNTYAAAABBBMn6CeLGP3oF93Gb9hnxmTj5oYmrRxwNUMZ1cz7E/Pd67jfx4Lcv2kP5
ssh-ed25519 AAAAC3NzaC1ZD0iNTESAAAATEsR2+deJUpByCw6imm0SNFKykr0khNg94Zo17Z1v
ssh-rsa AAAAB3NzaC1yc2EAAQABAAAABAcQxFhwSCGM2byB3C9h01V7wIy9mQxb3KBwgchtPfAR5JGFuPapzb3JSFb9v8QWZyWG3J5iZ61QVs8AN0gp0sxZNzr+
-----END SSH HOST KEY KEYS-----
[ 22.198766] cloud-init[3277]: Cloud-init v. 19.3-45.amzn2 finished at Mon, 04 Apr 2022 08:52:07 +0000. Datasource DataSourceEc2.

```

Annotations:

- A callout box points to the top right of the log area with the text "Download logs to archive or analyze in detail."
- A callout box points to the bottom left of the log area with the text "Shows the system logs."
- A callout box points to the bottom left of the log area with the text "For boot or networking issues, use the EC2 serial console for troubleshooting. Choose the Connect button to..."

Figure 3.19 Debugging a virtual machine with the help of system logs

The log contains all log messages that would be displayed on the monitor of your machine if you were running it on-premises. Watch out for any log messages stating that an error occurred during startup. If the error message is not obvious, you should contact the vendor of the AMI, AWS Support, or post your question in the AWS Developer Forums at [forums.aws.amazon.com](#).

This is a simple and efficient way to access your system logs without needing an SSH connection. Note that it will take several minutes for a log message to appear in the log viewer.

3.2.2 Monitoring the load of a virtual machine

AWS can help you answer another question: is your virtual machine close to its maximum capacity? Follow these steps to open the EC2 instance's metrics:

1. Go to EC2 in the AWS Management Console: [console.aws.amazon.com/ec2/](#).
2. Open the list of all virtual machines choosing *Instances* from the sub navigation.
3. Select the running virtual machine by clicking the row in the table.
4. Select the Monitoring tab at lower right.
5. Click the three dots at the upper-right of the *Network in (bytes)* metric and choose *Enlarge*.

You'll see a graph that shows the virtual machine's utilization of incoming networking traffic, similar to figure [Figure 3.20](#). There are metrics for CPU, network, and disk usage. As AWS is looking at your VM from the outside, there is no metric indicating the memory usage. You can publish a memory metric yourself, if needed. The metrics are updated every 5 minutes if you use basic monitoring, or every minute if you enable detailed monitoring of your virtual machine, which costs extra.

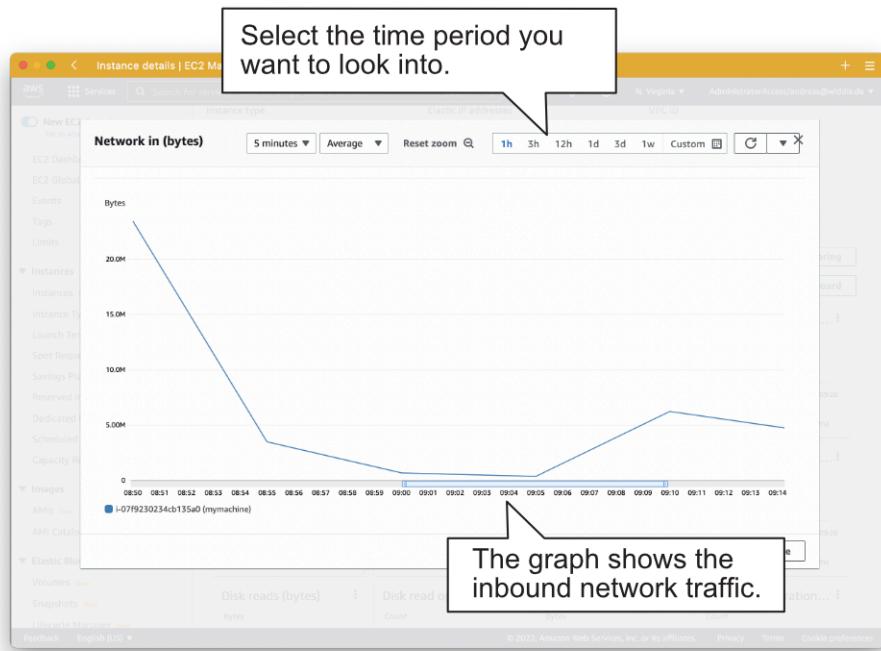


Figure 3.20 Gaining insight into a virtual machine's incoming network traffic with the CloudWatch metric

Checking the metrics of your EC2 instance is helpful when debugging performance issues. You will also learn how to increase or decrease your infrastructure based on these metrics in chapter 17.

Metrics and logs help you monitor and debug your virtual machines. Both tools can help ensure that you're providing high-quality services in a cost-efficient manner. Look at Monitoring Amazon EC2 in the AWS documentation at docs.aws.amazon.com/AWSEC2/latest/UserGuide/monitoring_ec2.html if you are looking for more detailed information about monitoring your virtual machines.

3.3 Shutting down a virtual machine

To avoid incurring charges, you should always turn off virtual machines you're not using them. You can use the following four actions to control a virtual machine's state:

- *Start*—You can always start a stopped virtual machine. If you want to create a

completely new machine, you'll need to launch a virtual machine.

- *Stop*—You can always stop a running virtual machine. A stopped virtual machine doesn't incur charges, except for attached resources like network-attached storage. A stopped virtual machine can be started again but likely on a different host. If you're using network-attached storage, your data persists.
- *Reboot*—Have you tried turning it off and on again? If you need to reboot your virtual machine, this action is what you want. You won't lose any persistent data when rebooting a virtual machine because it stays on the same host.
- *Terminate*—Terminating a virtual machine means deleting it. You can't start a virtual machine that you've already terminated. The virtual machine is deleted, usually together with dependencies like network-attached storage and public and private IP addresses. A terminated virtual machine doesn't incur charges.

WARNING The difference between stopping and terminating a virtual machine is important. You can start a stopped virtual machine. This isn't possible with a terminated virtual machine. If you terminate a virtual machine, you delete it.

Figure [Figure 3.21](#) illustrates the difference between stopping and terminating an EC2 instance, with the help of a flowchart.

It is always possible to stop a running machine and to start a stopped machine.



But terminating is deleting your virtual machine.



Figure 3.21 Difference between stopping and terminating a virtual machine

Stopping or terminating unused virtual machines saves costs and prevents you from being surprised by an unexpected bill from AWS. You may want to stop or terminate unused virtual machines when:

- You have launched virtual machines to implement a proof-of-concept. After finishing the project, the virtual machines are no longer needed. Therefore, you can terminate them.
- You are using a virtual machine to test a web application. As no one else uses the virtual machine, you can stop it before you knock off work, and start it back up again the following day.
- One of your customers canceled their contract. After backing up relevant data, you can terminate the virtual machines that had been used for your former customer.

After you terminate a virtual machine, it's no longer available and eventually disappears from the list of virtual machines.

SIDE BAR**Cleaning up**

cleanup

Terminate the virtual machine named `mymachine` that you started at the beginning of this chapter:

1. Go to EC2 in the AWS Management Console: console.aws.amazon.com/ec2/.
2. Open the list of all virtual machines choosing Instances from the sub navigation.
3. Select the running virtual machine by clicking the row in the table.
4. Click the Instance state button and select Terminate instance.

3.4 Changing the size of a virtual machine

It is always possible to change the size of a virtual machine. This is one of the advantages of using the cloud, and it gives you the ability to scale vertically. If you need more computing power, increase the size of the EC2 instance or wise versa.

In this section, you'll learn how to change the size of a running virtual machine. To begin, follow these steps to start a small virtual machine:

1. Go to EC2 in the AWS Management Console: console.aws.amazon.com/ec2/.
2. Click the *Launch instances* button.
3. Type in `growingup` as the name of the virtual machine.
4. Choose the Amazon Linux 2 AMI.
5. Select instance type `t2.micro`.
6. Select the option *Proceed without a key pair*.
7. Keep the default for network and storage.
8. Select the IAM instance profile `ec2-ssm-core` under advanced details.
9. Launch the instance.

You've now started an EC2 instance of type `t2.micro`. This is one of the smallest virtual machines available on AWS.

Use the Session Manager to connect to the instance as demonstrated in the previous section, and execute `cat /proc/cpuinfo` and `free -m` to see information about the machine's capabilities. The output should look similar to this:

```
$ cat /proc/cpuinfo
processor      : 0
vendor_id     : GenuineIntel
cpu family    : 6
model         : 63
model name    : Intel(R) Xeon(R) CPU E5-2676 v3 @ 2.40GHz
stepping       : 2
microcode     : 0x46
cpu MHz       : 2399.915
cache size    : 30720 KB
[...]
```

```
$ free -m
      total    used   free   shared   buff/cache   available
Mem:      965      93    379        0        492      739
Swap:       0       0       0
```

Your virtual machine provides a single CPU core and 965 MB of memory. If your application is having performance issues, increasing the instance size can solve the problem. Use your machine's metrics as described in section 3.3 to find out if you are running out of CPU or networking capacity. Would your application benefit from additional memory? If so, increasing the instance size will improve the application's performance as well.

If you need more CPUs, more memory, or more networking capacity, there are many other sizes to choose from. You can even change the virtual machine's instance family and generation. To increase the size of your VM, you first need to stop it:

1. Go to EC2 in the AWS Management Console: console.aws.amazon.com/ec2/.
2. Click *Instances* in the sub menu to jump to an overview of your virtual machines.
3. Select your running VM from the list by clicking it.
4. Click *Instance state* and *Stop instance*.

WARNING Starting a virtual machine with instance type m5.large incurs charges. Go to aws.amazon.com/ec2/pricing if you want to see the current on-demand hourly price for an m4.large virtual machine.

After waiting for the virtual machine to stop, you can change the instance type:

1. Click the *Actions* button and select *Instance settings*.
2. Click *Change instance type*.
3. Select m5.large as the new instance type and press the *Apply* button as shown in figure [Figure 3.22](#).

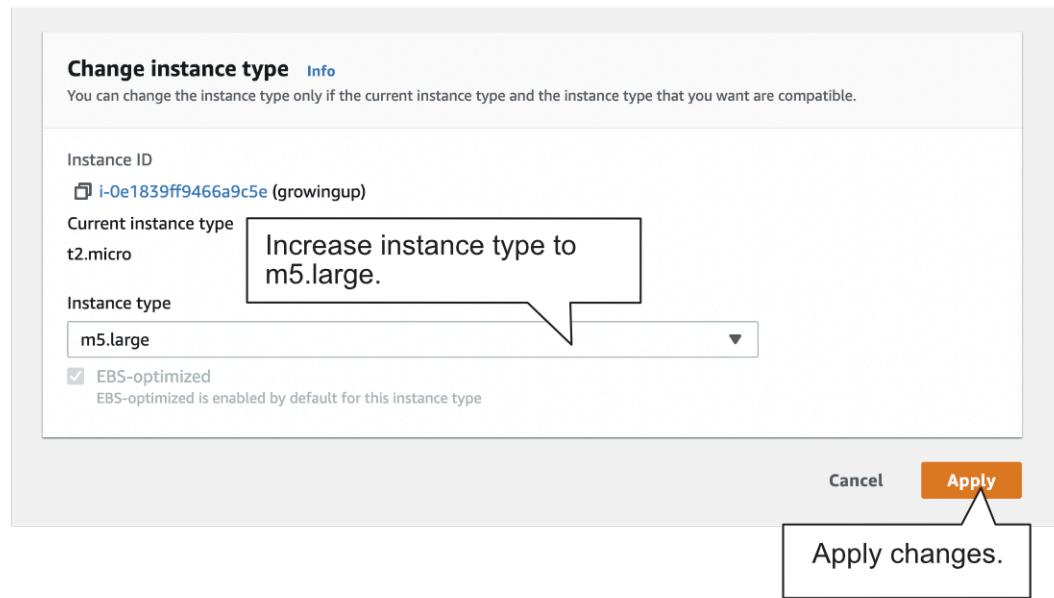


Figure 3.22 Increasing the size of your virtual machine by selecting m4.large as Instance Type

You've now changed the size of your virtual machine and are ready to start it again. To do so, select your EC2 instance and choose *Start instance* under *Instance state*. Your VM will start with more CPUs, more memory, and increased networking capabilities.

Use the Session Manager to connect to your EC2 instance, and execute `cat /proc/cpuinfo` and `free -m` to see information about its CPU and memory again. The output should look similar to this:

```
$ cat /proc/cpuinfo
processor      : 0
vendor_id     : GenuineIntel
cpu family    : 6
model         : 85
model name    : Intel(R) Xeon(R) Platinum 8259CL CPU @ 2.50GHz
stepping       : 7
microcode     : 0x500320a
cpu MHz       : 3117.531
cache size    : 36608 KB
[...]

processor      : 1
vendor_id     : GenuineIntel
cpu family    : 6
model         : 85
model name    : Intel(R) Xeon(R) Platinum 8259CL CPU @ 2.50GHz
stepping       : 7
microcode     : 0x500320a
cpu MHz       : 3100.884
cache size    : 36608 KB
[...]
```

```
$ free -m
      total    used    free   shared   buff/cache   available
Mem:     7737      108    7427        0        202      7406
Swap:      0       0       0
```

Your virtual machine can use two CPU cores and offers 7,737 MB of memory, compared to a single CPU core and 965 MB of memory before you increased the VM's size.

SIDE BAR Cleaning up



cleanup

Terminate the EC2 instance named `growingup` of type `m5.large` to stop paying for it:

1. Go to EC2 in the AWS Management Console: console.aws.amazon.com/ec2/.
2. Open the list of all virtual machines choosing Instances from the sub navigation.
3. Select the running virtual machine by clicking the row in the table.
4. In the Actions menu, choose Instance state and Terminate instance.

3.5 1.5 Starting a virtual machine in another data center

AWS offers data centers all over the world. Take the following criteria into account when deciding which region to choose for your cloud infrastructure:

- *Latency*—Which region offers the shortest distance between your users and your infrastructure?
- *Compliance* --Are you allowed to store and process data in that country?
- *Service availability*—AWS does not offer all services in all regions. Are the services you are planning to use available in the region? Check out the service availability region table at docs.aws.amazon.com/AWSEC2/latest/UserGuide/monitoring_ec2.html or awservices.info.
- *Costs*—Service costs vary by region. Which region is the most cost-effective region for your infrastructure?

Let's assume you have customers not just in the United States but in Australia as well. At the moment you are only operating EC2 instances in N. Virginia (US). Customers from Australia complain about long loading times when accessing your website. To make your Australian customers happy, you decide to launch an additional VM in Australia.

Changing a data center is simple. The Management Console always shows the current data center you're working in, on the right side of the main navigation menu. So far, you've worked in the

data centers located in *N. Virginia (US)*, called `us-east-1`. To change the data center, click *N. Virginia* and select *Asia Pacific (Sydney)* from the menu. Figure [Figure 3.23](#) shows how to jump to the data center in Sydney also called `ap-southeast-2`.

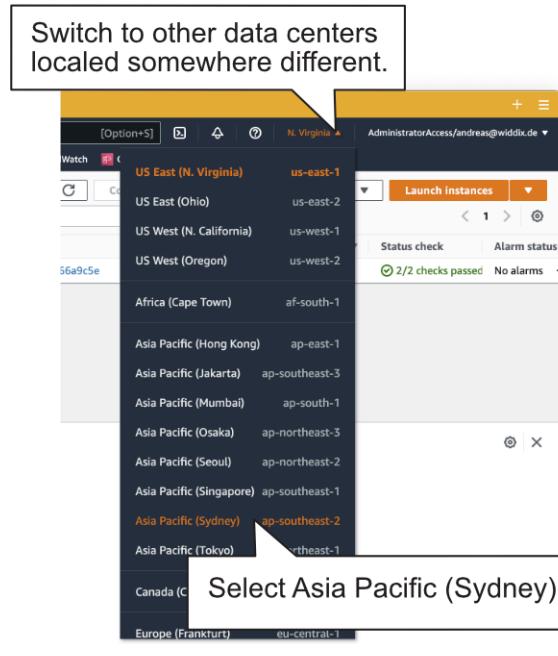


Figure 3.23 Changing the data center's location from N. Virginia to Sydney in the Management Console

AWS groups its data centers into these regions:

<ul style="list-style-type: none"> • US East, N. Virginia (us-east-1) 	<ul style="list-style-type: none"> • US East, Ohio (us-east-2)
<ul style="list-style-type: none"> • US West, N. California, us-west-1 	<ul style="list-style-type: none"> • US West, Oregon (us-west-2)
<ul style="list-style-type: none"> • Africa, Cape Town (af-south-1) 	<ul style="list-style-type: none"> • Asia Pacific, Hong Kong (ap-east-1)
<ul style="list-style-type: none"> • Asia Pacific, Jakarta (ap-southeast-3) 	<ul style="list-style-type: none"> • Asia Pacific, Mumbai (ap-south-1)
<ul style="list-style-type: none"> • Asia Pacific, Osaka (ap-northeast-3) 	<ul style="list-style-type: none"> • Asia Pacific, Seoul (ap-northeast-2)
<ul style="list-style-type: none"> • Asia Pacific, Singapore (ap-southeast-1) 	<ul style="list-style-type: none"> • Asia Pacific, Sydney (ap-southeast-2)
<ul style="list-style-type: none"> • Asia Pacific, Tokyo (ap-northeast-1) 	<ul style="list-style-type: none"> • Canada, Central (ca-central-1)
<ul style="list-style-type: none"> • Europe, Frankfurt (eu-central-1) 	<ul style="list-style-type: none"> • Europe, Ireland (eu-west-1)
<ul style="list-style-type: none"> • Europe, London (eu-west-2) 	<ul style="list-style-type: none"> • Europe, Milan (eu-south-1)
<ul style="list-style-type: none"> • Europe, Paris (eu-west-3) 	<ul style="list-style-type: none"> • Europe, Stockholm (eu-north-1)
<ul style="list-style-type: none"> • Middle East, Bahrain (me-south-1) 	<ul style="list-style-type: none"> • South America, São Paulo (sa-east-1)

You can specify the region for most AWS services. The regions are independent of each other; data isn't transferred between regions. Typically, a region is a collection of three or more data centers located in the same area. Those data centers are well connected to each other and offer the ability to build a highly available infrastructure, as you'll discover later in this book. Some AWS services, like IAM where you created the `ec2-ssm-core` role, or the CDN and the Domain Name System (DNS) service, act globally on top of these regions and even on top of some additional data centers .

Next, start a virtual machine in a data center in Sydney. Follow these steps to do so:

1. Go to EC2 in the AWS Management Console: console.aws.amazon.com/ec2/.
2. Click the *Launch instances* button.
3. Type in `sydney` as the name of the virtual machine.
4. Choose the Amazon Linux 2 AMI.
5. Select instance type `t2.micro`.
6. Select the option *Proceed without a key pair*.
7. Tick the *Allow HTTP traffic from the internet* option.
8. Keep the defaults for storage.
9. Select the IAM instance profile `ec2-ssm-core` under advanced details.
10. Launch the instance.

You did it! A virtual machine is running in a data center in Sydney. Let's proceed with installing a web server on it. To do so, you have to connect to your virtual machine via Session Manager as you did in the previous sections.

Use the following commands to install and start an Apache web server on your virtual machine.

```
sudo yum install httpd -y
```

Next, start the web server and make sure it will get started whenever the machine starts automatically.

```
sudo systemctl start httpd
sudo systemctl enable httpd
```

To access the default web site served by Apache, you need to know the public IPv4 address of your EC2 instance. Get this information by selecting your virtual machine and look into the details via Management Console. Or use execute the following command in the terminal of your EC2 instance.

```
curl http://169.254.169.254/latest/meta-data/public-ipv4
```

Open `$PublicIp` in your browser. Don't forget to replace `$PublicIp` with the public IPv4 address of your EC2 instance. For example, 52.54.202.9. A demo website appears.

The public IPv4 address assigned to your EC2 instance is suspect to change. For example, when you stop and start your instance, AWS assigns a new public IPv4 address. Therefore, you will learn how to attach a fixed public IP address to the virtual machine in the following section.

3.6 Allocating a public IP address

You've already launched some virtual machines while reading this book. Each VM was connected to a public IP address automatically. But every time you launched or stopped a VM, the public IP address changed. If you want to host an application under a fixed IP address, this won't work. AWS offers a service called *Elastic IPs* for allocating fixed public IP addresses.

Using a fixed public IP address is useful, in case clients aren't able to resolve a DNS name, a firewall rule based on IP addresses is required, or in case you do not want to update DNS records to avoid the delay until all clients resolve to the new IP address.

Therefore, allocate a public IP address and associate with your EC2 instance named `syndey`:

1. Open the Management Console, and go to the EC2 service.
2. Choose *Elastic IPs* from the submenu. You'll see an overview of public IP addresses allocated by you.
3. Allocate a public IP address by clicking *Allocate Elastic IP address* as shown in figure [Figure 3.24](#).

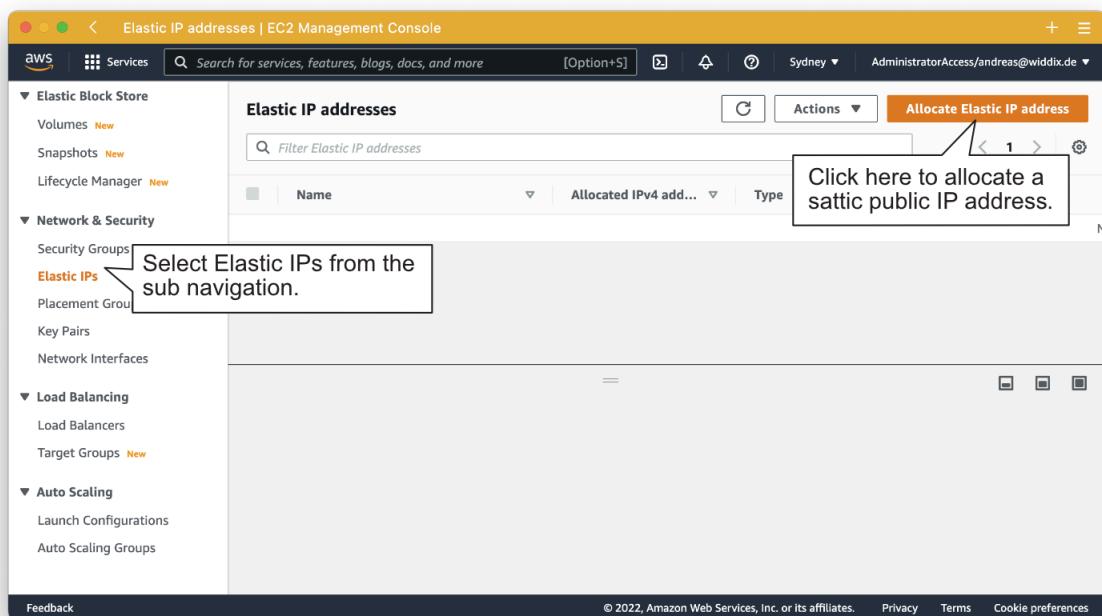


Figure 3.24 Overview of public IP addresses connected to your account in the current region

4. Select *Amazon's pool of IPv4 addresses* and click the *Allocate* button as shown in figure [Figure 3.25](#).

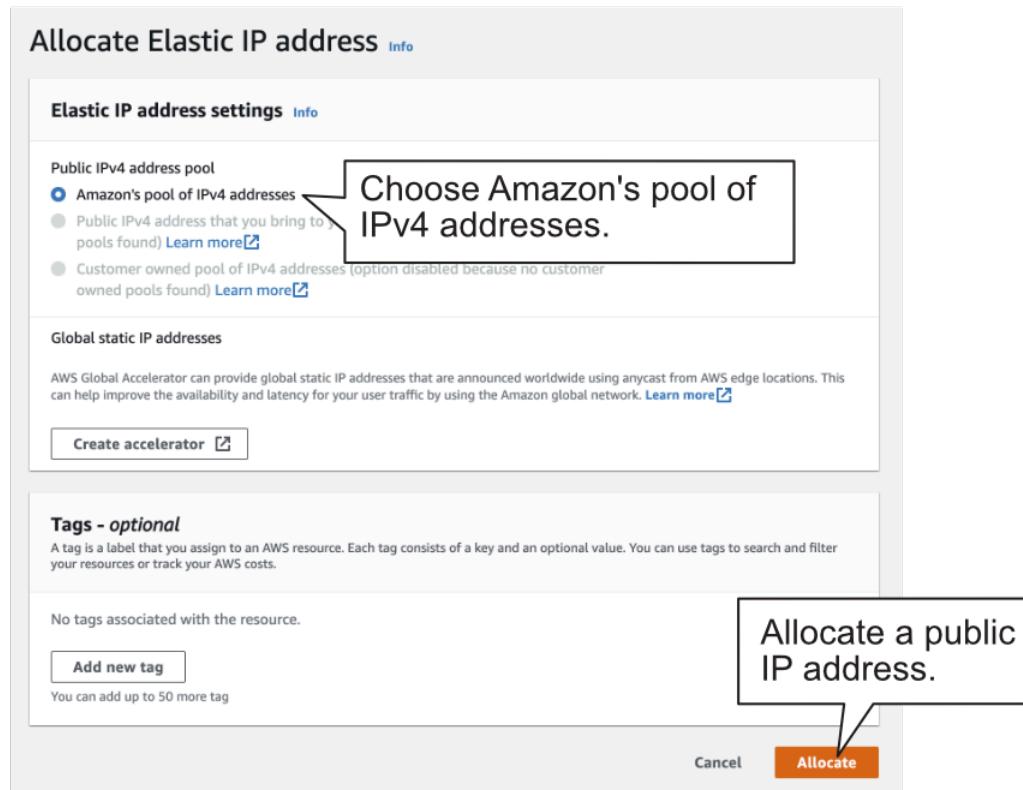


Figure 3.25 Allocating a public IPv4 address

5. To associate the Elastic IP with your EC2 instance, select the public IP address you just allocated and click the *Actions* button and choose *Associate Elastic IP address* as demonstrated in figure [Figure 3.26](#).
6. Select resource type *Instance* and select your EC2 instance named *sydney* from the dropdown list.
7. Press the *Associate* button.

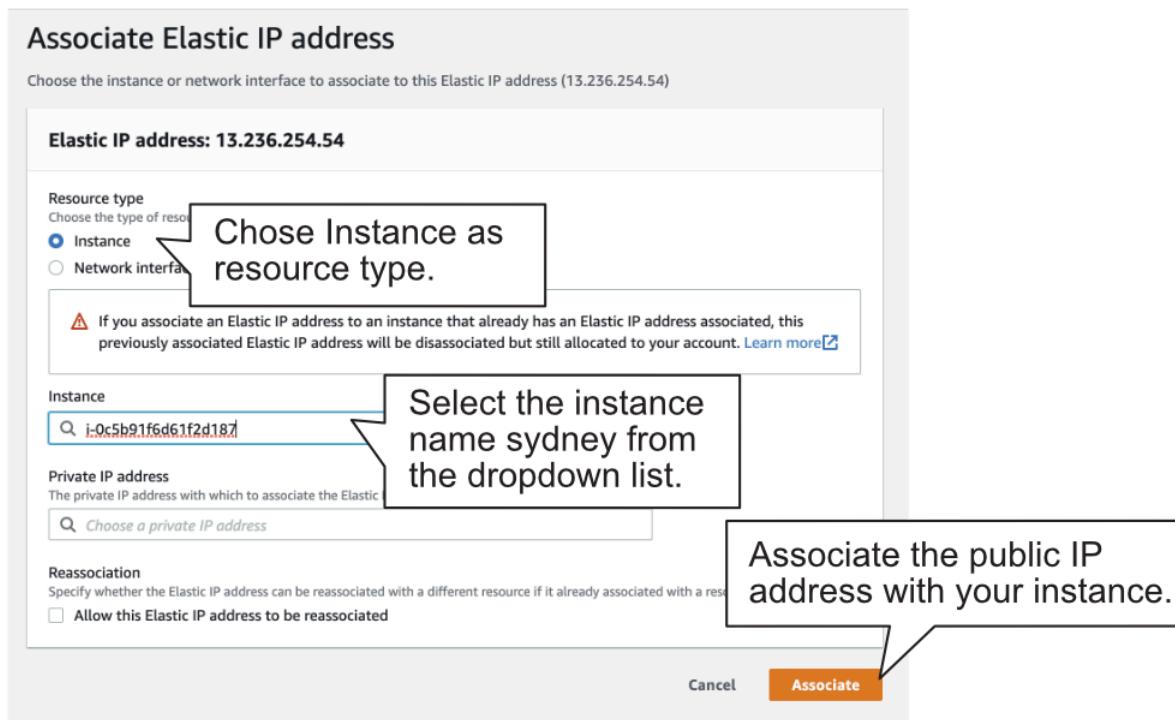


Figure 3.26 Associating an Elastic IP address with an EC2 instance

Hurray, your virtual machine is now accessible through the public IP address you allocated at the beginning of this section. Point your browser to this IP address, and you should see the placeholder page as you did in section 3.5.

Allocating a static public IP address can be useful if you want to make sure the endpoint to your application doesn't change, even if you have to replace the virtual machine behind the scenes. For example, assume that virtual machine A is running and has an associated Elastic IP. The following steps let you replace the virtual machine with a new one without changing the public IP address:

1. Start a new virtual machine B to replace running virtual machine A.
2. Install and start applications as well as all dependencies on virtual machine B.
3. Disassociate the Elastic IP from virtual machine A, and associate it with virtual machine B.

Requests using the Elastic IP address will now be routed to virtual machine B, with a short interruption while moving the Elastic IP. You can also connect multiple public IP addresses with a virtual machine by using multiple network interfaces, as described in the next section. This can be useful if you need to host different applications running on the same port, or if you want to use a unique fixed public IP address for different websites.

WARNING IPv4 addresses are scarce. To prevent stockpiling Elastic IP addresses, AWS will charge you for Elastic IP addresses that aren't associated with a virtual machine. You'll clean up the allocated IP address at the end of the next section.

3.7 Adding an additional network interface to a virtual machine

In addition to managing public IP addresses, you can control your virtual machine's network interfaces. It is possible to add multiple network interfaces to a VM and control the private and public IP addresses associated with those network interfaces.

Here are some typical use cases for EC2 instances with multiple network interfaces:

- Your web server needs to answer requests by using multiple TLS/SSL certificates, and you can't use the Server Name Indication (SNI) extension due to legacy clients.
- You want to create a management network separated from the application network, and therefore your EC2 instance needs to be accessible from two networks. Figure [Figure 3.27](#) illustrates an example.
- Your application requires or recommends the use of multiple network interfaces (for example, network and security appliances).

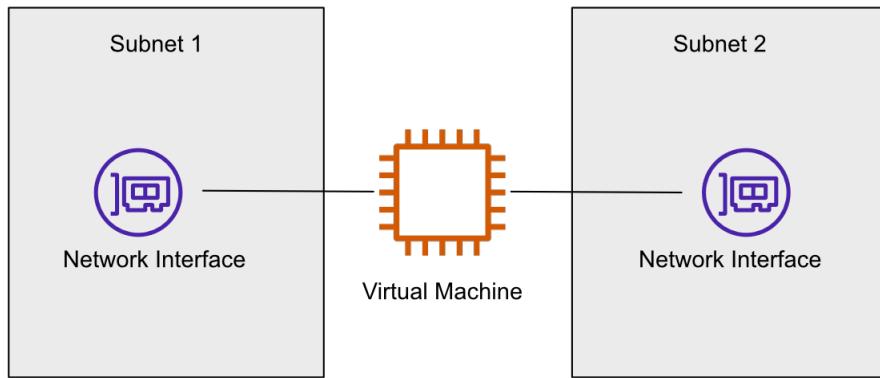


Figure 3.27 A virtual machine with two network interfaces in two different subnets

In the following, you will use an additional network interface to connect a second public IP address to your EC2 instance. Follow these steps to create an additional networking interface for your virtual machine:

1. Open the Management Console, and go to the EC2 service.
2. Select *Network Interfaces* from the submenu.
3. The default network interface of your virtual machine is shown in the list. Note the subnet ID of this network interface.
4. Click *Create network interface*.

5. Enter 2nd interface as the description as shown in figure [Figure 3.28](#).
6. Choose the subnet you noted down in step 3.
7. Select *Auto-assign* private IPv4 address.
8. Choose the security group named `launch-wizard-1`.
9. Click *Create network interface*.

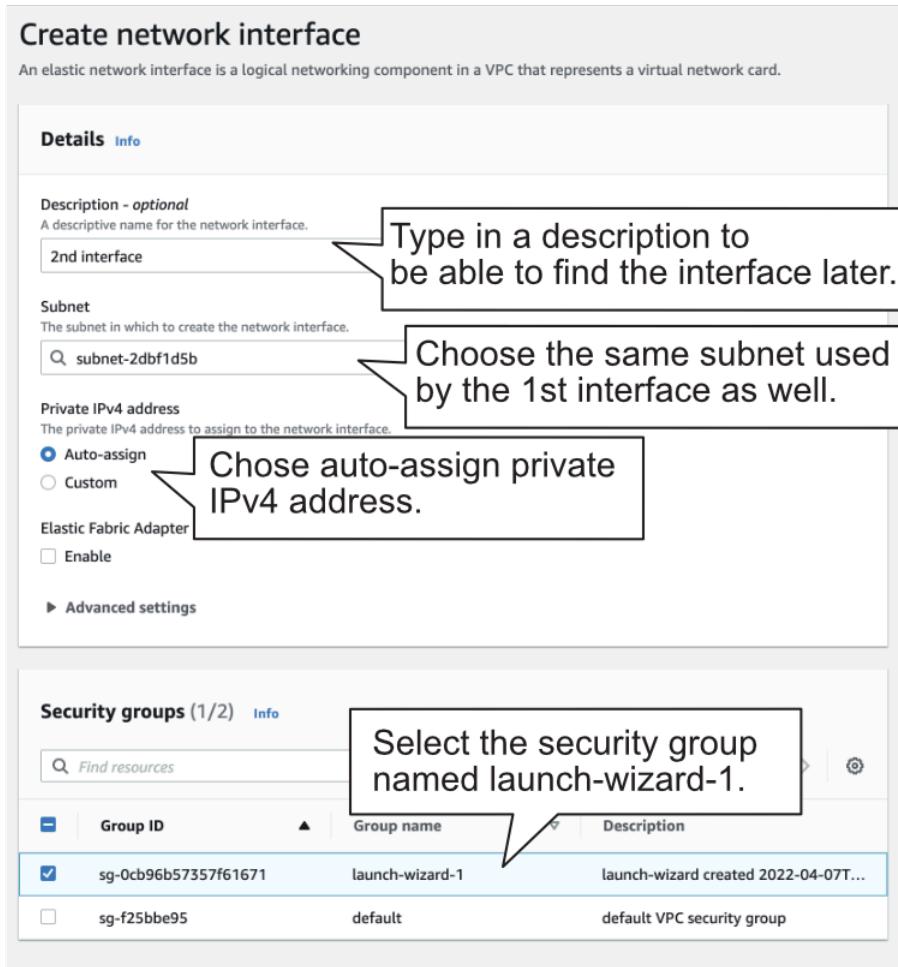


Figure 3.28 Creating an additional networking interface for your virtual machine

After the new network interface's state changes to *Available*, you can attach it to your virtual machine. Select the new 2nd interface network interface, and choose *Attach* from the *Actions* menu. A dialog opens like shown in figure [Figure 3.29](#). Choose the only available Instance ID, and click *Attach*.

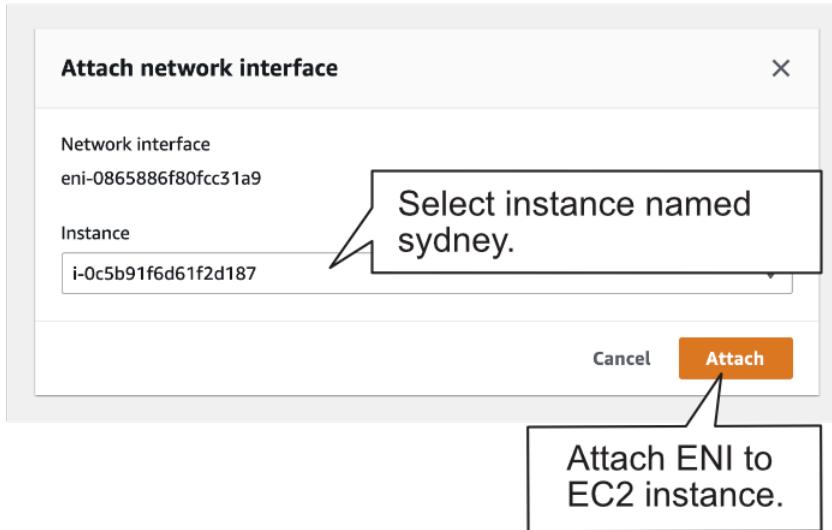


Figure 3.29 Attaching an additional networking interface to your virtual machine

You've attached an additional networking interface to your virtual machine. Next, you'll associate an additional public IP address to the additional networking interface. To do so, note down the network interface ID of the 2nd interface shown in the overview—`eni-0865886f80fcc31a9` in our example—and follow these steps:

1. Open the AWS Management Console, and go to the EC2 service.
2. Choose *Elastic IPs* from the submenu.
3. Click *Allocate Elastic IP address* as you did in section 3.6.
4. Select the newly created public IP address, and choose *Associate Elastic IP address* from the *Actions* menu.
5. Select *Network interface* as the resource type as shown in figure [Figure 3.30](#).
6. Select your 2nd interface's ID.
7. Select the only available private IP of your network interface.
8. Click *Associate* to finish the process.

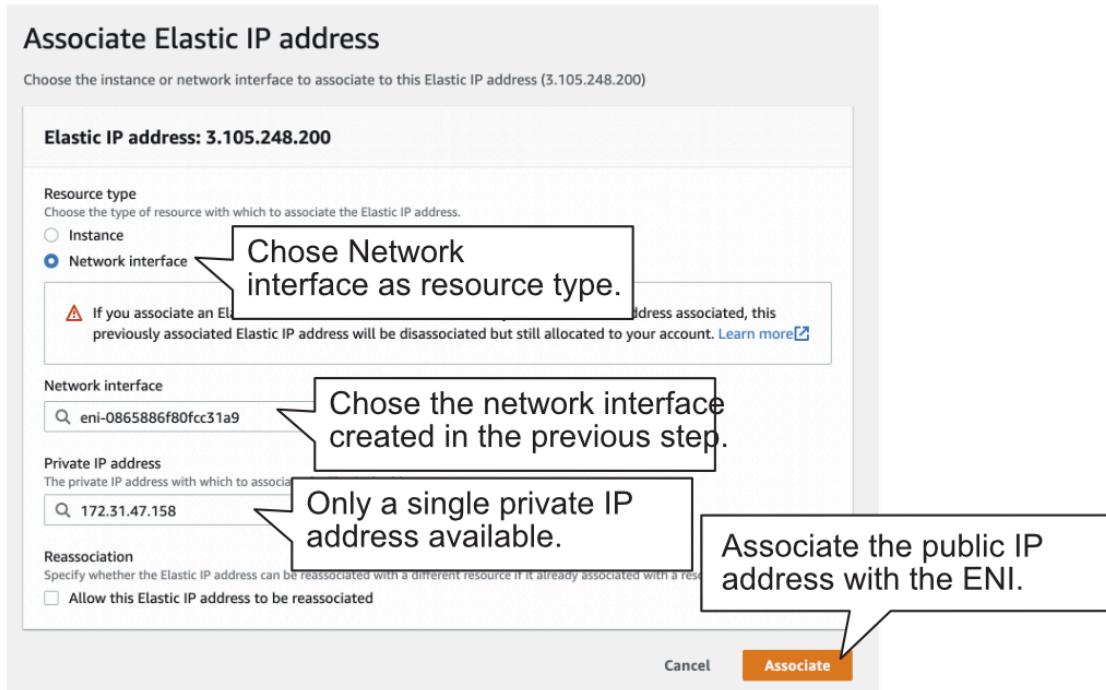


Figure 3.30 Associating a public IP address with the additional networking interface

Your virtual machine is now reachable under two different public IP addresses. This enables you to serve two different websites, depending on the public IP address. You need to configure the web server to answer requests depending on the public IP address.

Use the Session Manager to connect to your EC2 instance named `sydney` and execute `ifconfig` in the terminal which will output the network configuration of your virtual machine.

```
$ ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 9001
      inet 172.31.33.219 netmask 255.255.240.0 broadcast 172.31.47.255 ①
          inet6 fe80::495:5fff:fea6:abde prefixlen 64 scopeid 0x20<link>
              ether 06:95:5f:a6:ab:de txqueuelen 1000 (Ethernet)
              RX packets 68382 bytes 80442006 (76.7 MiB)
              RX errors 0 dropped 0 overruns 0 frame 0
              TX packets 35228 bytes 4219870 (4.0 MiB)
              TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

eth1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 9001
      inet 172.31.47.158 netmask 255.255.240.0 broadcast 172.31.47.255 ②
          inet6 fe80::4a2:8fff:feea:bbba prefixlen 64 scopeid 0x20<link>
              ether 06:a2:8f:ea:bb:ba txqueuelen 1000 (Ethernet)
              RX packets 22 bytes 1641 (1.6 KiB)
              RX errors 0 dropped 0 overruns 0 frame 0
              TX packets 33 bytes 2971 (2.9 KiB)
              TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
[...]
```

- ① The primary network interface uses private IP address 172.31.33.219.
- ② The secondary network interface uses private IP address 172.31.47.158.

Each network interface is connected to a private and a public IP address. You'll need to

configure the web server to deliver different websites depending on the IP address. Your virtual machine doesn't know anything about its public IP address, but you can distinguish the requests based on the private IP addresses.

First you need two websites. Run the following commands on your virtual machine in Sydney via Session Manager to download two simple placeholder websites:

```
$ sudo -s
$ mkdir /var/www/html/a
$ wget -P /var/www/html/a \
[CA] https://raw.githubusercontent.com/AWSinAction/code3/main/chapter03/a/index.html
$ mkdir /var/www/html/b
$ wget -P /var/www/html/b \
[CA] https://raw.githubusercontent.com/AWSinAction/code3/main/chapter03/b/index.html
```

Next you need to configure the web server to deliver the websites depending on which IP address is called. To do so, add a file named `a.conf` under `/etc/httpd/conf.d`. For example, by using the editor `nano`.

```
nano /etc/httpd/conf.d/a.conf
```

Copy and paste the following file content. Change the IP address from `172.31.x.x` to the IP address from the `ifconfig` output for the networking interface `eth0`:

```
<VirtualHost 172.31.x.x:80>
    DocumentRoot /var/www/html/a
</VirtualHost>
```

Press `CTRL + X` and select `y` to save the file.

Repeat the same process for a configuration file named `b.conf` under `/etc/httpd/conf.d` with the following content. Change the IP address from `172.31.y.y` to the IP address from the `ifconfig` output for the networking interface `eth1`:

```
<VirtualHost 172.31.y.y:80>
    DocumentRoot /var/www/html/b
</VirtualHost>
```

To activate the new web server configuration, execute `systemctl restart httpd`.

Next, go to the *Elastic IP* overview in the Management Console. Copy both public IP addresses, and open them with your web browser. You should get the answer "Hello A!" or "Hello B!" depending on the public IP address you're calling. Thus you can deliver two different websites, depending on which public IP address the user is calling. Congrats—you're finished!

SIDE BAR**Cleaning up**

cleanup

It's time to clean up your setup:

1. Terminate the virtual machine and wait until it is terminated.
2. Go to Networking Interfaces and, select and delete the 2nd networking interface.
3. Change to Elastic IPs, and select and release the two public IP addresses by clicking Release Elastic IP address from the Actions menu.
4. Go to Security Groups, and delete the `launch-wizard-1` security group you created.

That's it. Everything is cleaned up, and you're ready for the next section.

NOTE

You switched to the AWS region in Sydney earlier. Now you need to switch back to the region US East (N. Virginia). You can do so by selecting US East (N. Virginia) from the region chooser in the main navigation menu of the Management Console.

3.8 Optimizing costs for virtual machines

Usually you launch virtual machines *on demand* in the cloud to gain maximum flexibility. AWS calls them on-demand instances, because you can start and stop VMs on-demand, whenever you like, and you're billed for every second or hour the machine is running.

SIDE BAR**Billing unit: Seconds**

Most EC2 instances running Linux (such as Amazon Linux or Ubuntu) are billed per second. The minimum charge per instance is 60 seconds. For example, if you terminate a newly launched instance after 30 seconds, you have to pay for 60 seconds. But if you terminate an instance after 61 seconds, you pay exactly for 61 seconds.

EC2 instances running Amazon Linux, Windows, and Ubuntu are billed per second. Other instances are billed per hour. See aws.amazon.com/ec2/pricing/ for details.

Besides stopping or downsizing EC2 instances, there are two options to reduce costs: *savings plans* and *spot instances*. Both help to reduce costs but decrease your flexibility. With a spot instance, leverage unused capacity in an AWS data center. A spot instance comes with a discount

of up to 90% compared to on-demand instances. However, AWS might terminate a spot instance anytime when the resources are needed for someone else. So spot instances are for stateless and fault-tolerant workloads only. With savings plans you commit to a certain amount of resource consumption for 1 or 3 years and get a discount in turn. Therefore, savings plans are a good fit, in case you are running workloads with planning security. Table 3.2 summarizes the differences between the pricing options.

Table 3.2 Differences between on-demand instances, savings plans, and spot instances

	On-demand Instances	Savings Plans	Spot Instances
Price	High	Medium	Low
Flexibility	High	Low	Medium
Reliability	High	High	Low
Scenarios	Dynamic workloads (for example, for a news site) or proof-of-concept	Predictable and static workloads (for example, for a business application)	Batch workloads (for example, for data analytics, media encoding, ...)

3.8.1 Commit to usage, get discount

AWS offers two types of savings plans for EC2:

1. **Compute Savings Plans** do not apply to EC2 only, but to Fargate (Container), and Lambda (Serverless) as well.
2. **EC2 Instance Savings Plans** apply to EC2 instances only.

When purchasing a Compute Savings Plan, you need to specify the following details:

1. **Term:** 1-year or 3-year
2. **Hourly commitment:** in USD
3. **Payment Option:** all/partial/no upfront

For example, when committing to \$1 per hour for 1 year and paying \$8,760.00 upfront, you will get an m5.large EC2 instance at a discount of 31% in US East (N. Virginia). As you might have guessed already, the discount between on-demand and savings plans differs based on term, payment option, and even region. Find more details at aws.amazon.com/savingsplans/compute-pricing/.

An EC2 Instance Savings Plan applies to EC2 instances only. Therefore, it does not provide the flexibility to migrate a workload from virtual machines (EC2) to containers (Fargate). However, EC2 Instance Savings Plans offer a higher discount compared to Compute Savings Plans.

When purchasing an EC2 Instance Savings Plan, you need to specify the following details:

1. **Term:** 1-year or 3-year
2. **Region:** US East (N. Virginia), for example.
3. **Instance Family:** m5, for example

4. **Hourly commitment:** in USD
5. **Payment Option:** all/partial/no upfront

So the savings plan only applies to EC2 instances of a certain instance family in a certain region. Note, that you are able to modify the instance family of a savings plan later, if needed.

Let's look at the example from above again. When committing to \$1 per hour for 1 year of `m5` instances running in `us-east-1` and paying \$8,760.00 upfront, you will get an `m5.large` EC2 instance at a discount of 42% in US East (N. Virginia). Compare that to the 31% discount when purchasing a Compute Savings Plan instead.

WARNING **Buying a reservation will incur costs for 1 or 3 years. That's why we did not add an example for this section.**

Think of savings plans as a way to optimize your AWS bill. Buying a savings plan does not have any effect on the running EC2 instances. Also, an on-demand instance gets billed under the conditions of a savings plan automatically. There is no need to restart or modify an EC2 instance.

SIDE BAR Capacity Reservations

As mentioned, savings plans do not have any effect on EC2 instances but on billing only. In contrast, EC2 Capacity Reservations allow you to reserve capacities in AWS's data centers. A capacity reservation comes with an hourly fee. For example, when you reserve the capacity for an `m5.large` instance, you will pay the typical on-demand fee of \$0.096 per hour, no matter if an EC2 instance of that type is running or not. In return, AWS guarantees the capacity to launch an `m5.large` instance anytime.

With on-demand instances, you might run into the situation, that AWS cannot fulfill your request of spinning up a virtual machine. A situation that might happen for rare instance types, during peak hours, or special situations like outages causing many AWS customers to replace their failed instances.

In case you need to guarantee, that you are able to spin up an EC2 instance anytime, consider EC2 Capacity Reservations.

In summary, we highly recommend purchasing savings plans for workloads, where predicting the resource consumption for the next year is possible. It is worth noting, that it is not necessary to cover 100% of your usage with savings plans. Reducing costs is also possible, by committing to a smaller fraction of your workload.

3.8.2 Leveraging spare compute capacity

AWS is operating data centers at large scale. Thereby emerges spare capacity. For example, because AWS has to build and provision data centers and machines in advance to be able to fulfill future needs for on-demand capacity. But spare capacity does not generate revenue. That's why AWS tries to reduce spare capacity within their data centers. One of doing so, is offering spot instances.

Here is the deal. With spot instances you get a significant discount on the on-demand price without the need to commit using capacity in advance. In turn, a spot instance will start only when AWS decides that there is enough spare capacity available. In addition, a spot instance might be terminated by AWS at any time on short notice.

For example, when writing this on April 8th, 2022 the spot price for an `m5.large` instance in US East (N. Virginia) is \$0.039 per hour. That's a discount of about 60% compared to the \$0.096 on-demand price. The spot price for EC2 instances used to be very volatile. Nowadays, the spot price changes much more slowly.

But who is crazy enough to use virtual machines that might be terminated by AWS at any time with notice two minutes before the machine gets interrupted? Here are a few scenarios:

- Scanning objects stored on S3 for viruses and malware, by processing tasks stored in a queue.
- Converting media files into different formats, where the process orchestrator will restart failed jobs automatically.
- Processing parts of the requests for a web application, when the system is designed for fault-tolerance.

On top of that, we are using spot instances for test systems where dealing with short outages is worth the cost savings.

As discussed in the previous section, using savings plans does not require any changes to your EC2 instances. But to use spot instances, you have to launch new EC2 instances and also plan for interrupted virtual machines.

Next, you will launch your first spot instance.

1. Go to EC2 in the AWS Management Console: console.aws.amazon.com/ec2/.
2. Select *Spot Requests* from the sub navigation.
3. Click the *Request Spot Instances* button.
4. Select *Manually configure launch parameters* as shown in figure [Figure 3.31](#).
5. Choose an *Amazon Linux 2 AMI*.
6. Do not configure a key pair, select *(optional)* instead.
7. Expand the additional launch parameters section.
8. Select the IAM instance profile `ec2-ssm-core` to be able to connect to the spot instance

by using the Session Manager.

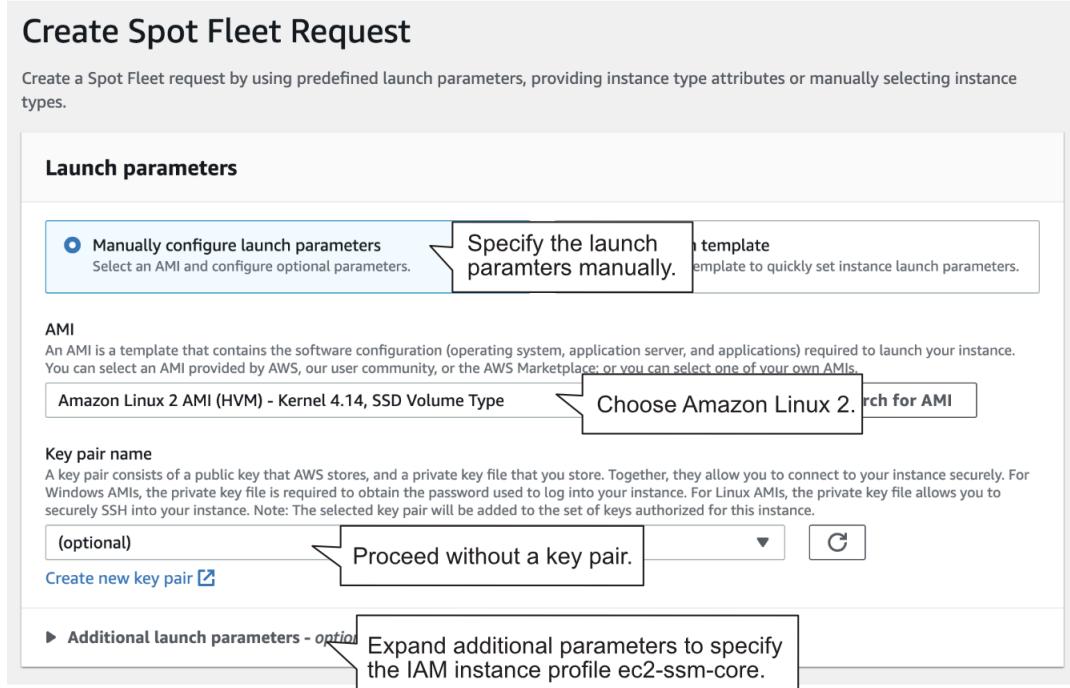


Figure 3.31 Step 1 of creating a spot fleet request

9. Keep the defaults for *Addition request details*.
10. Set the total target capacity to 1 instance as demonstrated in figure [Figure 3.32](#).

Additional request details

Apply defaults Keep the defaults.

IAM fleet role
[aws-ec2-spot-fleet-tagging-role](#)

Keep the request valid from now for 1 year
Terminate the instances when the request expires

Target capacity

Total target capacity
Set your total target capacity (number of instances or vCPUs) to launch. If you specified a launch template, you can allocate part of the target capacity as On-Demand. The number of On-Demand Instances always persists, while Spot Instances can be scaled.

A single instance is fine for testing. instances ▾

Include On-Demand base Allocate part of target capacity as On-Demand instances

Maintain target capacity Automatically replace interrupted Spot Instances

Set maximum cost for Spot Instances Set the maximum amount per hour that you're willing to pay for all the Spot Instances in your fleet

Figure 3.32 Step 2 of creating a spot fleet request

11. Chose *Manually select instance types* as shown in figure [Figure 3.33](#).
12. Empty the list of pre-populated instance types by selecting all instance types and clicking the *Delete* button.
13. Click the *Add instance types* button and select `t2.micro` from the list.

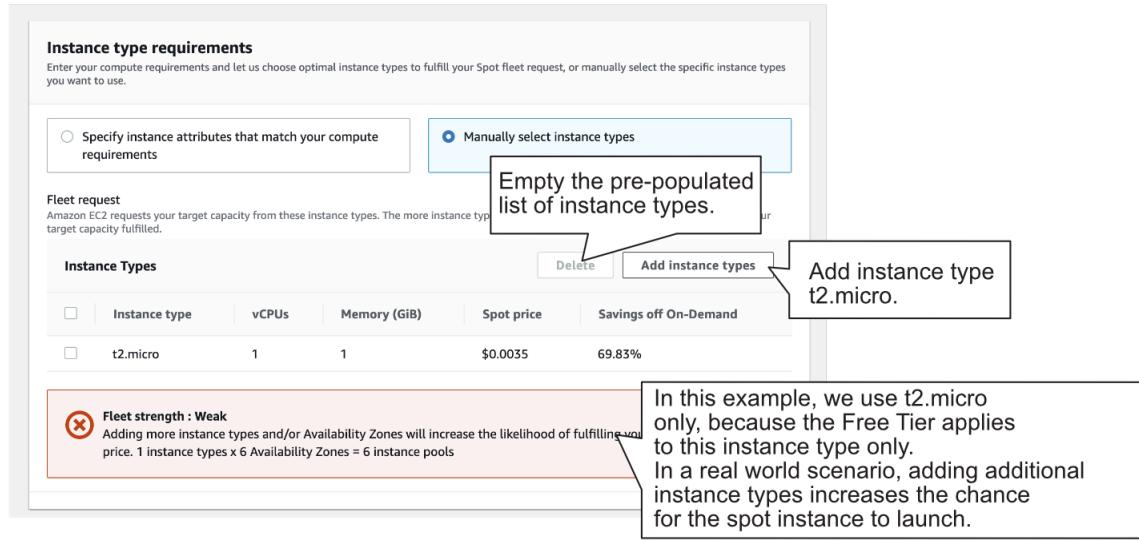


Figure 3.33 Step 3 of creating a spot fleet request

14. Choose allocation strategy *Capacity optimized* to increase the availability of a spot instance as shown in figure [Figure 3.34](#).
15. Press the *Launch* button to create a spot fleet request.

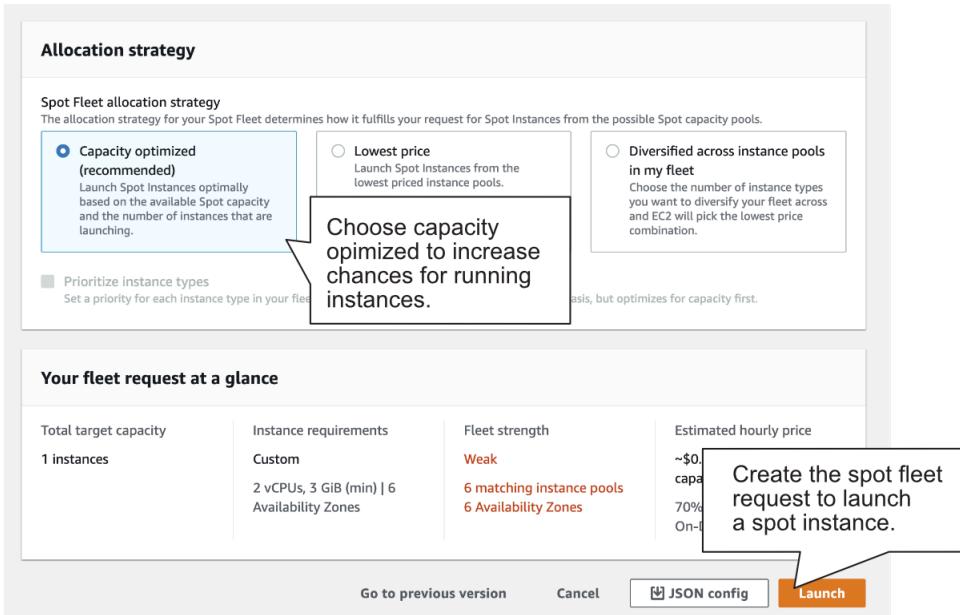


Figure 3.34 Step 4 of creating a spot fleet request

16. Two items appear in the list of spot requests. Wait until both the instance and fleet request reach status *fulfilled* as shown in figure [Figure 3.35](#).

Spot Requests						
<input type="button" value="C"/> Actions ▾ View						
<input type="text"/> Search for requests						
Request ID	Request type	Instance type	State	Capacity		
sir-8qjeaspq	instance	t2.micro	active	i-061bc...	fulfilled	one-time
sfr-01decfec-fe8...	fleet	t2.micro	active	1 of 1	fulfilled	request

Figure 3.35 Step 4 of creating a spot fleet request

17. Select *Instances* from the sub navigation.
18. The list of EC2 instances includes your first spot instance.

The spot instance is ready for your workload. But be aware of the fact, that AWS might terminate the spot instance anytime to free capacity for other workloads. AWS notifies you two minutes before terminating a spot instance. One way, to get notified about an interruption is to ask the EC2 metadata service about planned instance actions.

Use the Session Manager to connect with your EC2 instance and execute the following command

to send an HTTP request to the EC2 metadata service, which is only accessible from your virtual machine. Most likely, the HTTP request will result in a 404 error which is a good sign, because AWS did not mark your spot instance for termination.

```
$ curl http://169.254.169.254/latest/meta-data/spot/instance-action
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <title>404 - Not Found</title>
</head>
<body>
  <h1>404 - Not Found</h1>
</body>
</html>
sh-4.2$
```

In case, the HTTP request results into something like shown in the following snippet, your instance will be terminated within two minutes.

```
{"action": "stop", "time": "2022-04-08T12:12:00Z"}
```

In summary, spot instances help AWS to reduce spare capacity in their data centers and save us costs. However, you need to make sure that your application tolerates interruptions of spot instances which might cause engineering effort.

SIDEBAR
Cleaning up


cleanup

Terminate the spot instance.

- 1. Go to the list of Spot Requests.**
- 2. Select the fleet request.**
- 3. Press the Action and Cancel request button.**
- 4. Make sure to tick the Terminate instances check box and click the Confirm button.**

3.9 Summary

- When launching a virtual machine on AWS, you chose between a wide variety of operating systems: Amazon Linux, Ubuntu, Windows, and many more.
- Modifying the size of a virtual machine is simple: stop the virtual machine, modify the instance type -which defines the number of CPUs, as well as the amount of memory and storage-, and start the virtual machine.
- Using logs and metrics can help you to monitor and debug your virtual machine.
- AWS offers data centers all over the world. Starting VMs in Sydney works the same as starting a machine in North Virginia.
- Choosing the data center by considering network latency, legal requirements, costs, as well as available features.
- Allocating and associating a public IP address to your virtual machine gives you the flexibility to replace a VM without changing the public IP address.
- Committing to a certain compute usage for 1 or 3 years to reduce costs of virtual machines by buying savings plans.
- Use spare capacity at significant discount but with the risk of AWS terminating your virtual machine in case the capacity is needed elsewhere.

4

Programming your infrastructure: The command-line, SDKs, and CloudFormation

This chapter covers

- Starting a virtual machine with the CLI
- Starting a virtual machine with JavaScript SDK for Node.js
- Understanding the idea of infrastructure as code
- Using CloudFormation to start a virtual machine

Imagine that you want to provide room lighting as a service. To switch off the lights in a room using software, you need a hardware device like a relay connected to the light circuit. This hardware device must have some kind of interface that lets you send commands via software. With a relay and an interface, you can offer room lighting as a service.

To run a virtual machine, a lot of hardware and software is needed. Power supply, networking gear, host machine, operating system, virtualization layer, and much more. Luckily, AWS runs the hardware and software for us. Even better, we can control all of that with software. AWS provides an *application programming interface (API)* that we can use to control every part of AWS with HTTPS requests. In the end, you can write software that spins up VMs on AWS as well as in-memory caches, data warehouses, and much more.

Calling the HTTP API is very low-level and requires a lot of repetitive work, like authentication, data (de)serialization, and so on. That's why AWS offers tools on top of the HTTP API that are easier to use. Those tools are:

- *Command-line interface (CLI)*—Use the CLI to call the AWS API from your terminal.
- *Software development kit (SDK)*—SDKs, available for most programming languages, make it easy to call the AWS API from your programming language of choice.
- *AWS CloudFormation*—Templates are used to describe the state of the infrastructure. AWS CloudFormation translates these templates into API calls.

SIDE BAR**Not all examples are covered by the Free Tier**

The examples in this chapter are not all covered by the Free Tier. A special warning message appears when an example incurs costs. As for the other examples, as long as you don't run them longer than a few days, you won't pay anything for them. Keep in mind that this applies only if you created a fresh AWS account for this book and nothing else is going on in your AWS account. Try to complete the chapter within a few days; you'll clean up your account at the end.

On AWS, everything can be controlled via an API. You interact with AWS by making calls to the REST API using the HTTPS protocol, as figure 1.9 illustrates. Everything is available through the API. You can start a virtual machine with a single API call, create 1 TB of storage, or start a Hadoop cluster over the API. By everything, we really mean *everything*. You'll need some time to understand the consequences. By the time you finish this book, you'll ask why the world wasn't always this easy.

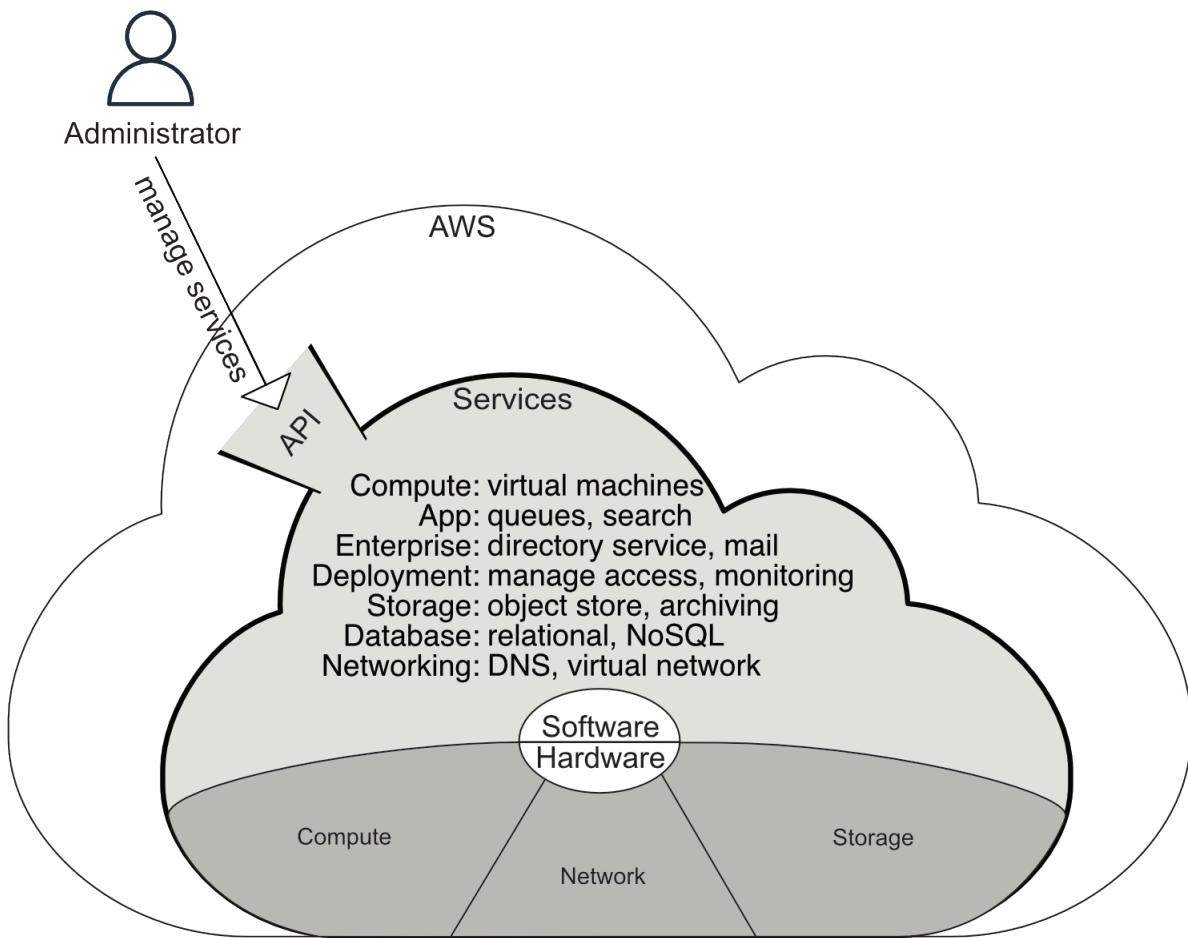


Figure 4.1 The AWS cloud is composed of hardware and software services accessible via an API.

Let's look at how the API works. Imagine you uploaded a few files to the object store S3 (you will learn about S3 in chapter 7). Now you want to list all the files in the S3 object store to check if the upload was successful. Using the raw HTTP API, you send a GET request to the API endpoint using the HTTP protocol:

```
GET / HTTP/1.1
Host: BucketName.s3.amazonaws.com
Authorization: [...]
```

- ➊ HTTP method GET, HTTP resource /, using HTTP protocol version 1.1
- ➋ Specifies the host name; keep in mind that TCP/IP only knows about IP addresses and ports.
- ➌ Authentication information (details omitted)

The HTTP response will look like this:

```
HTTP/1.1 200 OK          ①
x-amz-id-2: [...]
x-amz-request-id: [...]
Date: Mon, 09 Feb 2015 10:32:16 GMT      ②
Content-Type: application/xml            ③

<?xml version="1.0" encoding="UTF-8"?>    ④
<ListBucketResult xmlns="http://s3.amazonaws.com/doc/2006-03-01/">
[...]
</ListBucketResult>
```

- ① Using HTTP protocol version 1.1; status code 200 signals a success.
- ② An HTTP header shows when the response was generated.
- ③ The response body is an XML document.
- ④ The response body starts here.

Calling the API directly using plain HTTPS requests is inconvenient. The easy way to talk to AWS is by using the CLI or SDKs, as you learn in this chapter. But the API is the foundation of all those tools.

4.1 Automation and the DevOps movement

The *DevOps movement* aims to bring software development and operations together. This usually is accomplished in one of two ways:

- Using mixed teams with members from both operations and development. Developers become responsible for operational tasks like being on-call. Operators are involved from the beginning of the software development cycle, which helps make the software easier to operate.
- Introducing a new role that closes the gap between developers and operators. This role communicates a lot with both developers and operators, and cares about all topics that touch both worlds.

The goal is to develop and deliver software to the customer rapidly without a negative impact on quality. Communication and collaboration between development and operations are therefore necessary.

The trend toward automation has helped DevOps culture bloom, as it codifies the cooperation between development and operations. You can only do multiple deployments per day if you automate the whole process. If you commit changes to the repository, the source code is automatically built and tested against your automated tests. If the build passes the tests, it's automatically installed in your testing environment. This triggers some acceptance tests. After

those tests have been passed, the change is propagated into production. But this isn't the end of the process; now you need to carefully monitor your system and analyze the logs in real time to ensure that the change was successful.

If your infrastructure is automated, you can spawn a new system for every change introduced to the code repository and run the acceptance tests isolated from other changes that were pushed to the repository at the same time. Whenever a change is made to the code, a new system is created (virtual machine, databases, networks, and so on) to run the change in isolation.

4.2 Why should you automate?

Why should you automate instead of using the graphical AWS Management Console? A script or a blueprint can be reused, and will save you time in the long run. You can build new infrastructures quickly with ready-to-use modules from your former projects, or automate tasks that you will have to do regularly. Automating your infrastructure also enhances your software development process, for example by using of a deployment pipeline.

Another benefit is that a script or blueprint is the most detailed documentation you can imagine (even a computer understands it). If you want to reproduce on Monday what you did last Friday, a script is worth its weight in gold. If you're sick and a coworker needs to take care of your tasks, they will appreciate your blueprints.

You're now going to install and configure the CLI. After that, you can get your hands dirty and start scripting.

The AWS CLI is one tool for automating AWS. Read on to learn how it works.

4.3 Using the command-line interface

The AWS CLI is a convenient way to interact with AWS from your terminal. It runs on Linux, macOS, and Windows. It provides a unified interface for all AWS services. Unless otherwise specified, the output is by default in JSON format.

4.3.1 Installing the CLI

How you proceed depends on your OS. If you're having difficulty installing the CLI, consult docs.aws.amazon.com/cli/latest/userguide/cli-chap-getting-started.html for a detailed description of many installation options.

LINUX X86 (64-BIT)

In your terminal, execute the following commands:

```
$ curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip"
$ unzip awscliv2.zip
$ sudo ./aws/install
```

Verify your AWS CLI installation by running `aws --version` in your terminal. The version should be at least 2.4.0.

LINUX ARM

In your terminal, execute the following commands:

```
$ curl "https://awscli.amazonaws.com/awscli-exe-linux-aarch64.zip" -o "awscliv2.zip"
$ unzip awscliv2.zip
$ sudo ./aws/install
```

Verify your AWS CLI installation by running `aws --version` in your terminal. The version should be at least 2.4.0.

MACOS

The following steps guide you through installing the AWS CLI on macOS using the installer:

1. Download the AWS CLI [installer](#).
2. Run the downloaded installer, and install the CLI by going through the installation wizard for **all users**.
3. Verify your AWS CLI installation by running `aws --version` in your terminal. The version should be at least 2.4.0.

WINDOWS

The following steps guide you through installing the AWS CLI on Windows using the MSI Installer:

1. Download the AWS CLI [installer](#).
2. Run the downloaded installer, and install the CLI by going through the installation wizard.
3. Run PowerShell as administrator by searching for “PowerShell” in the Start menu and choosing Run as Administrator from its context menu.
4. Type `Set-ExecutionPolicy Unrestricted` into PowerShell, and press Enter to execute the command. This allows you to execute the unsigned PowerShell scripts from our examples.
5. Close the PowerShell window; you no longer need to work as administrator.
6. Run PowerShell by choosing PowerShell from the Start menu.
7. Verify whether the CLI is working by executing `aws --version` in PowerShell. The version should be at least 2.4.0.

WARNING Setting the PowerShell execution policy to `Unrestricted` allows you to run unsigned scripts. There is a risk of running malicious scripts. Use it to run the scripts provided in our examples only. Check [about_Execution_Policies](#) to learn more.

4.3.2 Configuring the CLI

To use the CLI, you need to authenticate. Until now, you've been using the root AWS account. This account can do everything, good and bad. It's strongly recommended that you not use the AWS root account (you'll learn more about security in chapter 5), so let's create a new user.

To create a new user, follow the steps in figure [4.2](#):

1. Open the AWS Management Console at console.aws.amazon.com.
2. Click on *Services* and search for *IAM*.
3. Open the IAM service.

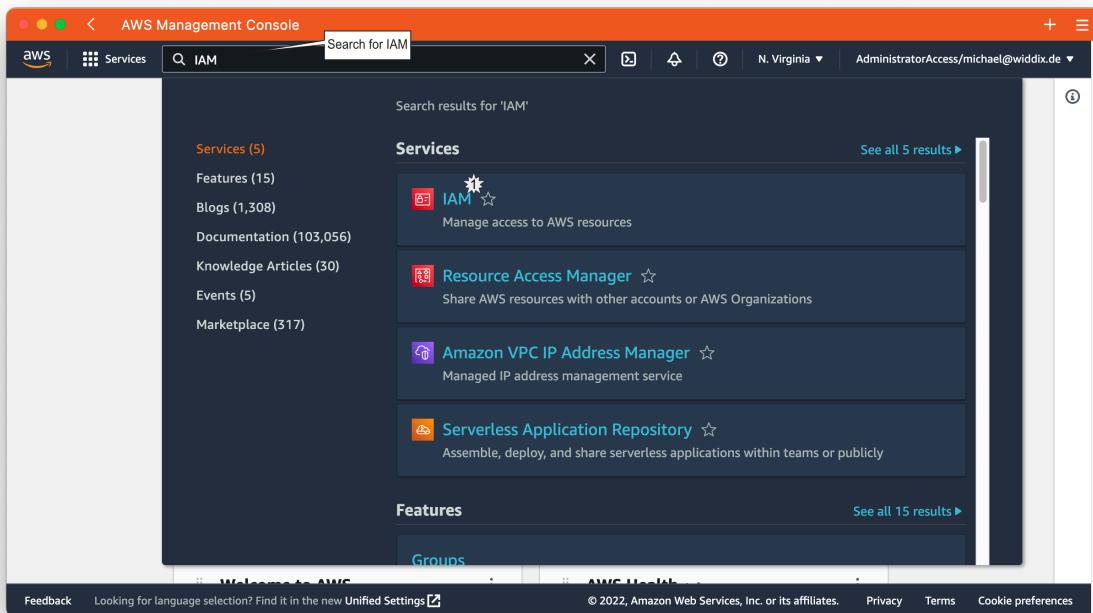


Figure 4.2 Open the IAM service

A page opens as shown in figure [4.3](#); select Users at left.

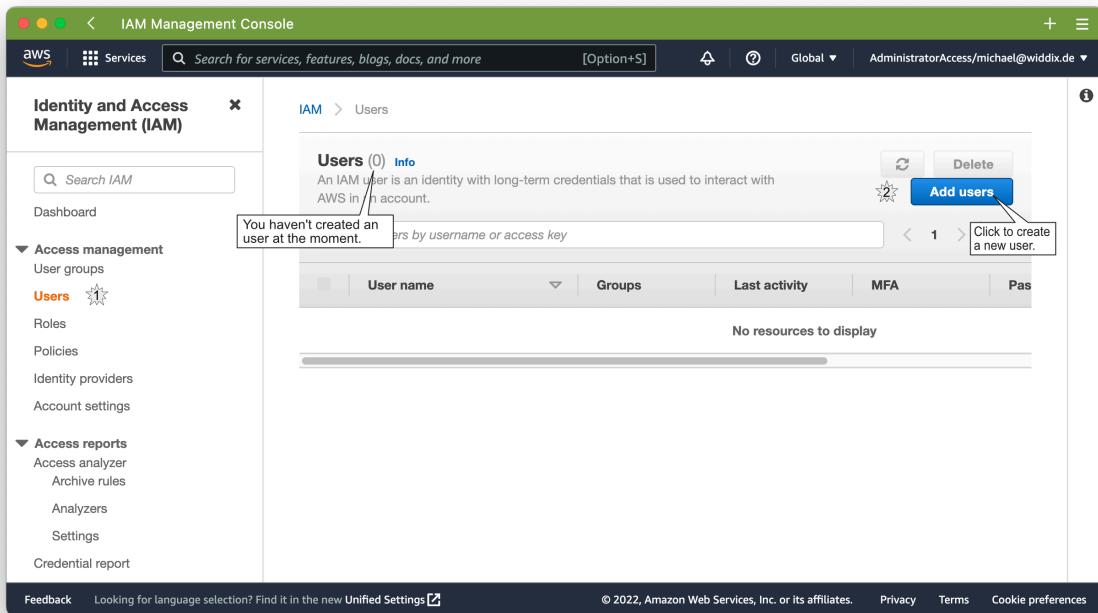


Figure 4.3 IAM users (empty)

Follow these steps to create a user:

1. Click *Add Users* to open the page shown in figure 4.4.
2. Enter `mycli` as the user name.
3. Under AWS credential type, select *Access key - Programmatic access*.
4. Click the *Next: Permissions* button.

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*	mycli	User name of the new user is mycli.
------------	-------	-------------------------------------

[Add another user](#)

Select AWS access type

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

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

* Required [Cancel](#) [Next: Permissions](#)

Figure 4.4 Creating an IAM user

In the next step, you have to define the permissions for the new user, as shown in figure 4.5.

1. Click *Attach Existing Policies Directly*.
2. Select the *AdministratorAccess* policy.
3. Click the *Next: Tags* button.

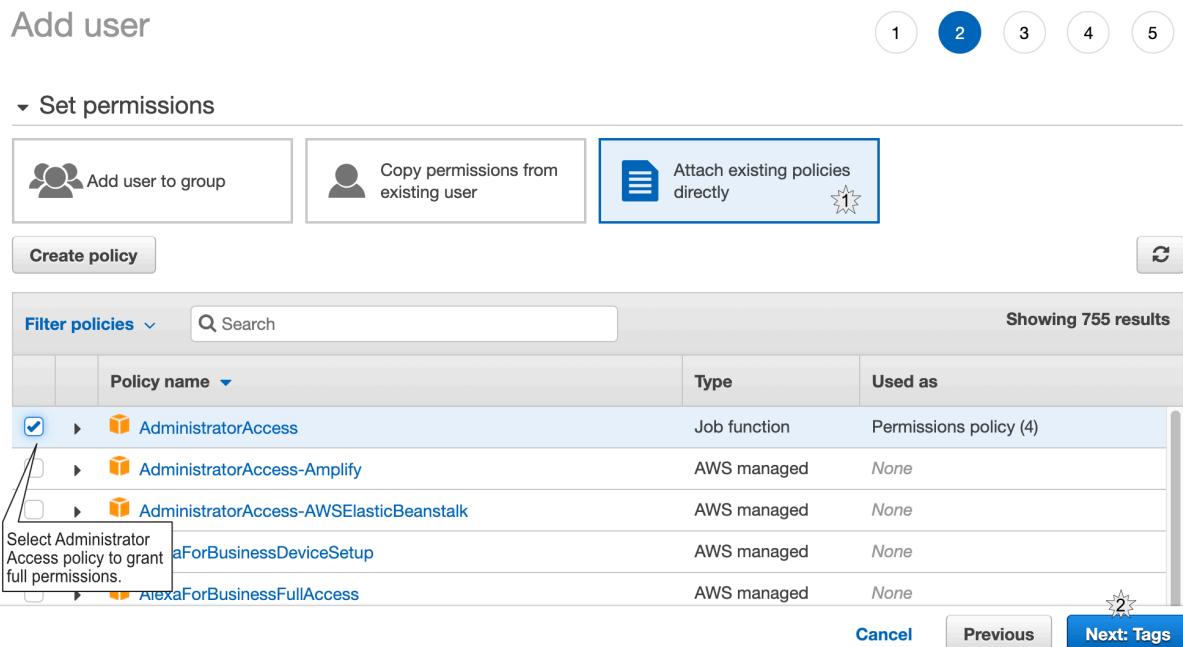


Figure 4.5 Setting permissions for an IAM user

No tags are needed (you learned about tags in chapter 2), click the *Next: Review* button.

The review page sums up what you have configured. Click the *Create user* button to save. Finally, you will see the page shown in figure 4.6. Click the *Show* link to display the secret value. You now need to copy the credentials to your CLI configuration. Read on to learn how this works.

WARNING Treat the access key ID and secret access key as top secret. Anyone who gets access to this credentials will have administrator access to your AWS account.

Add user

1 2 3 4 5

Success

You successfully created the users shown below. You can view and download user security credentials. You can also email users instructions for signing in to the AWS Management Console. This is the last time these credentials will be available to download. However, you can create new credentials at any time.

Users with AWS Management Console access can sign-in at: <https://widdix-dennis.signin.aws.amazon.com/console>

Download .csv

	User	Access key ID	Secret access key
▶	mycli	AKIAWJCZVSFGDCL6CVGK	***** Show

Close

Figure 4.6 Showing access key of an IAM user

Open the terminal on your computer (PowerShell on Windows or a shell on Linux and macOS, not the AWS Management Console), and run `aws configure`. You’re asked for four pieces of information:

1. *AWS access key ID*—Copy and paste this value from the Access key ID column (your browser window).
2. *AWS secret access key*—Copy and paste this value from the Secret access key column (your browser window).
3. *Default region name*—Enter `us-east-1`.
4. *Default output format*—Enter `json`.

In the end, the terminal should look similar to this:

```
$ aws configure
AWS Access Key ID [None]: AKIAIRUR3YLPOSVD7ZCA ①
AWS Secret Access Key [None]: SSKIng7jkAKERpcT3YphX4cD87sBYgWVw2enqBj7 ②
Default region name [None]: us-east-1
Default output format [None]: json
```

- ① Your value will be different! Copy it from your browser window.
- ② Your value will be different! Copy it from your browser window.

The CLI is now configured to authenticate as the user `mycli`. Switch back to the browser window, and click **Close** to finish the user-creation wizard.

It’s time to test whether the CLI works. Switch to the terminal window, and enter `aws ec2 describe-regions` to get a list of all available regions:

```
$ aws ec2 describe-regions
{
  "Regions": [
    {
      "Endpoint": "ec2.eu-north-1.amazonaws.com",
      "RegionName": "eu-north-1",
      "OptInStatus": "opt-in-not-required"
    },
    [...]
  {
    "Endpoint": "ec2.us-west-2.amazonaws.com",
    "RegionName": "us-west-2",
    "OptInStatus": "opt-in-not-required"
  }
]
```

It works! You can now begin to use the CLI.

4.3.3 Using the CLI

Suppose you want to get a list of all running EC2 instances of type t2.micro so you can see what is running in your AWS account. Execute the following command in your terminal:

```
$ aws ec2 describe-instances --filters "Name=instance-type,Values=t2.micro"
{
  "Reservations": [] ❶
}
```

- ❶ Empty list because you haven't created an EC2 instance

To use the AWS CLI, you need to specify a service and an action. In the previous example, the service is `ec2` and the action is `describe-instances`. You can add options with `--name value`:

```
$ aws <service> <action> [--name value ...]
```

One important feature of the CLI is the `help` keyword. You can get help at three levels of detail:

- `aws help`—Shows all available services.
- `aws <service> help`—Shows all actions available for a certain service.
- `aws <service> <action> help`—Shows all options available for the particular service action.

4.3.4 Automating with the CLI

SIDE BAR

IAM role ec2-ssm-core

The following script requires an IAM role named `ec2-ssm-core`. You created the role in [Creating an IAM Role](#).

Sometimes you need temporary computing power, like a Linux machine to test something. To do this, you can write a script that creates a virtual machine for you. The script will run on your local computer and does the following:

1. Starts a virtual machine.
2. Helps you to connect to the VM via Session Manager.
3. Waits for you to finish your temporary usage of the VM.
4. Terminates the virtual machine.

The script is used like this:

```
$ ./virtualmachine.sh
waiting for i-08f21510e8c4f4441 ...           ①
i-08f21510e8c4f4441 is up and running
connect to the instance using Session Manager
https://console.aws.amazon.com/systems-manager/ses[...]441
Press [Enter] key to terminate i-08f21510e8c4f4441 ...
terminating i-08f21510e8c4f4441 ...           ②
done.
```

- ① Waits until started
- ② Waits until terminated

Your virtual machine runs until you press the Enter key. When you press Enter, the virtual machine is terminated.

The CLI solution solves the following use cases:

- Creating a virtual machine.
- Getting the id of a virtual machine to connect via Session Manager.
- Terminating the virtual machine if it's no longer needed.

Depending on your OS, you'll use either Bash (Linux and macOS) or PowerShell (Windows) to script.

One important feature of the CLI needs explanation before you can begin. The `--query` option uses JMESPath syntax, which is a query language for JSON, to extract data from the result. This can be useful because usually you only need a specific field from the result. The following CLI command gets a list of all AMIs in JSON format.

SIDE BAR

Dealing with Long output

By default, the AWS CLI returns all output through your operating system's default pager program (on my system this is `less`). This is helpful to avoid massive amounts of data to be printed to your terminal.

There is one thing to know: To quit `less`, hit `q` and you will be back where you issued the AWS CLI command.

```
$ aws ec2 describe-images --filters \
[CA] "Name=name,Values=amzn2-ami-hvm-2.0.202*-x86_64-gp2" ①
{
  "Images": [
    {
      "ImageId": "ami-0ce1e3f77cd41957e",
      "State": "available"
      [...]
    },
    [...]
    {
      "ImageId": "ami-08754599965c30981",
      "State": "available"
    }
  ]
}
```

- ① The filter returns only Amazon Linux 2 images for AMD/Intel

The output is overwhelming. To start an EC2 instance, you need the `ImageId` without all the other information. With JMESPath, you can extract just that information.

```
$ aws ec2 describe-images --filters \
[CA] "Name=name,Values=amzn2-ami-hvm-2.0.202*-x86_64-gp2" \
[CA] --query "Images[0].ImageId" ①
"ami-146e2a7c"
```

- ① Return the first image id from the list

The output is wrapped in quotes. This is caused by the default setting of the AWS CLI to output all data in JSON format. JSON strings are enclosed in quotes. To change that, the `--output text` option can be used to format the output as multiple lines of tab-separated string values. This can be useful to pass the output to a text processor, like `grep`, `sed`, or `awk`.

```
$ aws ec2 describe-images --filters \
[CA] "Name=name,Values=amzn2-ami-hvm-2.0.202*-x86_64-gp2" \
[CA] --query "Images[0].ImageId" --output text ①
ami-146e2a7c
```

- ① Set the output format to plain text

With this short introduction to JMESPath, you're well equipped to extract the data you need.

SIDEBAR

Where is the code located?

All code can be found in the book's code repository on GitHub: github.com/AWSinAction/code3. You can download a snapshot of the repository at github.com/AWSinAction/code3/archive/main.zip.

Linux and macOS can interpret Bash scripts, whereas Windows prefers PowerShell scripts. So, we've created two versions of the same script.

LINUX AND MACOS

You can find the following listing in `/chapter04/virtualmachine.sh` in the book's code folder. You can run it either by copying and pasting each line into your terminal or by executing the entire script via:

```
chmod +x virtualmachine.sh ①
./virtualmachine.sh
```

- ① Ensures that the script is executable (only required once)

Listing 4.1 Creating and terminating a virtual machine from the CLI (Bash)

```
#!/bin/bash -e      ①
AMIID=$(aws ec2 describe-images --filters \
[CA] "Name=name,Values=amzn2-ami-hvm-2.0.202*-x86_64-gp2" \
[CA] --query "Images[0].ImageId" --output text)
VPCID=$(aws ec2 describe-vpcs --filter "Name=isDefault, Values=true" \
[CA] --query "Vpcs[0].VpcId" --output text)
SUBNETID=$(aws ec2 describe-subnets --filters \
[CA] "Name=vpc-id, Values=$VPCID" --query "Subnets[0].SubnetId" \
[CA] --output text)
INSTANCEID=$(aws ec2 run-instances --image-id "$AMIID" \
[CA] --instance-type t2.micro --subnet-id "$SUBNETID" \
[CA] --iam-instance-profile "Name=ec2-ssm-core" \
[CA] --query "Instances[0].InstanceId" --output text)
echo "waiting for $INSTANCEID ..."
aws ec2 wait instance-running --instance-ids "$INSTANCEID" ⑥
echo "$INSTANCEID is up and running"
echo "connect to the instance using Session Manager"
echo "https://console.aws.amazon.com/systems-manager/session-manager/$INSTANCEID"
read -r -p "Press [Enter] key to terminate $INSTANCEID ..."
aws ec2 terminate-instances --instance-ids "$INSTANCEID" > /dev/null ⑦
echo "terminating $INSTANCEID ..."
aws ec2 wait instance-terminated --instance-ids "$INSTANCEID" ⑧
echo "done."
```

- ① `-e` makes Bash abort if a command fails.
- ② Get the ID of Amazon Linux AMI.
- ③ Get the default VPC ID.
- ④ Get the default subnet ID.
- ⑤ Create and start the virtual machine.
- ⑥ Wait until the virtual machine is started.
- ⑦ Terminate the virtual machine.
- ⑧ Wait until the virtual machine is terminated.

SIDE BAR**Cleaning up**

cleanup

Make sure you terminate the virtual machine before you go on by pressing the enter key!

WINDOWS

You can find the following listing in `/chapter04/virtualmachine.ps1` in the book's code folder. Right-click the `virtualmachine.ps1` file, and select Run with PowerShell to execute the script.

Listing 4.2 Creating and terminating a virtual machine from the CLI (PowerShell)

```
$ErrorActionPreference = "Stop"          ①

$AMIID=aws ec2 describe-images --filters \
[CA] "Name=name,Values=amzn2-ami-hvm-2.0.202*-x86_64-gp2" \
[CA] --query "Images[0].ImageId" --output text
$VPCID=aws ec2 describe-vpcs --filter "Name=isDefault, Values=true" \
[CA] --query "Vpcs[0].VpcId" --output text
$SUBNETID=aws ec2 describe-subnets \
[CA] --filters "Name=vpc-id, Values=$VPCID" --query "Subnets[0].SubnetId" \
[CA] --output text
$instanceID=aws ec2 run-instances --image-id $AMIID \
[CA] --instance-type t2.micro --subnet-id $SUBNETID \
[CA] --iam-instance-profile "Name=ec2-ssm-core" \
[CA] --query "Instances[0].InstanceId" --output text
Write-Host "waiting for $instanceID ..."
aws ec2 wait instance-running --instance-ids $instanceID      ⑥
Write-Host "$instanceID is up and running"
Write-Host "connect to the instance using Session Manager"
Write-Host "https://console.aws.amazon.com/systems-manager/session-manager/$instanceID"
Write-Host "Press [Enter] key to terminate $instanceID ..."
Read-Host
aws ec2 terminate-instances --instance-ids $instanceID      ⑦
Write-Host "terminating $instanceID ..."
aws ec2 wait instance-terminated --instance-ids $instanceID    ⑧
Write-Host "done."
```

- ① Abort if the command fails.
- ② Get the ID of Amazon Linux AMI.
- ③ Get the default VPC ID.
- ④ Get the default subnet ID.
- ⑤ Create and start the virtual machine.
- ⑥ Wait until the virtual machine is started.
- ⑦ Terminate the virtual machine.
- ⑧ Wait until the virtual machine is terminated

SIDE BAR**Cleaning up**

cleanup

Make sure you terminate the virtual machine before you go on

The limitations of the CLI solution are as follows:

- It can handle only one virtual machine at a time.
- There is a different version for Windows than for Linux and macOS.
- It's a command-line application, not graphical.

Learn how to improve the CLI solution using the AWS SDK next.

4.4 Programming with the SDK

AWS offers SDKs for a number of programming languages and platforms:

• JavaScript / Node.js	• Java	• .NET
• PHP	• Python	• Ruby
• Go	• C++	

An AWS SDK is a convenient way to make calls to the AWS API from your favorite programming language. The SDK takes care of things like authentication, retry on error, HTTPS communication, and XML or JSON (de)serialization. You're free to choose the SDK for your favorite language, but in this book most examples are written in JavaScript and run in the Node.js runtime environment.

SIDE BAR**Installing and getting started with Node.js**

Node.js is a platform for executing JavaScript in an event-driven environment so you can easily build network applications. To install Node.js, visit nodejs.org and download the package that fits your OS. All examples in this book are tested with Node.js 14.

After Node.js is installed, you can verify if everything works by typing `node --version` into your terminal. Your terminal should respond with something similar to `v14.*`. Now you're ready to run JavaScript examples, like the Node Control Center for AWS. Your Node.js installation comes with an important tool called npm, which is the package manager for Node.js. Verify the installation by running `npm --version` in your terminal.

To run a JavaScript script in Node.js, enter `node script.js` in your terminal, where `script.js` is the name of the script file. We use Node.js in this book because it's easy to install, it requires no IDE, and the syntax is familiar to most programmers.

Don't be confused by the terms JavaScript and Node.js. If you want to be precise, JavaScript is the language and Node.js is the runtime environment. But don't expect anybody to make that distinction. Node.js is also called node.

Do you want to get started with Node.js? We recommend *Node.js in Action* (2nd Edition) from Alex Young, et al. (Manning, 2017), or the video course *Node.js in Motion* from PJ Evans, (Manning, 2018).

To understand how the AWS SDK for Node.js (JavaScript) works, let's create a Node.js (JavaScript) application that controls EC2 instances via the AWS SDK. The name of the application is *Node Control Center for AWS* or *nodecc* for short.

4.4.1 Controlling virtual machines with SDK: nodecc

The *Node Control Center for AWS (nodecc)* is for managing multiple temporary EC2 instances using a text UI written in JavaScript. *nodecc* has the following features:

- It can handle multiple virtual machines.
- It's written in JavaScript and runs in Node.js, so it's portable across platforms.
- It uses a textual UI.

Figure [4.7](#) shows what *nodecc* looks like.

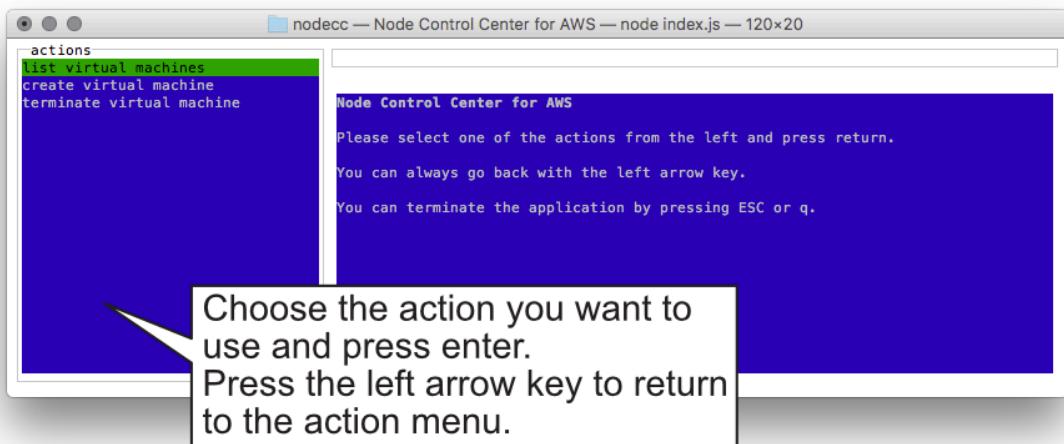


Figure 4.7 nodecc: start screen

SIDE BAR	IAM role ec2-ssm-core nodecc requires an IAM role named ec2-ssm-core. You created the role in Creating an IAM Role .
-----------------	--

You can find the *nodecc* application at `/chapter04/nodecc/` in the book's code folder. Switch to that directory, and run `npm install` in your terminal to install all needed dependencies. To start *nodecc*, run `node index.js`. You can always go back by pressing the left arrow key. You can quit the application by pressing Esc or q. The SDK uses the same settings you created for the CLI, so you're using the `mycli` user when running *nodecc*.

4.4.2 How nodecc creates a virtual machine

Before you can do anything with *nodecc*, you need at least one virtual machine. To start a virtual machine, choose the AMI you want, as figure 4.8 shows.

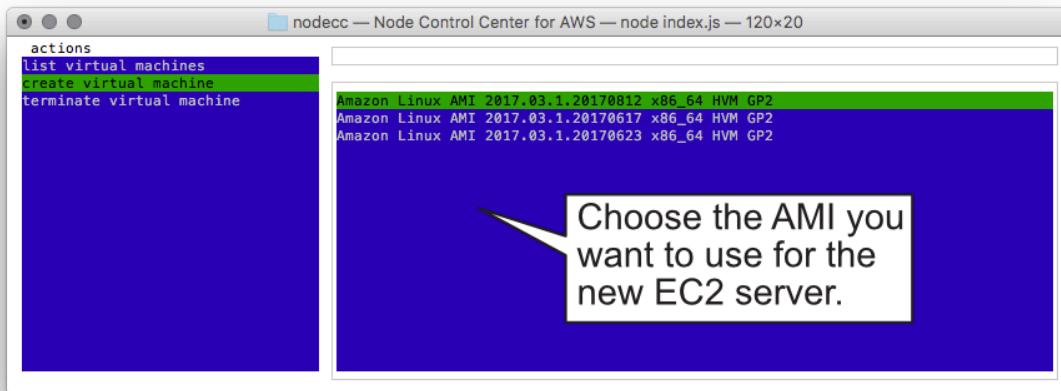


Figure 4.8 nodecc: creating a virtual machine (step 1 of 2)

The code that fetches the list of the available AMIs is located at `lib/listAMIs.js`.

Listing 4.3 Fetching the list of available AMIs /lib/listAMIs.js

```
const AWS = require('aws-sdk');      ①
const ec2 = new AWS.EC2({region: 'us-east-1'});    ②

module.exports = (cb) => {      ③
  ec2.describeImages({          ④
    Filters: [
      {
        Name: 'name',
        Values: ['amzn2-ami-hvm-2.0.202*-x86_64-gp2']
      }
    ], (err, data) => {
      if (err) {            ⑤
        cb(err);
      } else {              ⑥
        const amiIds = data.Images.map(image => image.ImageId);
        const descriptions = data.Images.map(image => image.Description);
        cb(null, {amiIds: amiIds, descriptions: descriptions});
      }
    });
};
```

- ① Require is used to load modules.
- ② Create an EC2 client.
- ③ module.exports makes this function available to users of the listAMIs module.
- ④ API call to list AMIs
- ⑤ In case of failure, err is set.
- ⑥ Otherwise, data contains all AMIs.

The code is structured in such a way that each action is implemented in the lib folder. The next step to create a virtual machine is to choose which subnet the virtual machine should be started in. You haven't learned about subnets yet, so for now select one randomly; see figure 4.9.

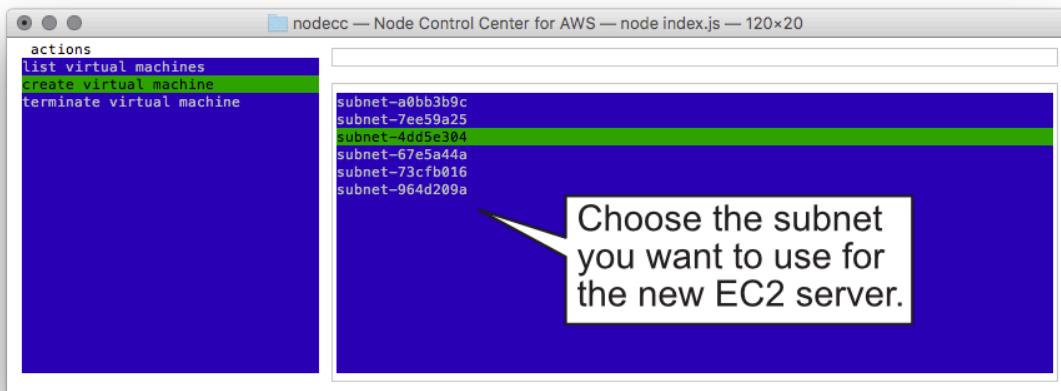


Figure 4.9 nodecc: creating a virtual machine (step 2 of 2)

The corresponding script is located at `lib/listSubnets.js`.

Listing 4.4 Fetching the list of available default subnets /lib/listSubnets.js

```
const AWS = /* ... */;
const ec2 = /* ... */;

module.exports = (cb) => {
  ec2.describeVpcs({    ①
    Filters: [{      ②
      Name: 'isDefault',
      Values: ['true']
    }]
  }, (err, data) => {
    if (err) {
      cb(err);
    } else {
      const vpcId = data.Vpcs[0].VpcId;    ③
      ec2.describeSubnets({    ④
        Filters: [{      ⑤
          Name: 'vpc-id',
          Values: [vpcId]
        }]
      }, (err, data) => {
        if (err) {
          cb(err);
        } else {
          const subnetIds = data.Subnets.map(subnet => subnet.SubnetId);
          cb(null, subnetIds);
        }
      });
    }
  });
};
```

- ① API call to list VPCs
- ② The filter selects default VPCs only
- ③ There can be only one default VPC
- ④ API call to list subnets

- ⑤ The filter selects subnets from the default VPC only

After you select the subnet, the virtual machine is created by `lib/createVM.js`, and you see a starting screen.

Listing 4.5 Launching an EC2 instance /lib/createVM.js

```
module.exports = (amiId, subnetId, cb) => {
  ec2.runInstances({
    ①
    IamInstanceProfile: { ②
      Name: 'ec2-ssm-core'
    },
    ImageId: amiId, ③
    MinCount: 1, ④
    MaxCount: 1, ④
    InstanceType: 't2.micro', ⑤
    SubnetId: subnetId ⑥
  }, cb);
};
```

- ① API call to launch an EC2 instance
- ② The IAM role and instance profile `ec2-ssm-core` was created in chapter 3.
- ③ Pass the selected AMI
- ④ Launch one instance
- ⑤ Of type `t2.micro` to stay in the free tier
- ⑥ Pass the selected subnet

Now it's time to find out some details of the newly created virtual machine. Use the left arrow key to switch to the navigation section.

4.4.3 How `nodecc` lists virtual machines and shows virtual machine details

One important use case that `nodecc` must support is showing the details of a VM that you can use to connect via Session Manager. Because `nodecc` handles multiple virtual machines, the first step is to select a VM, as shown in figure [4.10](#).

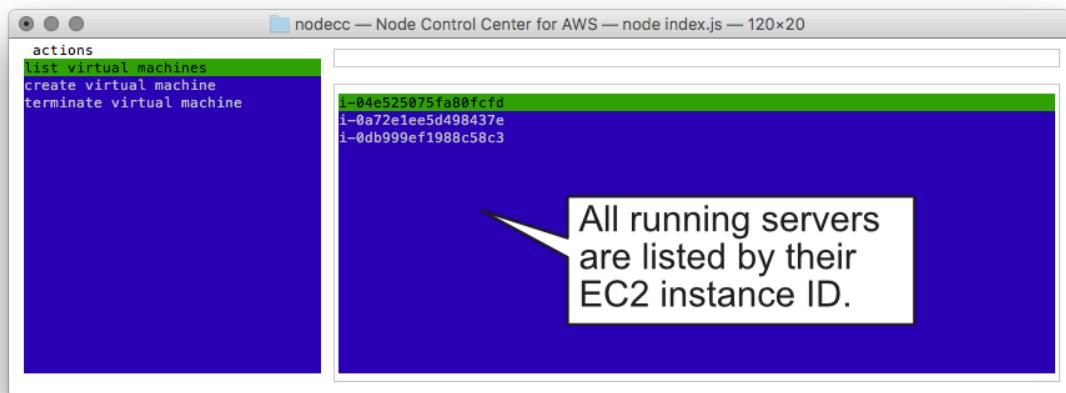


Figure 4.10 nodecc: listing virtual machines

Look at `lib/listVMs.js` to see how a list of virtual machines can be retrieved with the AWS SDK.

Listing 4.6 Listing EC2 instances /lib/listVMs.js

```
module.exports = (cb) => {
  ec2.describeInstances({ ①
    Filters: [{} ②
      Name: 'instance-state-name',
      Values: ['pending', 'running']
    ],
    MaxResults: 10 ③
  }, (err, data) => {
    if (err) {
      cb(err);
    } else {
      const instanceIds = data.Reservations
        .map(r => r.Instances.map(i => i.InstanceId))
        .flat();
      cb(null, instanceIds);
    }
  });
};
```

- ① API call to list EC2 instances
- ② The filter selects starting and running instances only
- ③ Show at most 10 instances

After you select the VM, you can display its details; see figure 4.11. You could use the `LaunchTime` to find old EC2 instances. You can connect to an instance using Session Manager. Press the left arrow key to switch back to the navigation section.

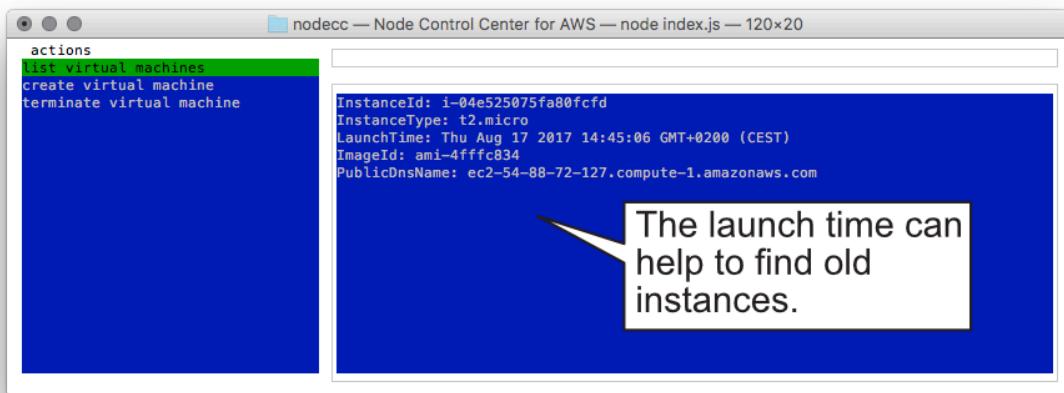


Figure 4.11 nodecc: showing virtual machine details

4.4.4 How nodecc terminates a virtual machine

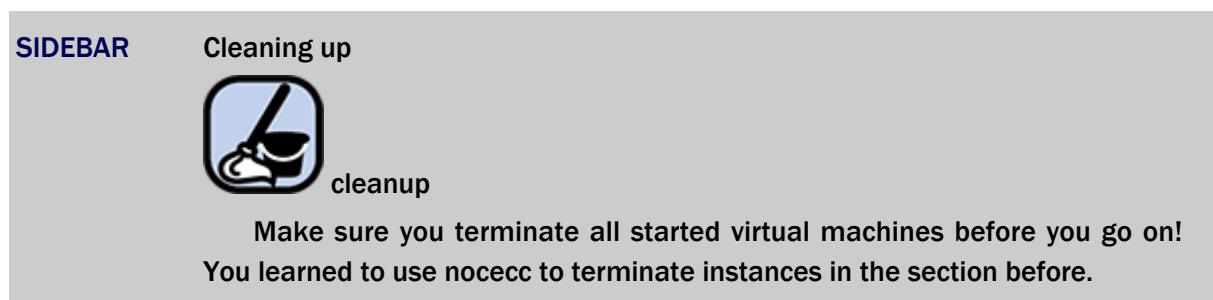
To terminate a virtual machine, you first have to select it. To list the virtual machines, use `lib/listVMs.js` again. After the VM is selected, `lib/terminateVM.js` takes care of termination.

Listing 4.7 Terminating an EC2 instance /lib/terminateVM.js

```
module.exports = (instanceId, cb) => {
  ec2.terminateInstances({
    ①
    InstanceIds: [instanceId] ②
  }, cb);
};
```

- ① API call to list EC2 instances
- ② Pass the selected instance id

That's *nodecc*: a text UI program for controlling temporary EC2 instances. Take some time to think about what you could create by using your favorite language and the AWS SDK. Chances are high that you might come up with a new business idea!



The hard parts about using the SDK are:

- The SDK (or better Node.js) follows an imperative approach. You provide all instructions, step by step, in the right order, to get the job done.
- You have to deal with dependencies (e.g., wait for the instance to be running before connecting to it)
- There is no easy way to update the instances that are running with new settings (e.g., change the instance type).

It's time to enter a new world by leaving the imperative world entering the declarative world.

4.5 Infrastructure as Code

Infrastructure as Code is the idea of using a high-level programming language to control infrastructures. Infrastructure can be any AWS resource, like a network topology, a load balancer, a DNS entry, and so on. In software development, tools like automated tests, code repositories, and build servers increase the quality of software engineering. If your infrastructure is defined as code, then you can apply these types of software development tools to your infrastructure and improve its quality.

WARNING Don't mix up the terms **Infrastructure as Code** and **Infrastructure as a Service (IaaS)**! IaaS means renting virtual machines, storage, and network with a pay-per-use pricing model.

4.5.1 Inventing an infrastructure language: JIML

For the purposes of learning the concepts behind Infrastructure as Code, let's invent a new language to describe infrastructure: JSON Infrastructure Markup Language (JIML). Figure 1.15 shows the infrastructure that will be created in the end.

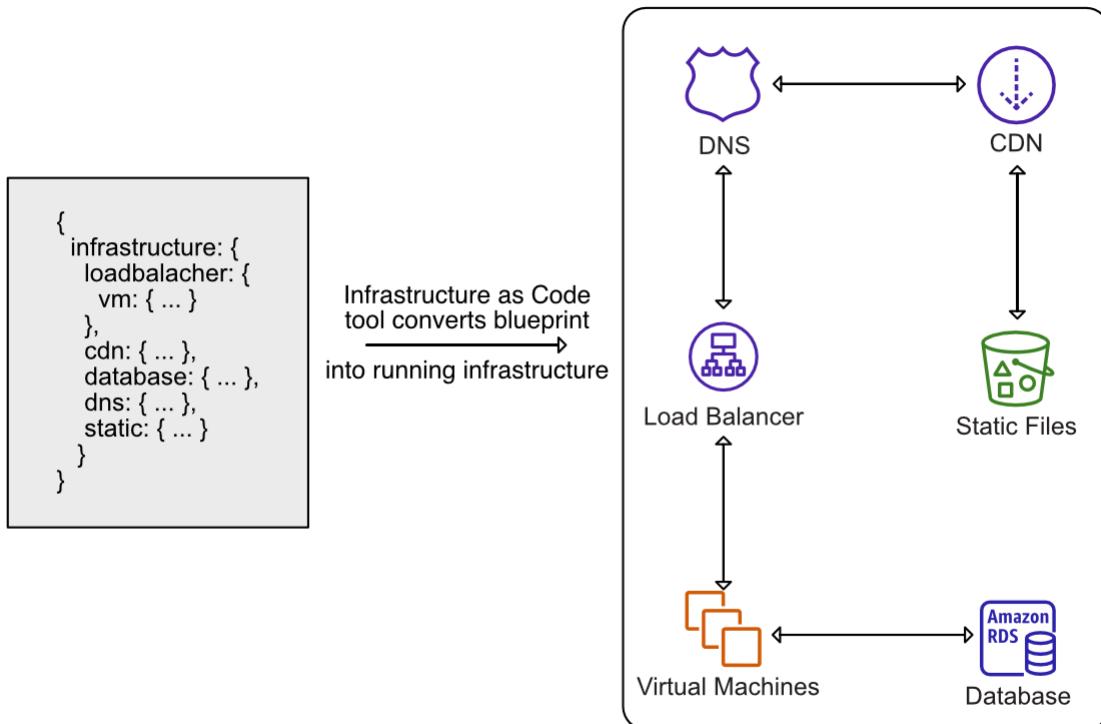


Figure 4.12 From JIML blueprint to infrastructure: infrastructure automation

The infrastructure consists of the following:

- Load balancer (LB)
- Virtual machines (VMs)
- Database (DB)
- DNS entry
- Content delivery network (CDN)
- Storage for static files

To reduce issues with syntax, let's say JIML is based on JSON. The following JIML code creates the infrastructure shown in figure 1.15. The \$ indicates a reference to an ID.

Listing 4.8 Infrastructure description in JIML

```
{
  "region": "us-east-1",
  "resources": [
    {
      "type": "loadbalancer", ①
      "id": "LB",
      "config": {
        "virtualmachines": 2, ②
        "virtualmachine": {
          "cpu": 2,
          "ram": 4,
          "os": "ubuntu"
        }
      },
      "waitFor": "$DB" ④
    },
    {
      "type": "cdn", ⑤
      "id": "CDN",
      "config": {
        "defaultSource": "$LB",
        "sources": [
          {
            "path": "/static/*",
            "source": "$BUCKET"
          }
        ]
      }
    },
    {
      "type": "database",
      "id": "DB", ⑥
      "config": {
        "password": "****",
        "engine": "MySQL"
      }
    },
    {
      "type": "dns", ⑦
      "config": {
        "from": "www.mydomain.com",
        "to": "$CDN"
      }
    },
    {
      "type": "bucket", ⑧
      "id": "BUCKET"
    }
  ]
}
```

- ① A load balancer is needed.
- ② Need two VMs.
- ③ VMs are Ubuntu Linux (4 GB memory, 2 cores).
- ④ LB can only be created if the database is ready.
- ⑤ A CDN is used that caches requests to the LB or fetches static assets (images, CSS files, ...) from a bucket.
- ⑥ Data is stored within a MySQL database.
- ⑦ A DNS entry points to the CDN.
- ⑧ A bucket is used to store static assets (images, CSS files, ...).

How can we turn this JSON into AWS API calls?

1. Parse the JSON input.
2. The JIML tool creates a dependency graph by connecting the resources with their dependencies.
3. The JIML tool traverses the dependency graph from the bottom (leaves) to the top (root) and a linear flow of commands. The commands are expressed in a pseudo language.
4. The commands in pseudo language are translated into AWS API calls by the JIML runtime.

The AWS API calls have to be made based on the resources defined in the blueprint. In particular, it is necessary to send the AWS API calls in the correct order. Let's look at the dependency graph created by the JIML tool, shown in figure 4.13.

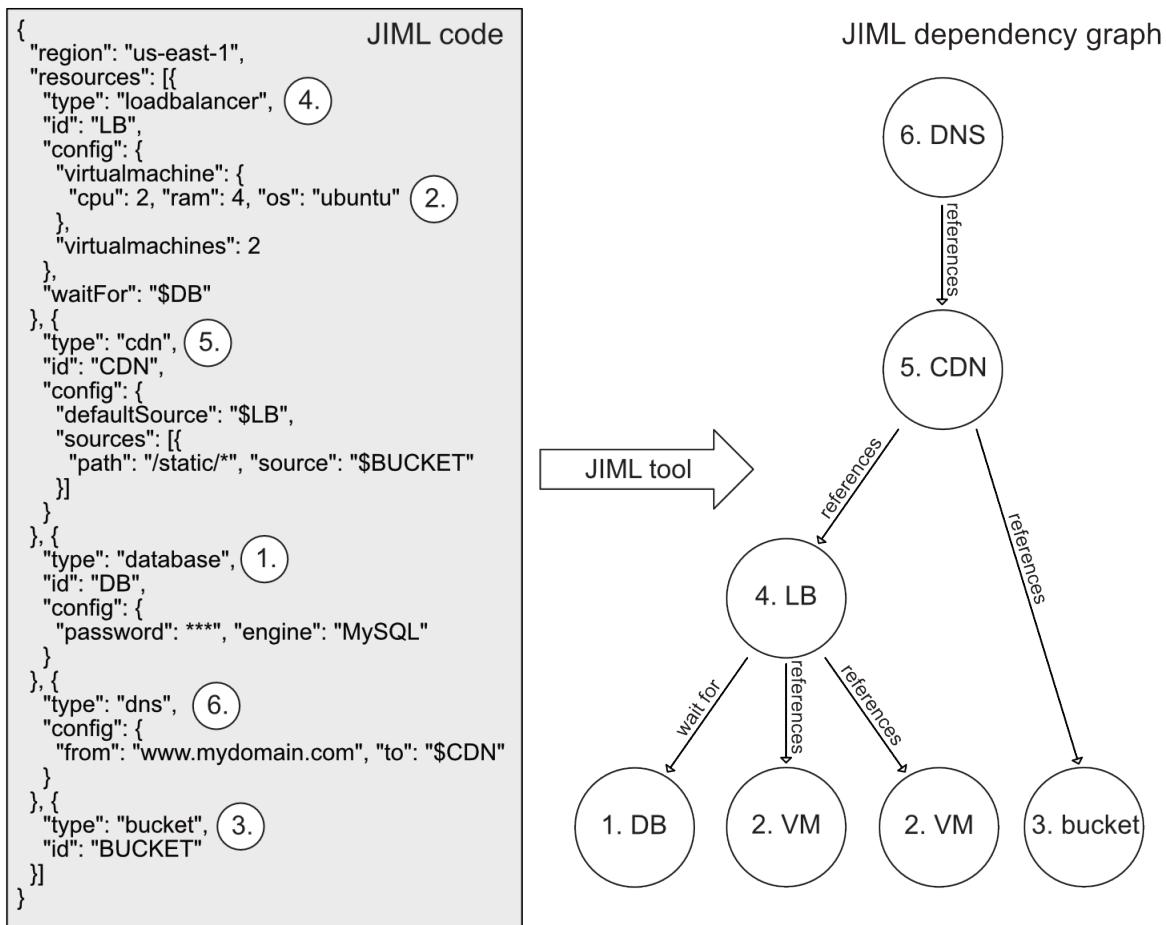


Figure 4.13 The JIML tool figures out the order in which resources need to be created

You traverse the dependency graph in figure 4.13 from bottom to top and from left to right. The nodes at the bottom have no children and therefore no dependencies: DB, VM, and bucket

The LB node depends on the DB node and the two VM nodes. The CDN node depends on the LB and the bucket node. Finally, the DNS node depends on the CDN node.

The JIML tool turns the dependency graph into a linear flow of commands using pseudo

language. The pseudo language represents the steps needed to create all the resources in the correct order. The nodes are easy to create because they have no dependencies, so they’re created first.

Listing 4.9 Linear flow of commands in pseudo language, derived from the dependency graph

```
$DB = database create {"password": "****", "engine": "MySQL"}          ①
$VM1 = virtualmachine create {"cpu": 2, "ram": 4, "os": "ubuntu"}        ②
$VM2 = virtualmachine create {"cpu": 2, "ram": 4, "os": "ubuntu"}        ③
$BUCKET = bucket create {}

await [$DB, $VM1, $VM2]                                              ④
$LB = loadbalancer create {"virtualmachines": [$VM1, $VM2]}           ⑤

await [$LB, $BUCKET]
$CDN = cdn create {...}                                                 ⑥

await $CDN
$DNS = dns create {...}                                                 ⑦

await $DNS
```

- ① Create the database.
- ② Create the virtual machine.
- ③ Create the bucket.
- ④ Wait for the dependencies.
- ⑤ Create the load balancer.
- ⑥ Create the CDN.
- ⑦ Create the DNS entry.

We’ll skip the last step—translating the commands from the pseudo language into AWS API calls. You’ve already learned everything you need to know about infrastructure as code: it’s all about dependencies. Let’s apply that newly learned ideas in practice.

4.6 Using AWS CloudFormation to start a virtual machine

In the previous section, we talked about JIML to introduce the concept of infrastructure as code. Luckily, AWS already offers a tool that does much better than JIML: *AWS CloudFormation*. CloudFormation is based on templates, which up to now we’ve called blueprints.

NOTE	When using CloudFormation, what we have been referring to as a blueprint so far is actually referred to as a CloudFormation template.
-------------	--

A *template* is a description of your infrastructure, written in JSON or YAML, that can be interpreted by CloudFormation. The idea of describing something rather than listing the

necessary actions is called a *declarative approach*. Declarative means you tell CloudFormation how your infrastructure should look. You aren't telling CloudFormation what actions are needed to create that infrastructure, and you don't specify the sequence in which the actions need to be executed.

The benefits of CloudFormation are as follows:

- *It's a consistent way to describe infrastructure on AWS.* If you use scripts to create your infrastructure, everyone will solve the same problem differently. This is a hurdle for new developers and operators trying to understand what the code is doing. CloudFormation templates are a clear language for defining infrastructure.
- *It handles dependencies.* Ever tried to register a web server with a load balancer that wasn't yet available? When you first start trying to automate infrastructure creation, you'll miss a lot of dependencies. Trust us: never try to set up complex infrastructure using scripts. You'll end up in dependency hell!
- *It's reproducible.* Is your test environment an exact copy of your production environment? Using CloudFormation, you can create two identical infrastructures. It is also possible to apply changes to both the test and production environment.
- *It's customizable.* You can insert custom parameters into CloudFormation to customize your templates as you wish.
- *It's testable.* If you can create your architecture from a template, it's testable. Just start a new infrastructure, run your tests, and shut it down again.
- *It's updatable.* CloudFormation supports updates to your infrastructure. It will figure out the parts of the template that have changed and apply those changes as smoothly as possible to your infrastructure.
- *It minimizes human failure.* CloudFormation doesn't get tired—even at 3 a.m.
- *It's the documentation for your infrastructure.* A CloudFormation template is a JSON or YAML document. You can treat it as code and use a version control system like Git to keep track of the changes.
- *It's free.* Using CloudFormation comes at no additional charge. If you subscribe to an AWS support plan, you also get support for CloudFormation.

We think CloudFormation is the most powerful tool available for managing infrastructure on AWS.

4.6.1 Anatomy of a CloudFormation template

A basic CloudFormation template is structured into five parts:

1. *Format version*—The latest template format version is 2010-09-09, and this is currently the only valid value. Specify this version; the default is to use the latest version, which will cause problems if new versions are introduced in the future.
2. *Description*—What is this template about?
3. *Parameters*—Parameters are used to customize a template with values: for example, domain name, customer ID, and database password.
4. *Resources*—A resource is the smallest block you can describe. Examples are a virtual machine, a load balancer, or an Elastic IP address.

5. *Outputs*—An output is comparable to a parameter, but the other way around. An output returns something from your template, such as the public name of an EC2 instance.

A basic template looks like the following listing.

Listing 4.10 CloudFormation template structure

```
--> ①
AWSTemplateFormatVersion: '2010-09-09'          ②
Description: 'CloudFormation template structure' ③
Parameters:
# [...]
④
Resources
# [...]
⑤
Outputs:
# [...]
⑥
```

- ① Start of a document
- ② The only valid version
- ③ What is this template about?
- ④ Defines parameters
- ⑤ Defines resources
- ⑥ Defines outputs

Let's take a closer look at parameters, resources, and outputs.

FORMAT VERSION AND DESCRIPTION

The only valid `AWSTemplateFormatVersion` value at the moment is `2010-09-09`. Always specify the format version. If you don't, CloudFormation will use whatever version is the latest one. As mentioned earlier, this means that if a new format version is introduced in the future, you'll end up using the wrong format and get into serious trouble.

`Description` isn't mandatory, but we encourage you to take some time to document what the template is about. A meaningful description will help you in the future to remember what the template is for. It will also help your coworkers.

PARAMETERS

A parameter has at least a name and a type. We encourage you to add a description as well.

Listing 4.11 CloudFormation parameter structure

```
Parameters:
Demo:           ①
Type: Number    ②
Description: 'This parameter is for demonstration' ③
```

- ① You can choose the name of the parameter.
- ② This parameter represents a number.
- ③ Description of the parameter

Valid types are listed in table [4.1](#).

Table 4.1 CloudFormation parameter types

Type	Description
String CommaDelimitedList	A string or a list of strings separated by commas
Number List<Number>	An integer or float, or a list of integers or floats
AWS::EC2::AvailabilityZone::Name List<AWS::EC2::AvailabilityZone::Name>	An Availability Zone, such as us-west-2a, or a list of Availability Zones
AWS::EC2::Image::Id List<AWS::EC2::Image::Id>	An AMI ID or a list of AMIs
AWS::EC2::Instance::Id List<AWS::EC2::Instance::Id>	An EC2 instance ID or a list of EC2 instance IDs
AWS::EC2::KeyPair::KeyName	An Amazon EC2 key-pair name
AWS::EC2::SecurityGroup::Id List<AWS::EC2::SecurityGroup::Id>	A security group ID or a list of security group IDs
AWS::EC2::Subnet::Id List<AWS::EC2::Subnet::Id>	A subnet ID or a list of subnet IDs
AWS::EC2::Volume::Id List<AWS::EC2::Volume::Id>	An EBS volume ID (network attached storage) or a list of EBS volume IDs
AWS::EC2::VPC::Id List<AWS::EC2::VPC::Id>	A VPC ID (virtual private cloud) or a list of VPC IDs
AWS::Route53::HostedZone::Id List<AWS::Route53::HostedZone::Id>	A DNS zone ID or a list of DNS zone IDs

In addition to using the Type and Description properties, you can enhance a parameter with the properties listed in table [4.2](#).

Table 4.2 CloudFormation parameter properties

Property	Description	Example
Default	A default value for the parameter	Default: 'm5.large'
NoEcho	Hides the parameter value in all graphical tools (useful for secrets)	NoEcho: true
AllowedValues	Specifies possible values for the parameter	AllowedValues: [1, 2, 3]
AllowedPattern	More generic than AllowedValues because it uses a regular expression	AllowedPattern: '[a-zA-Z0-9]*' allows only a-z, A-Z, and 0-9 with any length
MinLength, MaxLength	Defines how long a parameter can be	MinLength: 12
MinValue, MaxValue	Used in combination with the Number type to define lower and upper bounds	MaxValue: 10
ConstraintDescription	A string that explains the constraint when the constraint is violated.	ConstraintDescription: 'Maximum value is 10.'

A parameter section of a CloudFormation template could look like this:

```
Parameters:
  KeyName:
    Description: 'Key Pair name'
    Type: 'AWS::EC2::KeyPair::KeyName'          ①
  NumberOfVirtualMachines:
    Description: 'How many virtual machine do you like?'
    Type: Number
    Default: 1                                ②
    MinValue: 1
    MaxValue: 5                                ③
  WordPressVersion:
    Description: 'Which version of WordPress do you want?'
    Type: String
    AllowedValues: ['4.1.1', '4.0.1']           ④
```

- ① Only key pair names are allowed.
- ② The default is one virtual machine.
- ③ Prevent massive costs with an upper bound.
- ④ Restricted to certain versions

Now you should have a better feel for parameters. If you want to know everything about them, see the documentation at docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/parameters-section-structure.html or follow along in the book and learn by doing.

RESOURCES

A resource has at least a name, a type, and some properties.

Listing 4.12 CloudFormation resources structure

```
Resources:
  VM:
    Type: 'AWS::EC2::Instance'          ②
    Properties:
      # [...]                         ③
```

- ① Name or logical ID of the resource that you can choose
- ② The resource of type AWS::EC2::Instances defines a virtual machine
- ③ Properties needed for the type of resource

When defining resources, you need to know about the type and that type's properties. In this book, you'll get to know a lot of resource types and their respective properties. An example of a single EC2 instance follows.

Listing 4.13 CloudFormation EC2 instance resource

```
Resources:
  VM:          ①
    Type: 'AWS::EC2::Instance'      ②
    Properties:
      ImageId: 'ami-6057e21a'      ③
      InstanceType: 't2.micro'      ④
      SecurityGroupIds:
        - 'sg-123456'              ⑤
      SubnetId: 'subnet-123456'      ⑥
```

- ① Name or logical ID of the resource that you can choose
- ② The resource of type AWS::EC2::Intsances defines a virtual machine
- ③ The AMI defines the operating system of the vm
- ④ The instace type defines the number of vCPUs, memory, and more
- ⑤ You'll learn about this in chapter 5
- ⑥ You'll learn about this in chapter 5

Now you've described the virtual machine, but how can you output its public name?

OUTPUTS

A CloudFormation template's output includes at least a name (like parameters and resources) and a value, but we encourage you to add a description as well. You can use outputs to pass data from within your template to the outside.

Listing 4.14 CloudFormation outputs structure

```
Outputs:
  NameOfOutput:          ①
    Value: '1'            ②
    Description: 'This output is always 1'
```

- ① Name of the output that you can choose
- ② Value of the output

Static outputs like this one aren't very useful. You'll mostly use values that reference the name of a resource or an attribute of a resource, like its public name. If you see `!Ref NameOfSomething`, think of it as a placeholder for what is referenced by the name. A `!GetAtt 'NameOfSomething.AttributeOfSomething'` is similar to a ref but you select a specific attribute of the referenced resource.

Listing 4.15 CloudFormation outputs example

```
Outputs:
  ID:
    Value: !Ref Server      ①
    Description: 'ID of the EC2 instance'
  PublicName:
    Value: !GetAtt 'Server.PublicDnsName'      ②
    Description: 'Public name of the EC2 instance'
```

- ① References the EC2 instance
- ② Get the attribute PublicDnsName of the EC2 instance.

You'll get to know the most important attributes of `!GetAtt` while reading the book.

Now that we've taken a brief look at the core parts of a CloudFormation template, it's time to make one of your own.

4.6.2 Creating your first template

How do you create a CloudFormation template? Different options are available:

- Use a text editor or IDE to write a template from scratch.
- Start with a template from a public repository that offers a default implementation and adapt it to your needs.
- Use a template provided by your vendor.

AWS and their partners offer CloudFormation templates for deploying popular solutions: AWS Quick Starts at aws.amazon.com/quickstart/. Furthermore, we have open sourced the templates we are using in our day-to-day work on GitHub: github.com/widdix/aws-cf-templates.

Suppose you've been asked to provide a VM for a developer team. After a few months, the team realizes the VM needs more CPU power, because the usage pattern has changed. You can handle that request with the CLI and the SDK, but before the instance type can be changed, you must stop the EC2 instance. The process will be as follows:

1. Stop the instance.
2. Wait for the instance to stop.
3. Change the instance type.
4. Start the instance.
5. Wait for the instance to start.

A declarative approach like that used by CloudFormation is simpler: just change the `InstanceType` property and update the template. `InstanceType` can be passed to the template via a parameter. That's it! You can begin creating the template.

Listing 4.16 Template to create an EC2 instance with CloudFormation

```
---
AWSTemplateFormatVersion: '2010-09-09'
Description: 'AWS in Action: chapter 4'
Parameters:
  VPC:    ①
    Type: 'AWS::EC2::VPC::Id'
  Subnet:  ②
    Type: 'AWS::EC2::Subnet::Id'
  InstanceType: ③
    Description: 'Select one of the possible instance types'
    Type: String
    Default: 't2.micro'
    AllowedValues: ['t2.micro', 't2.small', 't2.medium']
Resources:
  SecurityGroup: ④
    Type: 'AWS::EC2::SecurityGroup'
  Properties:
    # [...]
  VM:      ⑤
    Type: 'AWS::EC2::Instance'
    Properties:
      ImageId: 'ami-06lac2e015473fbe2'
      InstanceType: !Ref InstanceType
      IamInstanceProfile: 'ec2-ssm-core'
      SecurityGroupIds: [!Ref SecurityGroup] ⑥
      SubnetId: !Ref Subnet
Outputs:
  InstanceId: ⑦
    Value: !Ref VM
    Description: 'Instance id (connect via Session Manager)'
```

- ① You'll learn about this in chapter 5.
- ② You'll learn about this in chapter 5.
- ③ The user defines the instance type.
- ④ You'll learn about this in chapter 5.
- ⑤ Defines a minimal EC2 instance
- ⑥ By referencing the security group an implicit dependency is declared
- ⑦ Returns the id of the EC2 instance

You can find the full code for the template at `/chapter04/virtualmachine.yaml` in the book's code folder. Please don't worry about VPC, subnets, and security groups at the moment; you'll get to know them in chapter 5.

SIDEBAR
Where is the template located?

You can find the template on GitHub. You can download a snapshot of the repository at github.com/AWSinAction/code3/archive/main.zip. The file we're talking about is located at `chapter04/virtualmachine.yaml`. On S3, the same file is located at s3.amazonaws.com/awsinaction-code3/chapter04/virtualmachine.yaml.

If you create an infrastructure from a template, CloudFormation calls it a *stack*. You can think of *template* versus *stack* much like *class* versus *object*. The template exists only once, whereas many stacks can be created from the same template.

1. Open the AWS Management Console at console.aws.amazon.com.
2. Click on *Services* and search for *CloudFormation*.
3. Open the CloudFormation service.
4. Select *Stacks* at left.

Figure 4.14 shows the empty list of CloudFormation stacks.

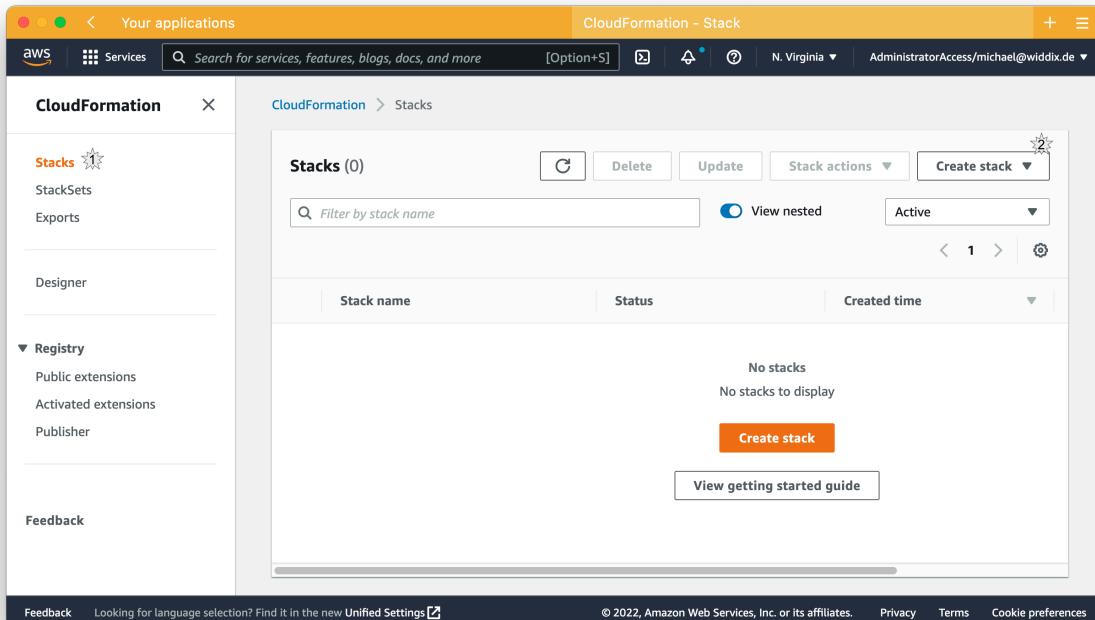


Figure 4.14 Overview of CloudFormation stacks

The following steps will guide you through creating your stack:

1. Click *Create Stack* and select *With new resources (standard)* to start a four-step wizard.
2. Select *Template is ready*.
3. Select *Amazon S3 URL* as your template source and enter the value s3.amazonaws.com/awsinaction-code3/chapter04/virtualmachine.yaml as shown in figure 4.15.
4. Continue by clicking *Next*.

Create stack

Prerequisite - Prepare template

Prepare template

Every stack is based on a template. A template is a JSON or YAML file that contains configuration information about the AWS resources you want to include in the stack.

Template is ready

Use a sample template

Create template in Designer

Specify template

A template is a JSON or YAML file that describes your stack's resources and properties.

Template source

Selecting a template generates an Amazon S3 URL where it will be stored.

Amazon S3 URL

Upload a template file

Amazon S3 URL

`https://s3.amazonaws.com/awsinaction-code3/chapter04/virtualmachine.yaml`

Specify the URL of the CloudFormation template.

Amazon S3 template URL

S3 URL: `https://s3.amazonaws.com/awsinaction-code3/chapter04/virtualmachine.yaml`

[View in Designer](#)

[Cancel](#)

[Next](#)

Figure 4.15 Creating a CloudFormation stack: selecting a template (step 1 of 4)

In the second step, you define the stack name and parameters. Give the stack a name like `myvm` and fill out the parameter values:

1. `InstanceType`—Select `t2.micro`.
2. `Subnet`—Select the first value in the drop-down list. You'll learn about subnets later.
3. `VPC`—Select the first value in the drop-down list. You'll learn about VPCs later.

Figure 4.16 shows the second step. Click *Next* after you've chosen a value for the stack name and every parameter, to proceed with the next step.

Specify stack details

Stack name

Stack name
myvm

Stack name can include letters (A-Z and a-z), numbers (0-9), and dashes (-).

Define a name for your CloudFormation stack.

Parameters

Parameters are defined in your template and allow you to input custom values when you create or update a stack.

InstanceType
Select one of the possible instance types'
t2.micro

Select instance type t2.micro.

Subnet
Just select one of the available subnets
subnet-84c65488

Select the first subnet from the drop-down list.

VPC
Just select the one and only default VPC
vpc-a17392c7

Select the only VPC from the drop-down list.

Cancel Previous Next

Figure 4.16 Creating a CloudFormation stack: defining parameters (step 2 of 4)

In the third step, you can define optional tags for the stack and advanced configuration. You can skip this step at this point in the book, because you will not use any advances features for now. All resources created by the stack will be tagged by CloudFormation by default. Click *Next* to go to the last step.

Step four displays a summary of the stack. At the bottom of the page, you are asked to *acknowledge the creation of IAM resources* as figure 4.17 shows. You can safely allow CloudFormation to create IAM resources for now. You will learn more about them in chapter 5.

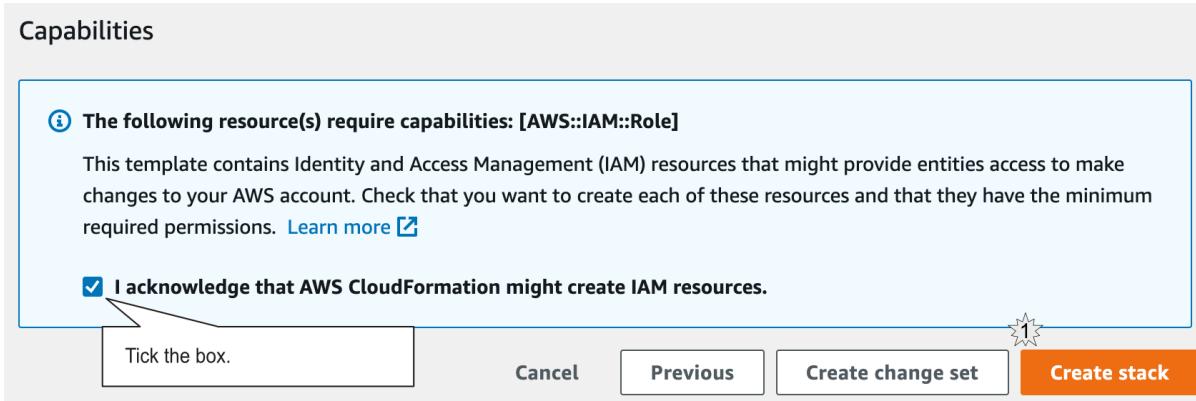


Figure 4.17 Creating a CloudFormation stack: summary (step 4 of 4)

Click *Create stack*. CloudFormation now starts to create the stack. If the process is successful, you'll see the screen shown in figure 4.18. As long as Status is CREATE_IN_PROGRESS, you need to be patient and hit the reload button from time to time. When Status is CREATE_COMPLETE, click the *Outputs* tab to see the id of the EC2 instance. Double-check the instance type in the EC2 Management Console.

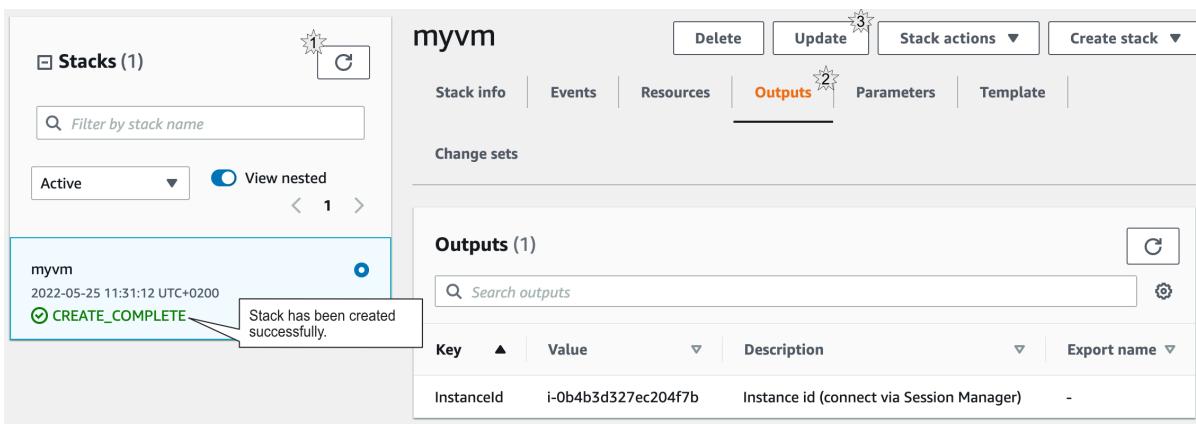


Figure 4.18 The CloudFormation stack has been created

Your stack is now created. But that's not the end of the story. CloudFormation supports updating and deleting stacks as well.

4.6.3 Updating infrastructure using CloudFormation

It's time to test modifying the instance type. Click the *Update Stack* button. The wizard that starts is similar to the one you used during stack creation. Figure 4.19 shows the first step of the wizard. Select *Use Current Template* and proceed with the next step by clicking the *Next* button.

Update stack

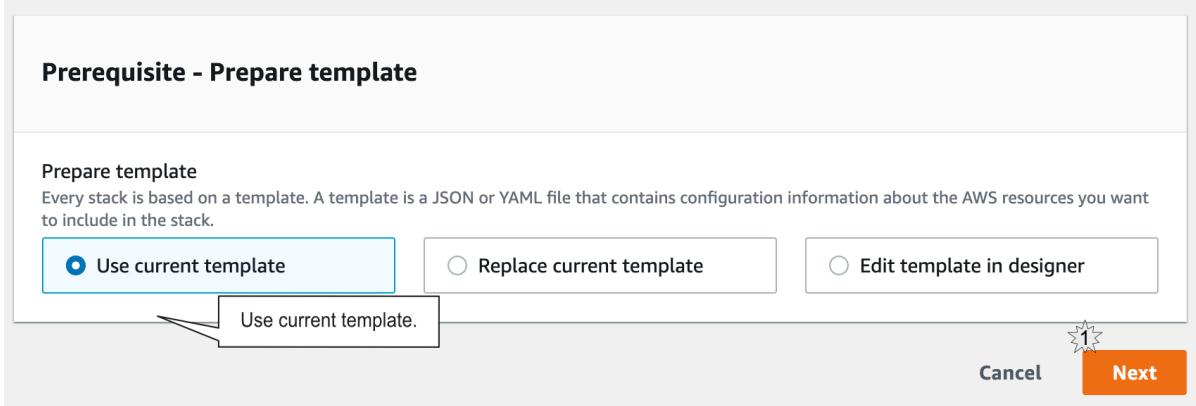


Figure 4.19 Updating the CloudFormation stack: summary (step 1 of 4)

In step 2, you need to change the `InstanceType` parameter value: choose `t2.small` to double or `t2.medium` to quadruple the computing power of your EC2 instance.

WARNING Starting a virtual machine with instance type `t2.small` or `t2.medium` will incur charges. See aws.amazon.com/ec2/pricing/ to find out the current hourly price.

Step 3 is about sophisticated options during the update of the stack. You don't need any of these features now, so skip the step by clicking *Next*. Step 4 is a summary; acknowledge the creation of IAM resources and click *Update stack*. The stack now has Status `UPDATE_IN_PROGRESS`. If you are quickly jumping to the EC2 Management Console you can see the instance to be stopped and started again with the new instance type. After a few minutes, Status should change to `UPDATE_COMPLETE`.

SIDEBAR

Alternatives to CloudFormation

If you don't want to write plain JSON or YAML to create templates for your infrastructure, there are a few alternatives to CloudFormation.

The AWS Cloud Development Kit (CDK) allows you to use a general purpose programming language to define your infrastructure. Under the hood, the CDK generates CloudFormation templates.

There are also tools that allow you to use infrastructure as code without needing CloudFormation. [Terraform](#) is the most popular option.

When you changed the parameter, CloudFormation figured out what needed to be done to achieve the end result. That's the power of a declarative approach: you say what the end result should look like, not how the end result should be achieved.

SIDE BAR**Cleaning up**

cleanup

Delete the stack by selecting it and clicking the Delete button.

4.7 Summary

- Use the CLI, one of the SDKs, or CloudFormation to automate your infrastructure on AWS.
- *Infrastructure as code* describes the approach of programming the creation and modification of your infrastructure, including virtual machines, networking, storage, and more.
- You can use the CLI to automate complex processes in AWS with scripts (Bash and PowerShell).
- You can use SDKs for nine programming languages and platforms to embed AWS into your applications and create applications like nodecc.
- CloudFormation uses a declarative approach in JSON or YAML: you only define the end state of your infrastructure, and CloudFormation figures out how this state can be achieved. The major parts of a CloudFormation template are parameters, resources, and outputs.

5

Securing your system: IAM, security groups, and VPC

This chapter covers

- Who is responsible for security?
- Keeping your software up to date
- Controlling access to your AWS account with users and roles
- Keeping your traffic under control with security groups
- Using CloudFormation to create a private network

If security is a wall, you'll need a lot of bricks to build that wall as shown in figure 5.1. This chapter focuses on the four most important bricks to secure your systems on AWS:

1. *Installing software updates*—New security vulnerabilities are found in software every day. Software vendors release updates to fix those vulnerabilities, and it's your job to install those updates as quickly as possible after they're released on your systems. Otherwise, your systems will be an easy victim for hackers.
2. *Restricting access to your AWS account*—This becomes even more important if you aren't the only one accessing your AWS account. For example, when coworkers and automated processes need access to your AWS account as well. A buggy script could easily terminate all your EC2 instances instead of only the one you intended. Granting only the permissions needed is key to securing your AWS resources from accidental or intended disastrous actions.
3. *Controlling network traffic to and from your EC2 instances*—You only want ports to be accessible if they must be. If you run a web server, the only ports you need to open to the outside world are port 80 for HTTP traffic and 443 for HTTPS traffic. Do not open any other ports for external access.
4. *Creating a private network in AWS*—Control network traffic by defining subnets and routing tables. Doing so allows you to specify private networks, that are not reachable from the outside.

One important brick is missing: securing your applications. We do not cover application security

in our book. When buying or developing applications, you should follow security standards. For example, you need to check user input and allow only the necessary characters, don't save passwords in plain text, and use TLS/SSL to encrypt traffic between your virtual machines and your users.

This is going to be a long chapter, because security is such an important topic there's a lot to cover. But don't worry, we'll take it step by step.

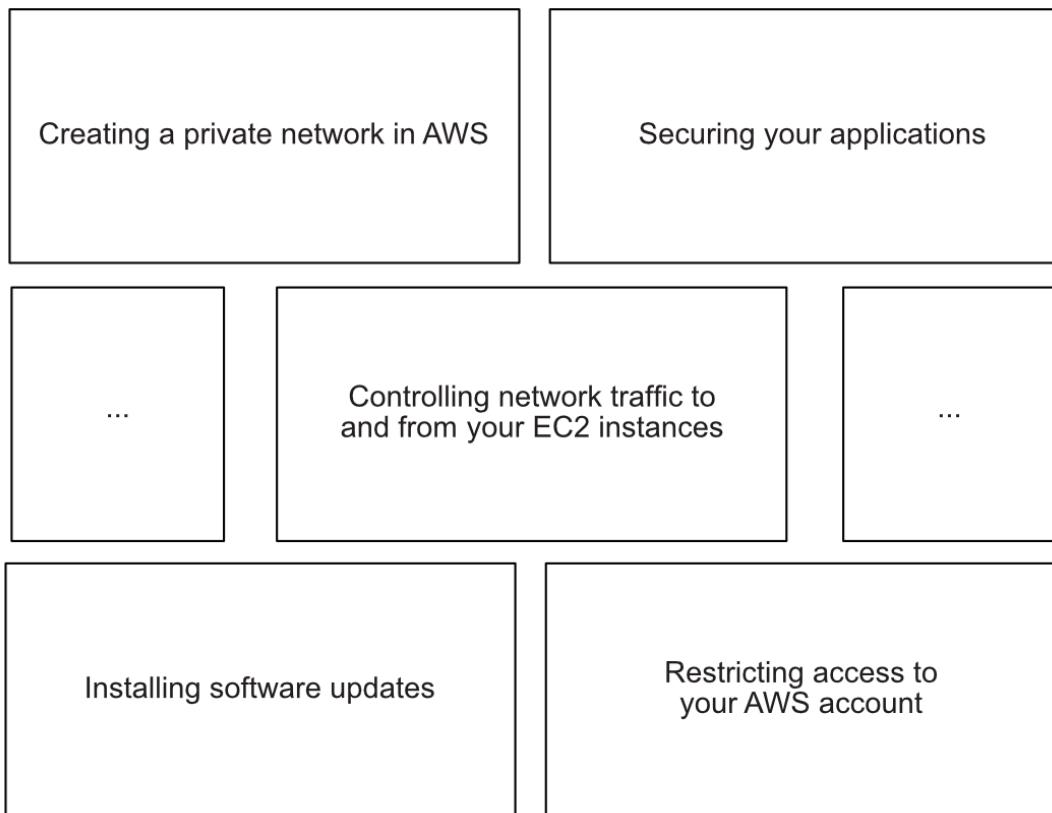


Figure 5.1 To achieve security of your cloud infrastructure and application, all security building blocks have to be in place.

SIDE BAR

Not all examples are covered by Free Tier

The examples in this chapter are not all covered by the Free Tier. A special warning message appears when an example incurs costs. As for the other examples, as long as you don't run them longer than a few days, you won't pay anything for them. Keep in mind that this applies only if you created a fresh AWS account for this book and nothing else is going on in your AWS account. Try to complete the chapter within a few days; you'll clean up your account at the end.

SIDE BAR**Chapter requirements**

To fully understand this chapter, you should be familiar with the following networking concepts:

- Subnet
- Route tables
- Access control lists (ACLs)
- Gateway
- Firewall
- Port
- Access management
- Basics of the Internet Protocol (IP), including IP addresses

Before we look at the four bricks, let's talk about how responsibility is divided between you and AWS.

5.1 Who's responsible for security?

The cloud is a shared-responsibility environment, meaning responsibility is shared between you and AWS. AWS is responsible for the following :

- Protecting the network through automated monitoring systems and robust internet access, to prevent Distributed Denial of Service (DDoS) attacks.
- Performing background checks on employees who have access to sensitive areas.
- Decommissioning storage devices by physically destroying them after end of life.
- Ensuring the physical and environmental security of data centers, including fire protection and security staff.

The security standards are reviewed by third parties; you can find an up-to-date overview at aws.amazon.com/compliance/.

What are your responsibilities?

- Configuring access management that restricts access to AWS resources like S3 and EC2 to a minimum, using AWS IAM.
- Encrypting network traffic to prevent attackers from reading or manipulating data (for example, using HTTPS).
- Configuring a firewall for your virtual network that controls incoming and outgoing traffic with security groups and NACLs.
- Encrypting data at rest. For example, enable data encryption for your database or other storage systems.
- Managing patches for the OS and additional software on virtual machines.

Security involves an interaction between AWS and you, the customer. If you play by the rules, you can achieve high security standards in the cloud. Want to dive into more details, check out aws.amazon.com/compliance/shared-responsibility-model/.

5.2 Keeping the operating system up to date

Not a week goes by without the release of an important update to fix security vulnerabilities in some piece of software or another. Sometimes the kernel is affected, or libraries like OpenSSL. Other times it's affecting environments like Java, Apache, and PHP; or an application like WordPress. If a security update is released, you must install it quickly, because the exploit may have already been released, or because unscrupulous people could look at the source code to reconstruct the vulnerability. You should have a working plan for how to apply updates to all running virtual machines as quickly as possible.

Amazon Linux 2 installs critical or important security updates automatically on startup while `cloud-init` is running. We highly recommend to install all the other updates as well. The following options are available:

1. *Install all updates at the end of the boot process* by including `yum -y update` in your user-data script. `yum` is the package manager used by on Amazon Linux 2.
2. *Install security updates at the end of the boot process only* by including `yum -y --security update` in your user-data script.
3. Use the *AWS Systems Manager Patch Manager* to install updates based on a patch baseline.

The first two options can be easily included in the user data of your EC2 instance. You can find the code in `/chapter05/ec2-yum-update.yaml` in the book's code folder. You install all updates as follows:

```
Instance:
  Type: 'AWS::EC2::Instance'
  Properties:
    # [...]
    UserData: !Base64 |
      #!/bin/bash -ex
      yum -y update
```

①

- ① Installs all updates

To install only security updates, do the following:

```
Instance:
  Type: 'AWS::EC2::Instance'
  Properties:
    # [...]
    UserData: !Base64 |
      #!/bin/bash -ex
      yum -y --security update
```

①

- ① Installs only security updates

The following challenges are still waiting for a solution:

- The problem with installing all updates is that your system might still be vulnerable. Some updates require a reboot (most notably kernel updates)!
- Installing updates on startup is not enough. Updates need to be installed continuously.

Before reinventing the wheel, it is a good strategy to research if AWS provides the building blocks needed to get the job done. Luckily, AWS Systems Manager (SSM) Patch Manager is a good choice to make patching more robust and stable.

The AWS Systems Manager provides a toolbox including a core set of features that are bundled into capabilities. Patch Manager is one such capability. The following core SSM features are bundled together to provide Patch Manager as figure [5.2](#) shows.

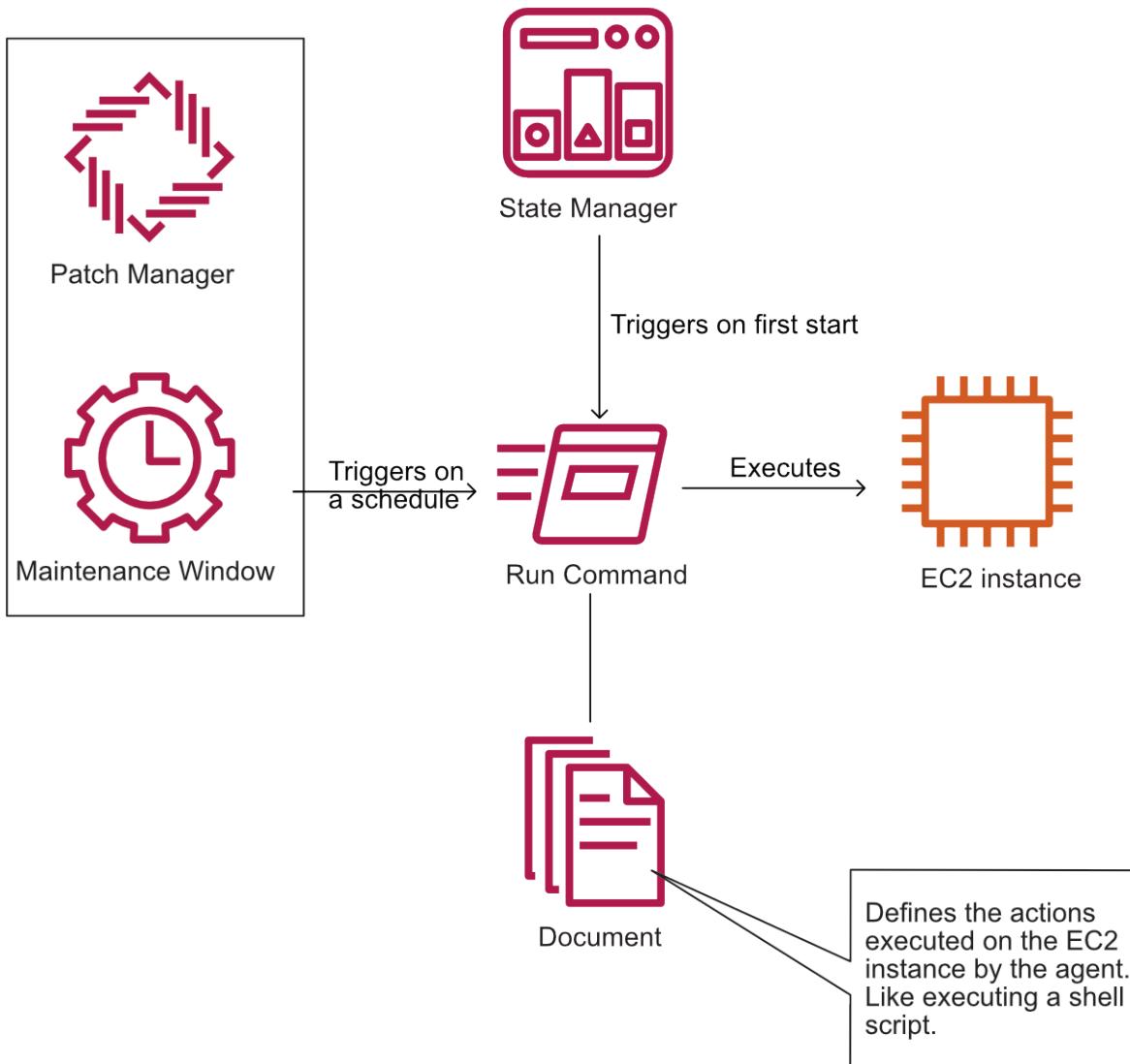


Figure 5.2 SSM features required for patch manager capability

- **Agent:** Pre-installed and auto-started on Amazon Linux 2 (also powers Session Manager).
- **Document:** Think of a document as a script on steroids. We use a pre-build document named `AWS-RunPatchBaseline` to install patches.
- **Run Command:** Executes a document on an EC2 instance.
- **Association:** Sends commands (via Run Command) to EC2 instances on a schedule or during startup (bundled into the capability named State Manager).
- **Maintenance Window:** Sends commands (via Run Command) to EC2 instances on a schedule during a time window.
- **Patch baseline:** Set of rules to approve patches for installation based on classification and severity. Luckily, AWS provides predefined patch baselines for various operating systems including Amazon Linux 2. The predefined patch baseline for Amazon Linux 2 approves all security patches that have a severity level of critical or important and all bug fixes. A seven day waiting period exists after the release of a patch before approval.

The following CloudFormation snippets defines a maintenance window to patch on a schedule as well as an association to patch on startup.

```

MaintenanceWindow:
  Type: 'AWS::SSM::MaintenanceWindow'
  Properties:
    AllowUnassociatedTargets: false
    Duration: 2
    Cutoff: 1
    Name: !Ref 'AWS::StackName'
    Schedule: 'cron(0 5 ? * SUN *)' ③
    ScheduleTimezone: UTC
  MaintenanceWindowTarget: ④
    Type: 'AWS::SSM::MaintenanceWindowTarget'
    Properties:
      ResourceType: INSTANCE
      Targets:
        - Key: InstanceIds
          Values:
            - !Ref Instance
      WindowId: !Ref MaintenanceWindow
  MaintenanceWindowTask:
    Type: 'AWS::SSM::MaintenanceWindowTask'
    Properties:
      MaxConcurrency: '1'
      MaxErrors: '1'
      Priority: 0
      Targets:
        - Key: WindowTargetIds
          Values:
            - !Ref MaintenanceWindowTarget
    TaskArn: 'AWS-RunPatchBaseline' ⑤
  TaskInvocationParameters:
    MaintenanceWindowRunCommandParameters:
      Parameters: ⑥
        Operation:
          - Install
      TaskType: 'RUN_COMMAND'
      WindowId: !Ref MaintenanceWindow
  AssociationRunPatchBaselineInstall:
    Type: 'AWS::SSM::Association' ⑦
    Properties:
      Name: 'AWS-RunPatchBaseline'
      Parameters:
        Operation:
          - Install
      Targets:
        - Key: InstanceIds
          Values:
            - !Ref Instance

```

- ① The maintenance window is 2 hours long. You can patch more than one EC2 instance if you wish!
- ② The last hour is reserved for commands to finish (all commands are started in the first hour).
- ③ The maintenance window is scheduled every Sunday morning at 5am UTC time. [Learn more about the syntax](#).
- ④ Assigns one EC2 instance to the maintenance window. You can also assign EC2 instances based on tags.
- ⑤ The AWS-RunPatchBaseline document is executed.

- ⑥ The document supports parameters. Operation can be set to `Install` or `Scan`. By default, a reboot happens if required by any patch.
- ⑦ The association ensures that patches are installed on startup. The same document with the same parameters are used.

There is one prerequisite missing: The EC2 instance needs read access to a set of [S3 buckets for Patch Manager to work](#).

```
InstanceRole:
  Type: 'AWS::IAM::Role'
  Properties:
    #[...]
    Policies:
      - PolicyName: PatchManager
        PolicyDocument:
          Version: '2012-10-17'
          Statement:
            - Effect: Allow
              Action: 's3:GetObject'
              Resource:
                - !Sub 'arn:aws:s3:::patch-baseline-snapshot-${AWS::Region}/*'
                - !Sub 'arn:aws:s3:::aws-ssm-${AWS::Region}/*'
```

Patch Manager can also visualise the patches that are waiting for installation. To gather the data, another association is needed.

```
AssociationRunPatchBaselineScan:
  Type: 'AWS::SSM::Association'
  Properties:
    ApplyOnlyAtCronInterval: true      ①
    Name: 'AWS-RunPatchBaseline'       ②
    Parameters:
      Operation:
        - Scan      ③
    ScheduleExpression: 'cron(0 0/1 * * ? *)'  ④
    Targets:
      - Key: InstanceIds
        Values:
          - !Ref Instance
```

- ① Do not run on startup. Unfortunately, the document `AWS-RunPatchBaseline` crashes when running more than once at the same time. Avoids a conflict with the association defined in `AssociationRunPatchBaselineInstall`.
- ② Uses the same document `AWS-RunPatchBaseline`.
- ③ But this time, `Operation` is set to `Scan`.
- ④ Run every hour.

It's time for a demo. Create the CloudFormation stack with the template located at s3.amazonaws.com/awsinaction-code3/chapter05/ec2-os-update.yaml by clicking on the [CloudFormation Quick-Create Link](#). Pick the default VPC and subnet. Wait for the stack creation to finish.

Visit the [AWS Systems Manager management console](#). Open **Patch Manager** in the navigation bar, and you see a nice dashboard as shown in figure [5.3](#).

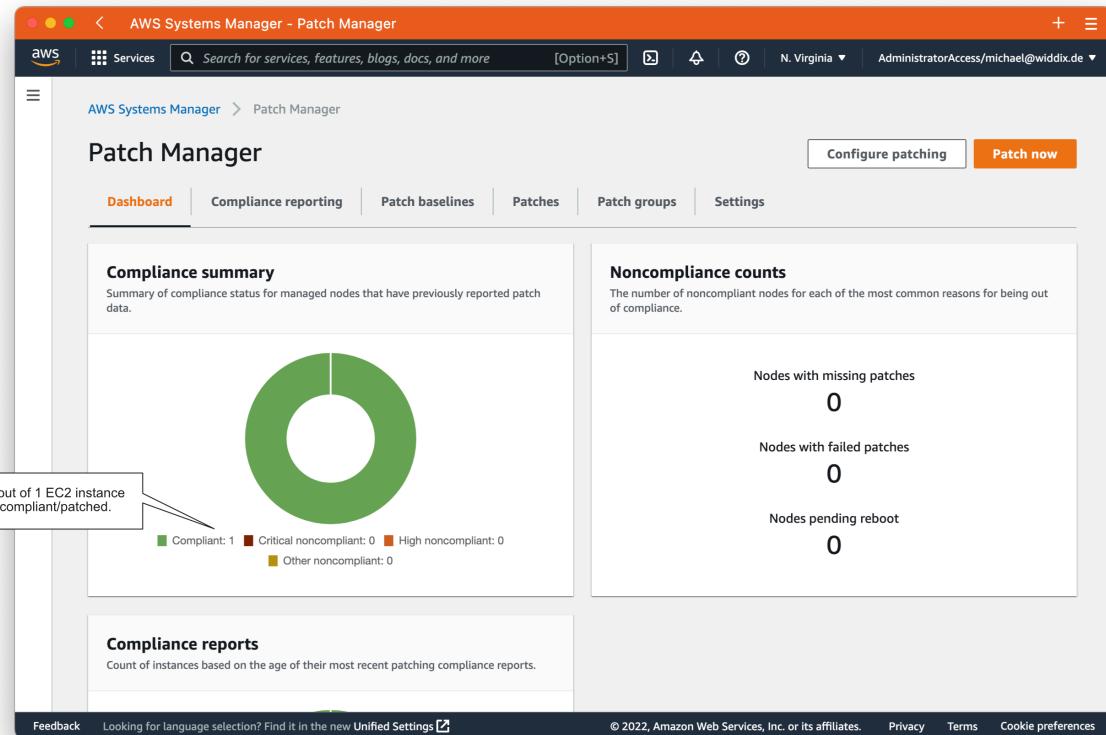
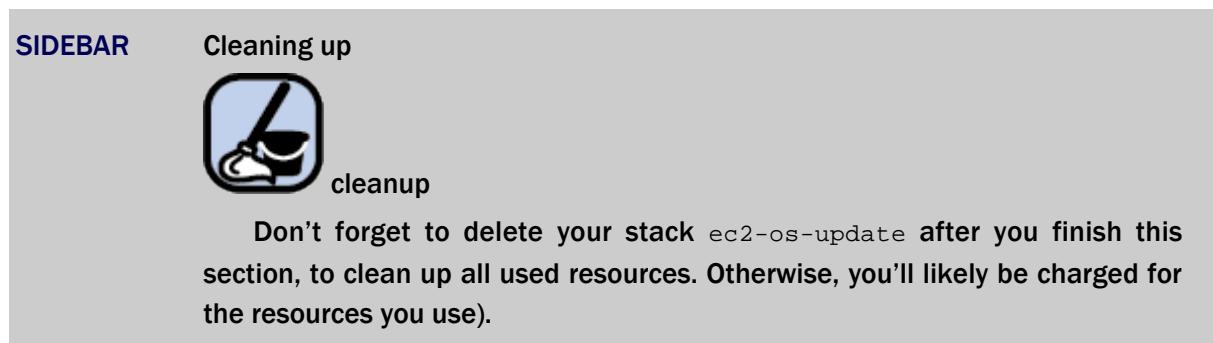


Figure 5.3 AWS Systems Manager Patch Manager dashboard

You can also patch instances manually by pressing the **patch now** button if needed.



5.3 Securing your AWS account

Securing your AWS account is critical. If someone gets access to your AWS account, they can steal your data, use resources at your expense, or delete all your data. As figure 5.4 shows, an AWS account is a basket for all the resources you own: EC2 instances, CloudFormation stacks, IAM users, and so on. Each AWS account comes with a root user granted unrestricted access to all resources. So far, you've used the AWS account root user to log into the Management Console and the user `mycli`—created in section 4.2—when using the CLI. In this section you will create an additional user to log into the Management Console, to avoid using the AWS account root user at all. Doing so allows you to manage multiple users, each restricted to the resources that are necessary for their roles.

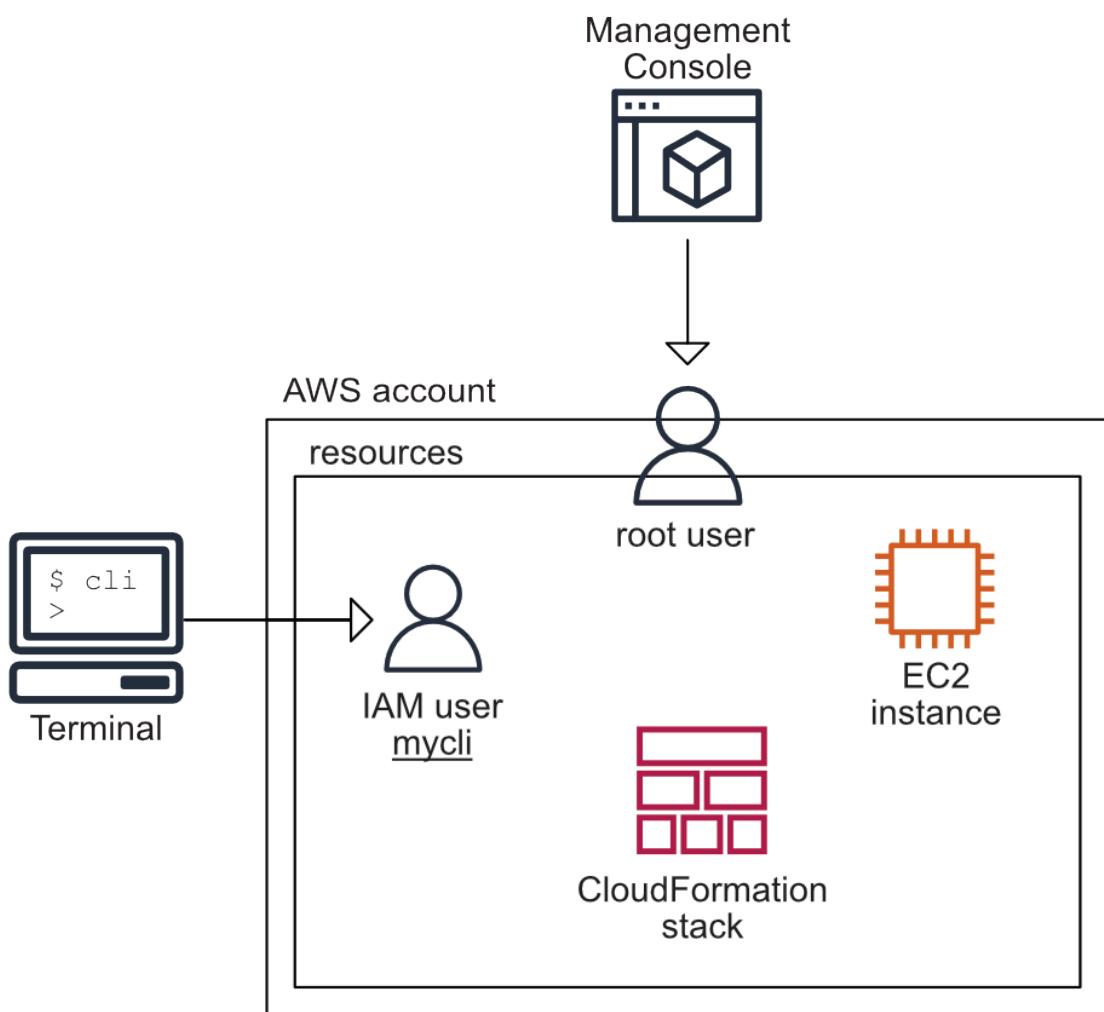


Figure 5.4 An AWS account contains all the AWS resources and comes with an AWS account root user by default.

To access your AWS account, an attacker must be able to authenticate to your account. There are

three ways to do so: using the AWS account root user, using an IAM user, or authenticating as an AWS resource like an EC2 instance. To authenticate as a AWS account root user or IAM user, the attacker needs the username and password or the access keys. To authenticate as an AWS resource like an EC2 instance, the attacker needs access to the machine to communicate with the instance metadata service (IMDS).

To protect yourself from an attacker stealing or cracking your passwords or access keys, you will enable multi-factor authentication (MFA) for your AWS account root user, to add an additional layer of security to the authentication process, in the following section.

5.3.1 Securing your AWS account's root user

We advise you to enable MFA for the AWS account root user of your AWS account. After MFA is activated, a password and a temporary token are needed to log in as the root user.

Follow these steps to enable MFA, as shown in figure [5.5](#):

1. Click your name in the navigation bar at the top right of the Management Console.
2. Select *Security credentials*.
3. Install an MFA app on your smartphone, one that supports the TOTP standard (such as Google Authenticator).
4. Expand the *Multi-factor authentication (MFA)* section.
5. Click *Activate MFA*.
6. Select *Virtual MFA device* and proceed with the next step.
7. Follow the instructions. Use the MFA app on your smartphone to scan the QR code that is displayed.

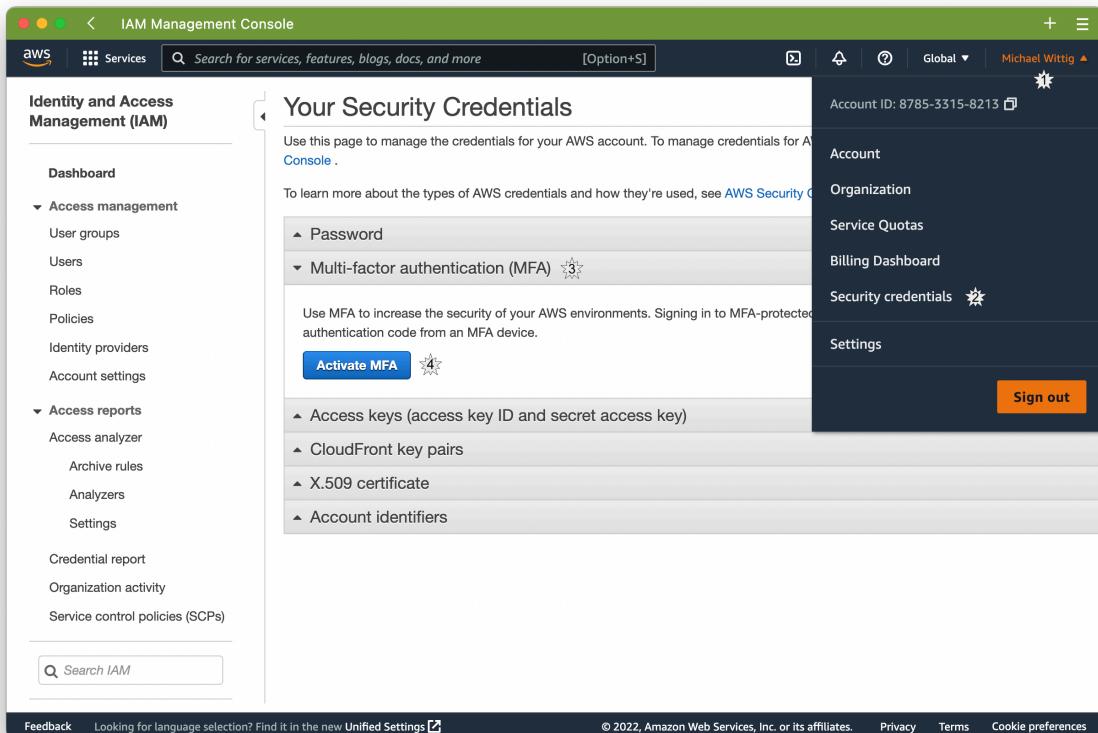


Figure 5.5 Protect your AWS account root user with multi-factor authentication (MFA)

If you’re using your smartphone as a virtual MFA device, it’s a good idea not to log in to the Management Console from your smartphone or to store the AWS account root user’s password on the phone. Keep the MFA token separate from your password. YubiKeys and hardware MFA tokens are also supported.

5.3.2 AWS Identity and Access Management (IAM)

Figure 5.6 shows an overview of all the core concepts of the *Identity and Access Management (IAM)* service. This service provides authentication and authorization for the AWS API. When you send a request to the AWS API, IAM verifies your identity and checks if you are allowed to perform the action. IAM controls who (authentication) can do what (authorization) in your AWS account. For example, is the user allowed to launch a new virtual machine?

- An *IAM user* is used to authenticate people or workloads running outside of AWS.
- An *IAM group* is a collection of IAM users with the same permissions.
- An *IAM role* is used to authenticate AWS resources, for example an EC2 instance.
- An *IAM identity policy* is used to define the permissions for a user, group, or role.

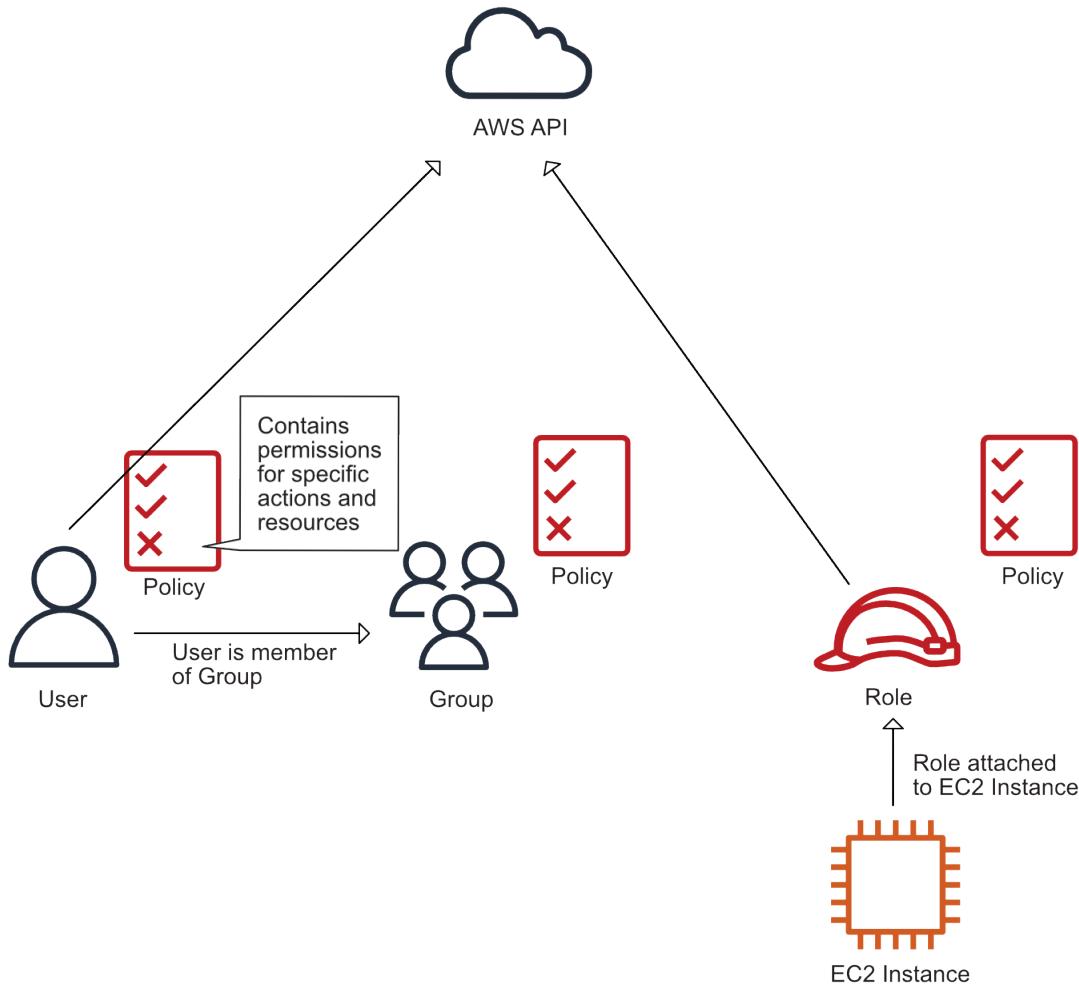


Figure 5.6 IAM concepts

Table 5.1 shows the differences between users and roles. Roles authenticate AWS entities such as EC2 instances. IAM users authenticate the people who manage AWS resources, for example system administrators, DevOps engineers, or software developers.

Table 5.1 Differences between AWS account root user, IAM user, and IAM role

	AWS account root user	IAM user	IAM role
Can have a password (needed to log into the AWS Management Console)	Always	Yes	No
Can have access keys (needed to send requests to the AWS API (for example, for CLI or SDK))	Yes (not recommended)	Yes	No
Can belong to a group	No	Yes	No
Can be associated with an EC2 instance, ECS container, Lambda function	No	No	Yes

By default, users and roles can't do anything. You have to create an identity policy stating what actions they're allowed to perform. IAM users and IAM roles use identity policies for authorization. Let's look at identity policies first.

5.3.3 Defining permissions with an IAM identity policy

By attaching one or multiple IAM identity policies to an IAM user or role, you are granting permissions to manage AWS resources. Identity policies are defined in JSON and contain one or more statements. A statement can either allow or deny specific actions on specific resources. The wildcard character * can be used to create more generic statements.

SIDE BAR
Identity versus resource policies

IAM policies come in two types. Identity policies are attached to users, groups, or roles. Resource policies are attached to resources. Only very few resource types support resource policies. One common example is the S3 bucket policy attached to S3 buckets.

If a policy contains the property `Principal` it is a resource policy. The `Principal` defines who is allowed to perform the action. Keep in mind that the principal can be set to the public.

The following identity policy has one statement that allows every action for the EC2 service, for all resources:

```
{
  "Version": "2012-10-17",          ①
  "Statement": [
    {
      "Effect": "Allow",           ②
      "Action": "ec2:*",           ③
      "Resource": "*"             ④
    }
  ]
}
```

- ① Specifies 2012-10-17 to lock down the version
- ② This statement allows access to actions and resources.
- ③ Any action offered by the EC2 service (wildcard *)
- ④ On any resource

If you have multiple statements that apply to the same action, `Deny` overrides `Allow`. The following identity policy allows all EC2 actions except terminating EC2 instances:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "ec2:*",
      "Resource": "*"
    },
    {
      "Effect": "Deny", ①
      "Action": "ec2:TerminateInstances", ②
      "Resource": "*"
    }
  ]
}
```

- ① Action is denied
- ② Terminating EC2 instances

The following identity policy denies all EC2 actions. The `ec2:TerminateInstances` statement isn't crucial, because `Deny` overrides `Allow`. When you deny an action, you can't allow that action with another statement:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Deny", ①
      "Action": "ec2:*",
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": "ec2:TerminateInstances", ②
      "Resource": "*"
    }
  ]
}
```

- ① Denies every EC2 action
- ② Allow isn't crucial; Deny overrides Allow.

So far, the `Resource` part has been set to `*` to apply to every resource. Resources in AWS have an Amazon Resource Name (ARN); figure 5.7 shows the ARN of an EC2 instance.

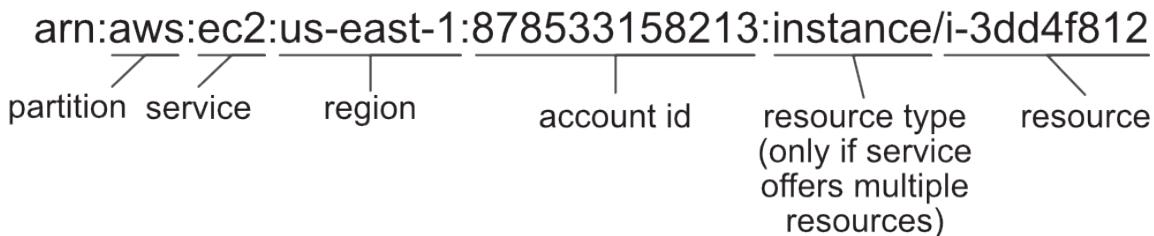


Figure 5.7 Components of an Amazon Resource Name (ARN) identifying an EC2 instance

To find out the account ID, you can use the CLI:

```
$ aws sts get-caller-identity --query "Account" --output text
111111111111 ①
```

- ① Account ID has always 12 digits.

If you know your account ID, you can use ARNs to allow access to specific resources of a service:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "ec2:TerminateInstances",
      "Resource":
[CA] "arn:aws:ec2:us-east-1:111111111111:instance/i-0b5c991e026104db9"
    }
  ]
}
```

The list of all IAM actions of a service and possible resource ARNs can be found here docs.aws.amazon.com/service-authorization/latest/reference/reference_policies_actions-resources-containers.html.

There are two types of identity policies:

Managed policy—If you want to create identity policies that can be reused in your account, a managed policy is what you’re looking for. There are two types of managed policies:

- *AWS managed policy*—An identity policy maintained by AWS. There are identity policies that grant admin rights, read-only rights, and so on.
- *Customer managed*—An identity policy maintained by you. It could be an identity policy that represents the roles in your organization, for example.

Inline policy—An identity policy that belongs to a certain IAM role, user, or group. An inline identity policy can’t exist without the IAM role, user, or group that it belongs to.

With CloudFormation, it’s easy to maintain inline identity policies; that’s why we use inline identity policies most of the time in this book. One exception is the `mycli` user: this user has the AWS managed policy `AdministratorAccess` attached.

WARNING Using managed policies can often conflict with following the least privilege principle. Managed policies usually set the `Resource` property to `*`.

5.3.4 Users for authentication, and groups to organize users

A user can authenticate using either a username and password, or access keys. When you log in to the Management Console, you’re authenticating with your username and password. When you use the CLI from your computer, you use access keys to authenticate as the `mycli` user.

You're using the AWS account root user at the moment to log in to the Management Console. You should create an IAM user instead, for two reasons.

- Creating IAM users allows you to set up a unique user for every person who needs to access your AWS account.
- You can grant access only to the resources each user needs, allowing you to follow the least privilege principle.

To make things easier if you want to add users in the future, you'll first create a group for all users with administrator access. Groups can't be used to authenticate, but they centralize authorization. So, if you want to stop your admin users from terminating EC2 instances, you only need to change the identity policy for the group instead of changing it for all admin users. A user can be a member of zero, one, or multiple groups.

It's easy to create groups and users with the CLI. Replace `$Password` in the following with a secure password:

```
$ aws iam create-group --group-name "admin"
$ aws iam attach-group-policy --group-name "admin" \
[CA] --policy-arn "arn:aws:iam::aws:policy/AdministratorAccess"
$ aws iam create-user --user-name "myuser"
$ aws iam add-user-to-group --group-name "admin" --user-name "myuser"
$ aws iam create-login-profile --user-name "myuser" --password '$Password'
```

The user `myuser` is ready to be used. But you must use a different URL to access the Management Console if you aren't using the AWS account root user: [\\$accountId.signin.aws.amazon.com/console](https://$accountId.signin.aws.amazon.com/console). Replace `$accountId` with the account ID that you extracted earlier with the `aws sts get-caller-identity` command.

SIDE BAR

Enabling MFA for IAM users

We encourage you to enable MFA for all users. To enable MFA for your users, follow these steps:

- Open the IAM service in the Management Console.
- Choose Users at left.
- Click on the `myuser` user.
- Select the Security credentials tab.
- Click the Manage link near to Assigned MFA device.
- The wizard to enable MFA for the IAM user is the same one you used for the AWS account root user.

We do recommend enabling MFA for all users, especially for users granted administrator access to all or some services.

WARNING Stop using the AWS account root user from now on. Always use `myuser` and the new link to the Management Console.

WARNING You should never copy a user's access keys to an EC2 instance; use IAM roles instead! Don't store security credentials in your source code. And never ever check them into your source code repository. Try to use IAM roles instead whenever possible as described in the next section.

5.3.5 Authenticating AWS resources with roles

There are various use cases where an EC2 instance needs to access or manage AWS resources. For example, an EC2 instance might need to:

- Back up data to the object store S3.
- Terminate itself after a job has been completed.
- Change the configuration of the private network environment in the cloud.

To be able to access the AWS API, an EC2 instance needs to authenticate itself. You could create an IAM user with access keys and store the access keys on an EC2 instance for authentication. But doing so is a hassle and violates security best practices, especially if you want to rotate the access keys regularly.

Instead of using an IAM user for authentication, you should use an IAM role whenever you need to authenticate AWS resources like EC2 instances. When using an IAM role, your access keys are injected into your EC2 instance automatically.

If an IAM role is attached to an EC2 instance, all identity policies attached to those roles are evaluated to determine whether the request is allowed. By default, no role is attached to an EC2 instance and therefore the EC2 instance is not allowed to make any calls to the AWS API.

The following example will show you how to use an IAM role for an EC2 instance. Do you remember the temporary EC2 instances from chapter 4? What if we forgot to terminate those VMs? A lot of money was wasted because of that. You'll now create an EC2 instance that stops itself automatically. The following snippet shows a one-liner terminating an EC2 instance after 5 minutes. The command `at` is used to execute the `aws ec2 stop-instances` with a 5-minute delay:

```
echo "aws ec2 stop-instances --instance-ids i-0b5c991e026104db9" \
[CA] | at now + 5 minutes
```

The EC2 instance needs permission to stop itself. Therefore, you need to attach an IAM role to the EC2 instance. The role contains an inline identity policy granting access to the

ec2:StopInstances action. Unfortunately, we can not lock down the action to the EC2 instance resource itself due to a cyclic dependency. Luckily, we can also grant permissions with additional conditions. One such condition is that a specific tag must be present. The following code shows how you define an IAM role with the help of CloudFormation:

```
Role:
  Type: 'AWS::IAM::Role'
  Properties:
    AssumeRolePolicyDocument:      ①
      Version: '2012-10-17'
      Statement:
        - Effect: Allow
          Principal: ②
            Service: 'ec2.amazonaws.com' ③
            Action: 'sts:AssumeRole' ④
    ManagedPolicyArns:
      - 'arn:aws:iam::aws:policy/AmazonSSMManagedInstanceCore'
    Policies:
      - PolicyName: ec2 ⑥
        PolicyDocument:
          Version: '2012-10-17'
          Statement:
            - Effect: Allow ⑧
              Action: 'ec2:StopInstances' ⑨
              Resource: '*' ⑩
              Condition: ⑪
                StringEquals:
                  'ec2:ResourceTag/aws:cloudformation:stack-id':
[CA] !Ref 'AWS::StackId'
```

- ① Who is allowed to assume this role?
- ② Specify the principal that is allowed access to the role.
- ③ Enter the EC2 service as the principle.
- ④ Allow the principle to assume the IAM role.
- ⑤ Define inline policies granting the role access to certain AWS resources and actions.
- ⑥ Define the name for the inline policy.
- ⑦ The policy document for the inline policy.
- ⑧ Allow ...
- ⑨ ... stopping virtual machines ...
- ⑩ ... for all EC2 instances ...
- ⑪ ... but only those who are tagged with the name of the CloudFormation stack.

To attach an inline role to an instance, you must first create an instance profile:

```
InstanceProfile:
  Type: 'AWS::IAM::InstanceProfile'
  Properties:
    Roles:
      - !Ref Role
```

The following listing shows how to attach the IAM role to the virtual machine:

```
Instance:
  Type: 'AWS::EC2::Instance'
  Properties:
    # [...]
    IamInstanceProfile: !Ref InstanceProfile
  UserData:
    'Fn::Base64': !Sub |
      #!/bin/bash -ex
      TOKEN=`curl -X PUT "http://169.254.169.254/latest/api/token" \
[CA] -H "X-aws-ec2-metadata-token-ttl-seconds: 21600"`
      INSTANCEID=`curl -H "X-aws-ec2-metadata-token: $TOKEN" \
[CA] -s "http://169.254.169.254/latest/meta-data/instance-id"`
      echo "aws ec2 stop-instances --region ${AWS::Region} \
[CA] --instance-ids $INSTANCEID" | at now + ${Lifetime} minutes
```

Create the CloudFormation stack with the template located at s3.amazonaws.com/awsinaction-code3/chapter05/ec2-iam-role.yaml by clicking on the [CloudFormation Quick-Create Link](#). Specify the lifetime of the EC2 instance via the parameter, and pick the default VPC and subnet as well. Wait until the amount of time specified as the lifetime has passed, and see if your EC2 instance is stopped in the EC2 Management Console. The lifetime begins when the server is fully started and booted.

SIDE BAR

Cleaning up



cleanup

Don't forget to delete your stack `ec2-iam-role` after you finish this section, to clean up all used resources. Otherwise, you'll likely be charged for the resources you use (even when your EC2 instance is stopped, you pay for the network attached storage).

You have learned how to use IAM users to authenticate people and IAM roles to authenticate EC2 instances or other AWS resources. You've also seen how to grant access to specific actions and resources by using an IAM identity policy. The next section will cover controlling network traffic to and from your virtual machine.

5.4 Controlling network traffic to and from your virtual machine

You only want traffic to enter or leave your EC2 instance if it has to do so. With a firewall, you control incoming (also called *inbound* or *ingress*) and outgoing (also called *outbound* or *egress*) traffic. If you run a web server, the only ports you need to open to the outside world are port 80 for HTTP traffic and 443 for HTTPS traffic. All other ports should be closed down. You should only open ports that must be accessible, just as you grant only the permissions you need with IAM. If you are using a firewall that allows only legitimate traffic, you close a lot of possible security holes. You can also prevent yourself from human failure, for example you prevent accidentally sending email to customers from a test system by not opening outgoing SMTP connections for test systems.

Before network traffic enters or leaves your EC2 instance, it goes through a firewalls provided by AWS. The firewalls inspects the network traffic and uses rules to decide whether the traffic is allowed or denied.

SIDE BAR IP vs. IP address

The abbreviation **IP** is used for Internet Protocol, whereas an **IP address** describes a specific address like **84.186.116.47**

Figure 5.8 shows how an SSH request from a source IP address 10.0.0.10 is inspected by the firewall and received by the destination IP address 10.10.0.20. In this case, the firewall allows the request because there is a rule that allows TCP traffic on port 22 between the source and the destination.

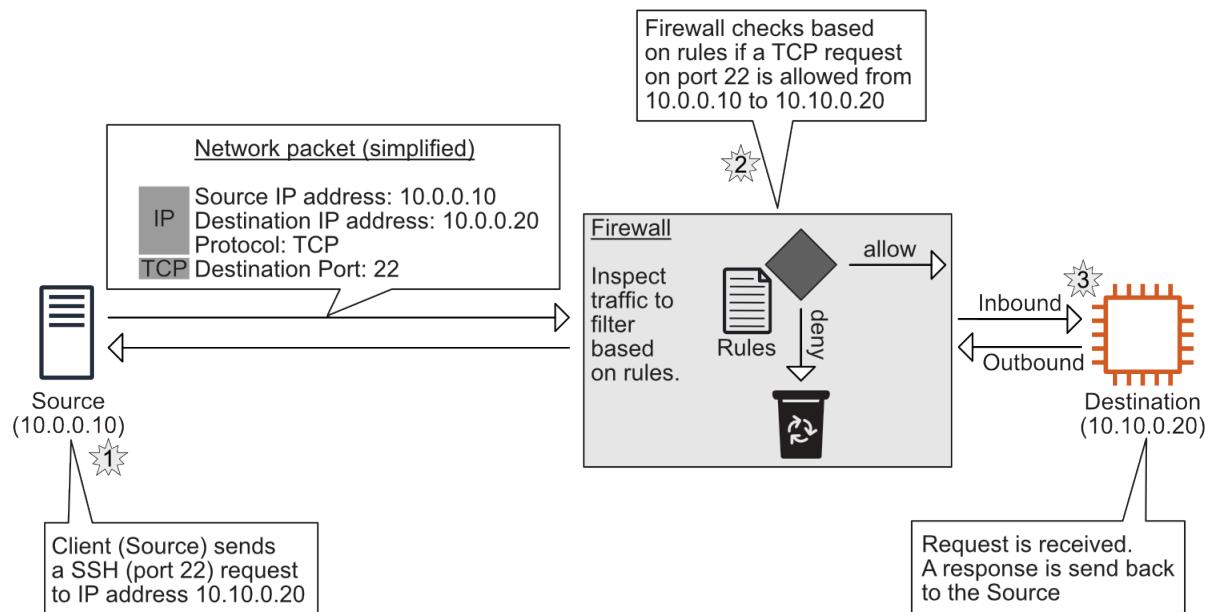


Figure 5.8 How an SSH request travels from source to destination, controlled by a firewall

AWS is responsible for the firewall, but you're responsible for the rules. By default, a security group does not allow any inbound traffic. You must add your own rules to allow specific incoming traffic. A security group contains a rule allowing all outbound traffic by default. If your use case requires a high level of networking security, you should remove the rule and add your own rules to control outgoing traffic.

SIDE BAR**Debugging or monitoring network traffic**

Imagine the following problem: Your EC2 instance does not accept SSH traffic as you want it to, but you can't spot any misconfiguration in your firewall rules. Two strategies are helpful:

1. Use the VPC Reachability Analyzer to simulate the traffic and see if the tool finds the configuration problem. Learn more here
docs.aws.amazon.com/vpc/latest/reachability/getting-started.html.
2. Enable VPC Flow Logs to get access to aggregated log messages containing rejected connections. Learn more here
docs.aws.amazon.com/vpc/latest/userguide/flow-logs.html.

5.4.1 Controlling traffic to virtual machines with security groups

Associate a security group with AWS resources such as EC2 instances to control traffic. It's common for EC2 instances to have more than one security group associated with them, and for the same security group to be associated with multiple EC2 instances.

A security group consists of a set of rules. Each rule allows network traffic based on the following:

- Direction (inbound or outbound)
- IP protocol (TCP, UDP, ICMP)
- Port
- Source/destination based on IP address, IP address range, or security group (works only within AWS)

In theory, you could define rules that allow all traffic to enter and leave your virtual machine; AWS won't prevent you from doing so. But it's best practice to define your rules, so they are as restrictive as possible.

Security group resources in CloudFormation are of type `AWS::EC2::SecurityGroup`. The following listing is in `/chapter05/firewall1.yaml` in the book's code folder: the template describes an empty security group associated with a single EC2 instance.

Listing 5.1 CloudFormation template: security group

```

---
[...]
Parameters:
  VPC:          ①
    # [...]
  Subnet:       ②
    # [...]
Resources:
  SecurityGroup: ③
    Type: 'AWS::EC2::SecurityGroup'
    Properties:
      GroupDescription: 'Learn how to protect your EC2 Instance.'
      VpcId: !Ref VPC
      Tags:
        - Key: Name
          Value: 'AWS in Action: chapter 5 (firewall)'
  Instance:     ④
    Type: 'AWS::EC2::Instance'
    Properties:
      # [...]
      SecurityGroupIds:
        - !Ref SecurityGroup ⑤
      SubnetId: !Ref Subnet

```

- ① You'll learn about this in section 6.5.
- ② You'll learn about this in section 6.5.
- ③ Defines the security group without any rules (by default, inbound traffic is denied and outbound traffic is allowed.) Rules will be added in the following sections.
- ④ Defines the EC2 instance
- ⑤ Associates the security group with the EC2 instance

To explore security groups, you can try the CloudFormation template located at s3.amazonaws.com/awsinaction-code3/chapter05/firewall1.yaml. Create a stack based on that template by clicking on the [CloudFormation Quick-Create Link](#), and then copy the `PublicIpAddress` from the stack output.

5.4.2 Allowing ICMP traffic

If you want to ping an EC2 instance from your computer, you must allow inbound Internet Control Message Protocol (ICMP) traffic. By default, all inbound traffic is blocked. Try `ping $PublicIpAddress` to make sure `ping` isn't working:

```

ping 34.205.166.12
PING 34.205.166.12 (34.205.166.12): 56 data bytes
Request timeout for icmp_seq 0
Request timeout for icmp_seq 1
[...]

```

You need to add a rule to the security group that allows inbound traffic, where the protocol equals ICMP. The following listing is in `/chapter05/firewall2.yaml` in the book's code folder.

Listing 5.2 CloudFormation template: security group that allows ICMP

```
SecurityGroup:
  Type: 'AWS::EC2::SecurityGroup'
  Properties:
    GroupDescription: 'Learn how to protect your EC2 Instance.'
    VpcId: !Ref VPC
    Tags:
      - Key: Name
        Value: 'AWS in Action: chapter 5 (firewall)'
    SecurityGroupIngress: ①
      - Description: 'allowing inbound ICMP traffic'
        IpProtocol: icmp ②
        FromPort: '-1' ③
        ToPort: '-1'
        CidrIp: '0.0.0.0/0' ④
```

- ① Rules allowing incoming traffic
- ② Specifies ICMP as the protocol
- ③ ICMP does not use ports. -1 means every port.
- ④ Allow traffic from any source IP address.

Update the CloudFormation stack with the template located at s3.amazonaws.com/awsinaction-code3/chapter05/firewall2.yaml and retry the ping command. It should work now:

```
$ ping 34.205.166.12
PING 34.205.166.12 (34.205.166.12): 56 data bytes
64 bytes from 34.205.166.12: icmp_seq=0 ttl=234 time=109.095 ms
64 bytes from 34.205.166.12: icmp_seq=1 ttl=234 time=107.000 ms
[...]
round-trip min/avg/max/stddev = 107.000/108.917/110.657/1.498 ms
```

Everyone's inbound ICMP traffic (every source IP address) is now allowed to reach your EC2 instance.

5.4.3 Allowing HTTP traffic

Once you can ping your EC2 instance, you want to run a web server. To do so, you must create a rule to allow inbound TCP requests on port 80. You also need a running web server.

Listing 5.3 CloudFormation template: security group that allows HTTP

```

SecurityGroup:
  Type: 'AWS::EC2::SecurityGroup'
  Properties:
    GroupDescription: 'Learn how to protect your EC2 Instance.'
    VpcId: !Ref VPC
    # [...]
    SecurityGroupIngress:
      # [...]
      - Description: 'allowing inbound HTTP traffic' ①
        IpProtocol: tcp ②
        FromPort: '80' ③
        ToPort: '80' ④
        CidrIp: '0.0.0.0/0' ⑤
  Instance:
    Type: 'AWS::EC2::Instance'
    Properties:
      # [...]
      UserData:
        'Fn::Base64': |
          #!/bin/bash -ex
          yum -y install httpd ⑥
          systemctl start httpd ⑦
          echo '<html>...</html>' > /var/www/html/index.html

```

- ① Adds a rule to allow incoming HTTP traffic
- ② HTTP is based on the TCP protocol.
- ③ The default HTTP port is 80.
- ④ You can allow a range of ports or set FromPort = ToPort.
- ⑤ Allows traffic from any source IP address
- ⑥ Install Apache HTTP Server on startup
- ⑦ Start Apache HTTP Server

Update the CloudFormation stack with the template located at s3.amazonaws.com/awsinaction-code3/chapter05/firewall3.yaml. Enter the public IP address in your browser to see a very basic test page.

5.4.4 Allowing HTTP traffic from a specific source IP address

So far, you're allowing inbound traffic on port 80 (HTTP) from every source IP address. It is possible to restrict access to only your own IP address for additional security as well.

SIDE BAR**What's the difference between public and private IP addresses?**

On my local network, I'm using private IP addresses that start with **192.168.0.***. My laptop uses **192.168.0.10**, and my iPad uses **192.168.0.20**. But if I access the internet, I have the same public IP address (such as **79.241.98.155**) for my laptop and iPad. That's because only my internet gateway (the box that connects to the internet) has a public IP address, and all requests are redirected by the gateway. (If you want to know more about this, search for network address translation.) Your local network doesn't know about this public IP address. My laptop and iPad only know that the internet gateway is reachable under **192.168.0.1** on the private network.

To find your public IP address, visit checkip.amazonaws.com/. For some of us, our public IP address changes from time to time, usually when you reconnect to the internet (which happens every 24 hours in my case).

Hard-coding the public IP address into the template isn't a good solution because your public IP address can change from time to time. But you already know the solution: parameters. You need to add a parameter that holds your current public IP address, and you need to modify the Security Group. You can find the following listing in /chapter05/firewall4.yaml in the book's code folder.

Listing 5.4 Security group allows traffic from source IP

```
Parameters:
  WhitelistedIpAddress: ❶
    Description: 'Whitelisted IP address'
    Type: String
    AllowedPattern: '^[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}$'
    ConstraintDescription: 'Enter a valid IPv4 address'
Resources:
  SecurityGroup:
    Type: 'AWS::EC2::SecurityGroup'
    Properties:
      GroupDescription: 'Learn how to protect your EC2 Instance.'
      VpcId: !Ref VPC
      # [...]
      SecurityGroupIngress:
        # [...]
        - Description: 'allowing inbound HTTP traffic'
          IpProtocol: tcp
          FromPort: '80'
          ToPort: '80'
          CidrIp: !Sub '${WhitelistedIpAddress}/32' ❷
```

- ❶ Public IP address parameter
- ❷ Uses `WhitelistedIpAddress/32` as a value to turn the IP address input into a CIDR

Update the CloudFormation stack with the template located at s3.amazonaws.com/awsinaction-code3/chapter05/firewall4.yaml. When asked for parameters, type in your public IP address for `WhitelistedIpAddress`. Now only your IP address

can open HTTP connections to your EC2 instance.

SIDE BAR
Classless Inter-Domain Routing (CIDR)

You may wonder what /32 means. To understand what's going on, you need to switch your brain into binary mode. An IP address is 4 bytes or 32 bits long. The /32 defines how many bits (32, in this case) should be used to form a range of addresses. If you want to define the exact IP address that is allowed, you must use all 32 bits.

But sometimes it makes sense to define a range of allowed IP addresses. For example, you can use 10.0.0.0/8 to create a range between 10.0.0.0 and 10.255.255.255, 10.0.0.0/16 to create a range between 10.0.0.0 and 10.0.255.255, or 10.0.0.0/24 to create a range between 10.0.0.0 and 10.0.0.255. You aren't required to use the binary boundaries (8, 16, 24, 32), but they're easier for most people to understand. You already used 0.0.0.0/0 to create a range that contains every possible IP address.

Now you can control network traffic that comes from outside a virtual machine or goes outside a virtual machine by filtering based on protocol, port, and source IP address.

5.4.5 Allowing HTTP traffic from a source security group

It is possible to control network traffic based on whether the source or destination belongs to a specific security group. For example, you can say that a MySQL database can only be accessed if the traffic comes from your web servers, or that only your proxy servers are allowed to access the web servers. Because of the elastic nature of the cloud, you'll likely deal with a dynamic number of virtual machines, so rules based on source IP addresses are difficult to maintain. This becomes easy if your rules are based on source security groups.

To explore the power of rules based on a source security group, let's look at the concept of a load balancer / ingress router / proxy. The client sends requests to the proxy. The proxy inspects the request and forwards it to the backend. The backend response is then passed back to the client by the proxy.

To implement the concept of an HTTP proxy, you must follow these two rules:

- Allow HTTP access to the proxy from 0.0.0.0/0 or a specific source address.
- Allow HTTP access to all backends only if the traffic source is the proxy.

Figure 5.9 shows that only the proxy is allowed to communicate with the backend over HTTP.

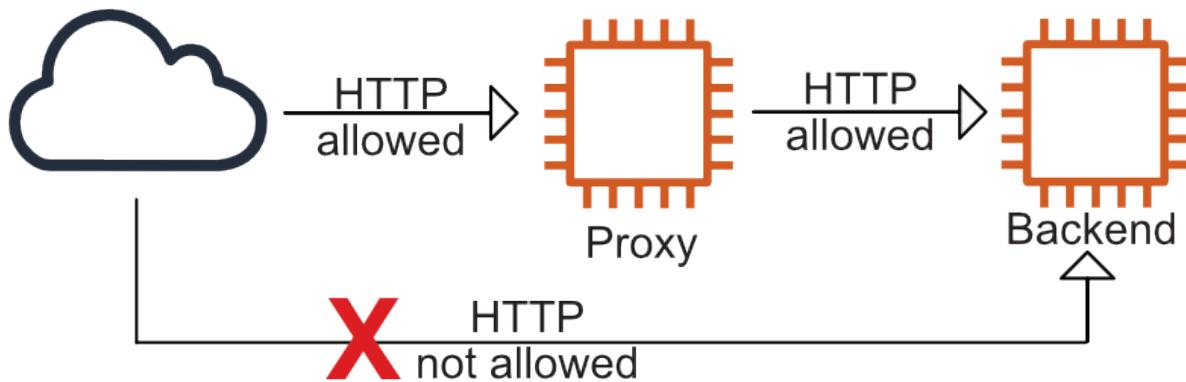


Figure 5.9 The proxy is the only HTTP entry point to the system (realized with security groups).

A security group allowing incoming HTTP traffic from anywhere needs to be attached to the proxy. All backend VMs are attached to a security group allowing HTTP traffic only if the source is the proxy's security group. The following listing shows the security groups defined in a CloudFormation template:

Listing 5.5 CloudFormation template: HTTP from proxy to backend

```

SecurityGroupProxy: ①
  Type: 'AWS::EC2::SecurityGroup'
  Properties:
    GroupDescription: 'Allowing incoming HTTP and ICMP from anywhere.'
    VpcId: !Ref VPC
    SecurityGroupIngress:
      - Description: 'allowing inbound ICMP traffic'
        IpProtocol: icmp
        FromPort: '-1'
        ToPort: '-1'
        CidrIp: '0.0.0.0/0'
      - Description: 'allowing inbound HTTP traffic'
        IpProtocol: tcp
        FromPort: '80'
        ToPort: '80'
        CidrIp: '0.0.0.0/0'
    SecurityGroupBackend: ②
      Type: 'AWS::EC2::SecurityGroup'
      Properties:
        GroupDescription: 'Allowing incoming HTTP from proxy.'
        VpcId: !Ref VPC
        SecurityGroupIngress:
          - Description: 'allowing inbound HTTP traffic from proxy'
            IpProtocol: tcp
            FromPort: '80'
            ToPort: '80'
            SourceSecurityGroupId: !Ref SecurityGroupProxy ③

```

- ① Security group attached to the proxy
- ② Security group attached to the backend
- ③ Allowing incoming HTTP traffic only from proxy

Update the CloudFormation stack with the template located at s3.amazonaws.com/awsinaction-code3/chapter05/firewall15.yaml. After the update is completed, the stack will show two outputs:

1. `ProxyPublicIpAddress`—Entry point into the system. Receives HTTP requests from the outside world.
2. `BackendPublicIpAddress`—You can connect to this EC2 instance only from the proxy.

Copy the `ProxyPublicIpAddress` output and open it in your browser. A `Hello AWS in Action!` page shows up.

But, when you copy the `BackendPublicIpAddress` output and try to open it in your browser, an error message like `` will appear. That's because the security group of the backend instance does not allow incoming traffic from your IP address but only from the proxy instance.

The only way to reach the backend instance is through the proxy instance. The `Hello AWS in Action!` page that appears when opening `$ProxyPublicIpAddress` in your browser, originates from the backend instance but gets passed through the proxy instance. Check the details of the HTTP request, for example with `curl` as shown in the following listing. You'll find a `x-backend` response header indicating, that the proxy forwarded your request to the backend named `app1` which points to the backend instance.

```
$ curl -I http://$ProxyPublicIpAddress
< HTTP/1.1 200 OK
[...]
< accept-ranges: bytes
< content-length: 91
< content-type: text/html; charset=UTF-8
< x-backend: app1 ①
<html><title>Hello AWS in Action!</title><body><h1>Hello AWS in Action!</h1></body></html>
```

- ① The `x-backend` header is injected by the proxy and indicates the backend answered the request.

SIDE BAR

Cleaning up



cleanup

Don't forget to delete your stack after you finish this section, to clean up all used resources. Otherwise, you'll likely be charged for the resources you use.

5.5 Creating a private network in the cloud: Amazon Virtual Private Cloud (VPC)

When you create a VPC, you get your own private network on AWS. *Private* means you can use the address ranges 10.0.0.0/8, 172.16.0.0/12, or 192.168.0.0/16 to design a network that isn't necessarily connected to the public internet. You can create subnets, route tables, Network Access Control Lists (NACLs), and gateways to the internet or a VPN endpoint.

SIDE BAR **IPv6**

Amazon Virtual Private Cloud supports IPv6 as well. You can create IPv4 only, IPv6 only, or IPv4 and IPv6 VPCs. To reduce complexity, we are sticking to IPv4 in this chapter.

Subnets allow you to separate concerns. We recommend to create at least two types of subnets:

- **Public subnets** for all resources, that need to be reachable from the Internet. For example, a load balancer of a Internet-facing web application.
- **Private subnets** for all resources, that should not be reachable from the Internet. For example, an application server or a database system.

What's the difference between a public and private subnet? A public subnet has a route to the internet; a private subnet doesn't.

For the purpose of understanding how a VPC works, you'll create a VPC to replicate the example from the previous section. You'll implement the proxy concept from the previous section by creating a public subnet that contains only the proxy. You'll also create a private subnet for the backend servers. You will not be able to access a backend server directly from the Internet as it will sit on a private subnet, the backend servers will only be accessible via the proxy server running in a public subnet.

The VPC uses the address space 10.0.0.0/16. To isolate different parts of the system, you'll add two public subnets and one private subnet to the VPC:

- 10.0.0.0/24 public subnet used later in this section to deploy a NAT gateway
- 10.0.1.0/24 public proxy subnet
- 10.0.2.0/24 private backend subnet

SIDE BAR **What does 10.0.0.0/16 mean?**

10.0.0.0/16 represents all IP addresses in 10.0.0.0 and 10.0.255.255. It's using CIDR notation (explained earlier in the chapter).

Figure [5.10](#) shows the architecture of the VPC.

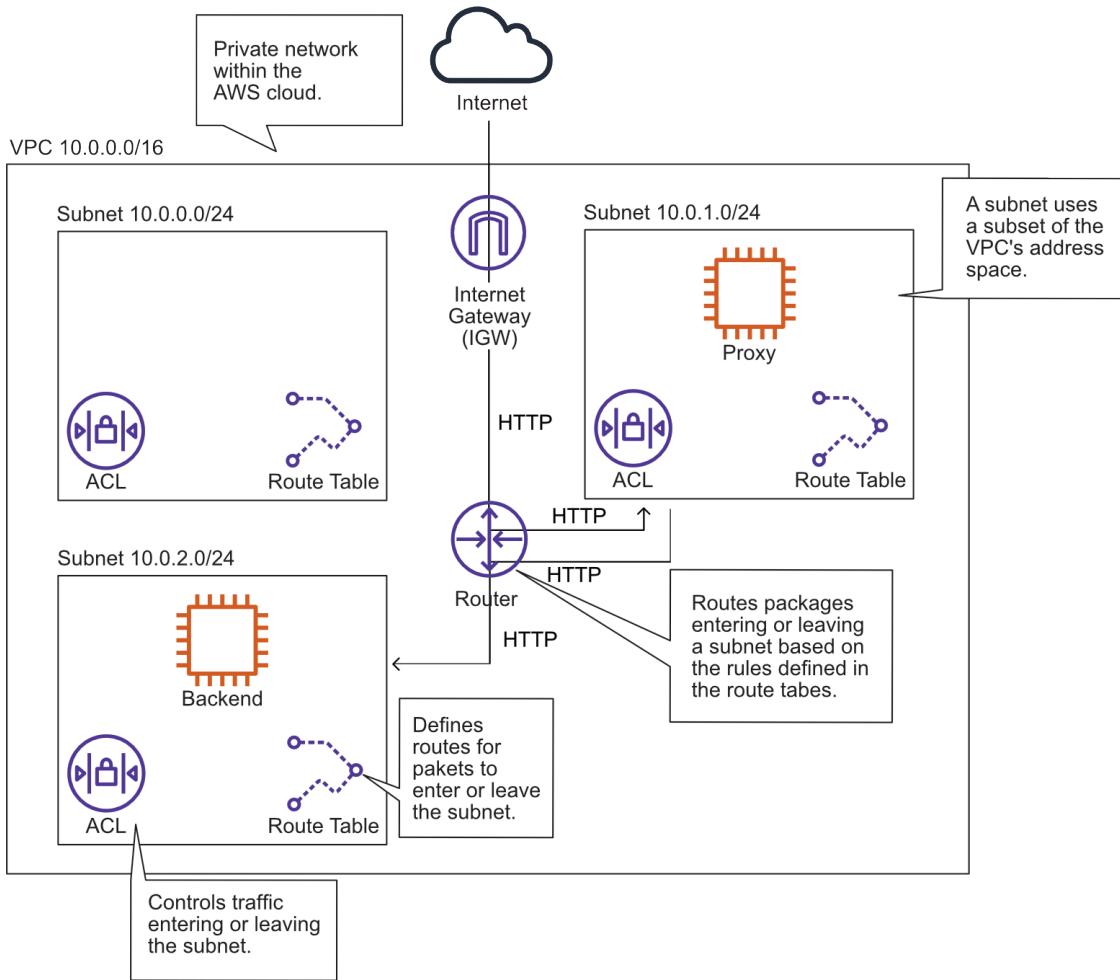


Figure 5.10 VPC with three subnets to secure a web application

You'll use CloudFormation to describe the VPC with its subnets. The template is split into smaller parts to make it easier to read in the book. As usual, you'll find the code in the book's code repository on GitHub: github.com/AWSinAction/code3. The template is located at `/chapter05/vpc.yaml``.

5.5.1 Creating the VPC and an internet gateway (IGW)

The first resources listed in the template are the VPC and the internet gateway (IGW). The IGW will translate the public IP addresses of your virtual machines to their private IP addresses using network address translation (NAT). All public IP addresses used in the VPC are controlled by this IGW.

```

VPC:
  Type: 'AWS::EC2::VPC'
  Properties:
    CidrBlock: '10.0.0.0/16'      ①
    EnableDnsHostnames: 'true'
    Tags:
      - Key: Name
        Value: 'AWS in Action: chapter 5 (VPC)'
  InternetGateway: ③
    Type: 'AWS::EC2::InternetGateway'
    Properties: {}
  VPCGatewayAttachment: ④
    Type: 'AWS::EC2::VPCGatewayAttachment'
    Properties:
      VpcId: !Ref VPC
      InternetGatewayId: !Ref InternetGateway

```

- ① The IP address space used for the private network
- ② Adds a Name tag to the VPC
- ③ An IGW is needed to enable traffic to and from the internet.
- ④ Attaches the internet gateway to the VPC

Next you'll define the subnet for the proxy.

5.5.2 Defining the public proxy subnet

The EC2 instance running the proxy needs to be reachable via the Internet. To achieve that, three steps are needed:

1. Create a subnet spanning a subsection of the IP address range assigned to the VPC.
2. Create a route table and attach it to the subnet.
3. Add the 0.0.0.0/0 route pointing to the Internet Gateway to the route table.
4. Create a Network Access Control List (NACL) and attach it to the subnet.

To allow traffic from the Internet to the proxy machine and from the proxy to the backend servers, you'll need these NACL rules:

- Internet to proxy: HTTP from 0.0.0.0/0 to 10.0.1.0/24 is allowed.
- proxy to backend: HTTP from 10.0.1.0/24 to 10.0.2.0/24 is allowed.

```

SubnetPublicProxy:
  Type: 'AWS::EC2::Subnet'
  Properties:
    AvailabilityZone: !Select [0, !GetAZs '' ]      ①
    CidrBlock: '10.0.1.0/24'      ②
    MapPublicIpOnLaunch: true
    VpcId: !Ref VPC
    Tags:
      - Key: Name
        Value: 'Public Proxy'
RouteTablePublicProxy: ③
  Type: 'AWS::EC2::RouteTable'
  Properties:
    VpcId: !Ref VPC
RouteTableAssociationPublicProxy: ④
  Type: 'AWS::EC2::SubnetRouteTableAssociation'
  Properties:
    SubnetId: !Ref SubnetPublicProxy
    RouteTableId: !Ref RouteTablePublicProxy
RoutePublicProxyToInternet:
  Type: 'AWS::EC2::Route'
  Properties:
    RouteTableId: !Ref RouteTablePublicProxy
    DestinationCidrBlock: '0.0.0.0/0'      ⑤
    GatewayId: !Ref InternetGateway
    DependsOn: VPCGatewayAttachment
NetworkAclPublicProxy: ⑥
  Type: 'AWS::EC2::NetworkAcl'
  Properties:
    VpcId: !Ref VPC
SubnetNetworkAclAssociationPublicProxy: ⑦
  Type: 'AWS::EC2::SubnetNetworkAclAssociation'
  Properties:
    SubnetId: !Ref SubnetPublicProxy
    NetworkAclId: !Ref NetworkAclPublicProxy

```

- ① Picks the first availability zone in the region. You'll learn availability zones in chapter 16.
- ② IP address space
- ③ Route table
- ④ Associates the route table with the subnet
- ⑤ Routes everything (0.0.0.0/0) to the IGW
- ⑥ Network Access Control List (NACL)
- ⑦ Associates the NACL with the subnet

There's an important difference between security groups and NACLs: security groups are stateful, but NACLs aren't. If you allow an inbound port on a security group, the corresponding response to requests on that port are allowed as well. A security group rule will work as you expect it to. If you open inbound port 80 on a security group, you can connect via HTTP.

That's not true for NACLs. If you open inbound port 80 on an NACL for your subnet, you still may not be able to connect via HTTP. In addition, you need to allow outbound ephemeral ports,

because the webserver accepts connections on port 80 but uses an ephemeral port for communication with the client. Ephemeral ports are selected from the range starting at 1024 and ending at 65535.

If you want to make an HTTP connection from within your subnet, you have to open outbound port 80 and inbound ephemeral ports as well.

There is another difference between security group rules and NACL rules: you have to define the priority for NACL rules. A smaller rule number indicates a higher priority. When evaluating an NACL the first rule that matches a package is applied; all other rules are skipped.

The proxy subnet allows clients sending HTTP requests on port 80. The following NACL rules are needed:

- Allow inbound port 80 (HTTP) from 0.0.0.0/0.
- Allow outbound ephemeral ports to 0.0.0.0/0.

We also want to make HTTP and HTTPS requests from the subnet to the Internet. The following NACL rules are needed:

- Allow inbound ephemeral ports from 0.0.0.0/0.
- Allow outbound port 80 (HTTP) to 0.0.0.0/0.
- Allow outbound port 443 (HTTPS) to 0.0.0.0/0.

Find the CloudFormation implementation of the above NACLs below:

```

NetworkAclEntryInPublicProxyHTTP: ①
  Type: 'AWS::EC2::NetworkAclEntry'
  Properties:
    NetworkAclId: !Ref NetworkAclPublicProxy
    RuleNumber: '110' ②
    Protocol: '6'
    PortRange:
      From: '80'
      To: '80'
    RuleAction: 'allow'
    Egress: 'false' ③
    CidrBlock: '0.0.0.0/0'
NetworkAclEntryInPublicProxyEphemeralPorts: ④
  Type: 'AWS::EC2::NetworkAclEntry'
  Properties:
    NetworkAclId: !Ref NetworkAclPublicProxy
    RuleNumber: '200'
    Protocol: '6'
    PortRange:
      From: '1024'
      To: '65535'
    RuleAction: 'allow'
    Egress: 'false'
    CidrBlock: '0.0.0.0/0'
NetworkAclEntryOutPublicProxyHTTP: ⑤
  Type: 'AWS::EC2::NetworkAclEntry'
  Properties:
    NetworkAclId: !Ref NetworkAclPublicProxy
    RuleNumber: '100'
    Protocol: '6'
    PortRange:
      From: '80'
      To: '80'
    RuleAction: 'allow'
    Egress: 'true' ⑥
    CidrBlock: '0.0.0.0/0'
NetworkAclEntryOutPublicProxyHTTPS: ⑦
  Type: 'AWS::EC2::NetworkAclEntry'
  Properties:
    NetworkAclId: !Ref NetworkAclPublicProxy
    RuleNumber: '110'
    Protocol: '6'
    PortRange:
      From: '443'
      To: '443'
    RuleAction: 'allow'
    Egress: 'true'
    CidrBlock: '0.0.0.0/0'
NetworkAclEntryOutPublicProxyEphemeralPorts: ⑧
  Type: 'AWS::EC2::NetworkAclEntry'
  Properties:
    NetworkAclId: !Ref NetworkAclPublicProxy
    RuleNumber: '200'
    Protocol: '6'
    PortRange:
      From: '1024'
      To: '65535'
    RuleAction: 'allow'
    Egress: 'true'
    CidrBlock: '0.0.0.0/0'

```

- ① Allows inbound HTTP from everywhere
- ② Rules are evaluated starting with the lowest numbered rule.
- ③ Inbound

- ④ Ephemeral ports used for short-lived TCP/IP connections
- ⑤ Allows outbound HTTP to everywhere
- ⑥ Outbound
- ⑦ Allows outbound HTTPS to everywhere
- ⑧ Ephemeral ports

We do recommend to start with using security groups to control traffic. If you want to add an extra layer of security, you should use NACLs on top. But doing so is optional, in our opinion.

5.5.3 Adding the private backend subnet

As shown in figure 5.11, the only difference between a public and a private subnet is that a private subnet doesn't have a route to the IGW.

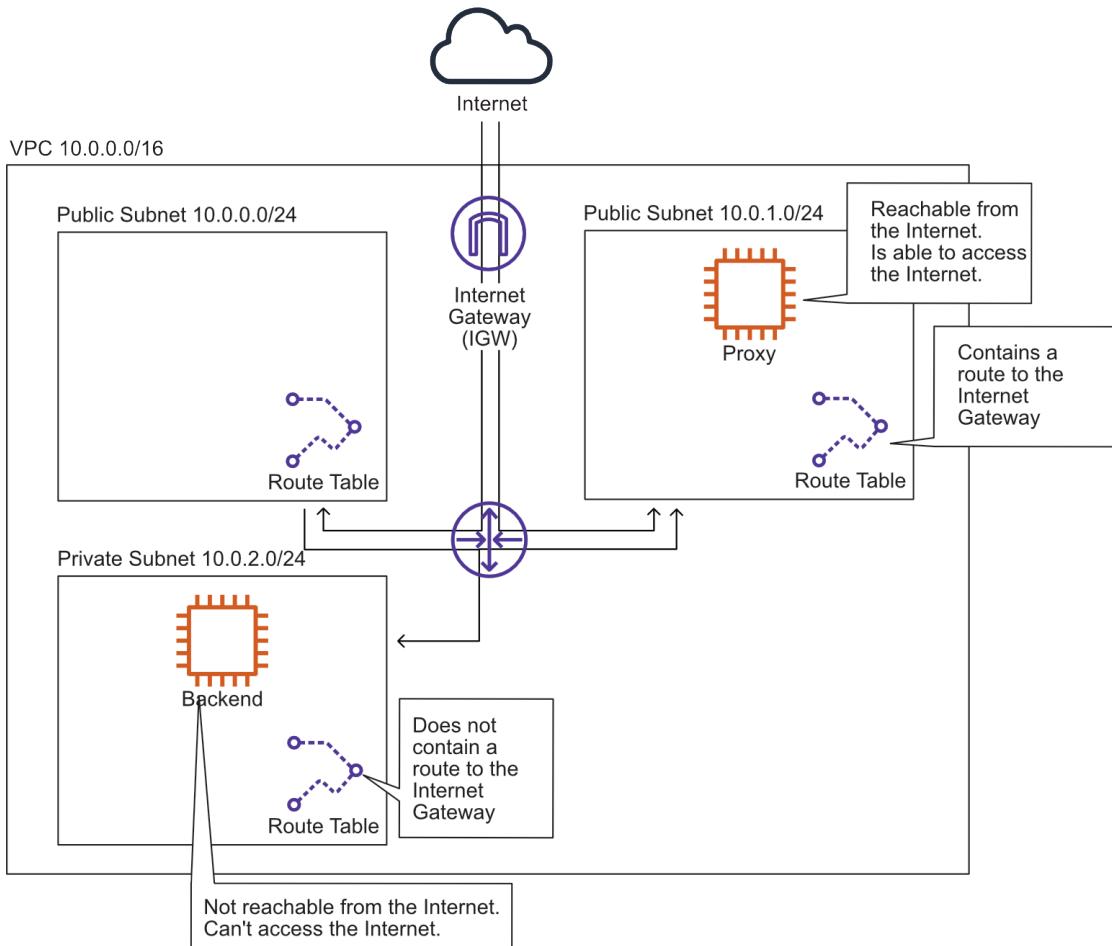


Figure 5.11 Private and public subnets

Traffic between subnets of a VPC is always routed by default. You can't remove the routes between the subnets. If you want to prevent traffic between subnets in a VPC, you need to use NACLs attached to the subnets.

The subnet for the web server has no additional routes and is therefore private.

```

SubnetPrivateBackend:
  Type: 'AWS::EC2::Subnet'
  Properties:
    AvailabilityZone: !Select [0, !GetAZs '']
    CidrBlock: '10.0.2.0/24'      ①
    VpcId: !Ref VPC
    Tags:
      - Key: Name
        Value: 'Private Backend'
RouteTablePrivateBackend: ②
  Type: 'AWS::EC2::RouteTable'
  Properties:
    VpcId: !Ref VPC
RouteTableAssociationPrivateBackend:
  Type: 'AWS::EC2::SubnetRouteTableAssociation'
  Properties:
    SubnetId: !Ref SubnetPrivateBackend
    RouteTableId: !Ref RouteTablePrivateBackend
NetworkAclPrivateBackend:
  Type: 'AWS::EC2::NetworkAcl'
  Properties:
    VpcId: !Ref VPC
SubnetNetworkAclAssociationPrivateBackend:
  Type: 'AWS::EC2::SubnetNetworkAclAssociation'
  Properties:
    SubnetId: !Ref SubnetPrivateBackend
    NetworkAclId: !Ref NetworkAclPrivateBackend

```

① Address space

② No route to the IGW

The backend subnet allows HTTP requests from the proxy subnet. The following NACLs are needed:

- Allow inbound port 80 (HTTP) from 10.0.1.0/24.
- Allow outbound ephemeral ports to 10.0.1.0/24.

We also want to make HTTP and HTTPS requests from the subnet to the Internet. Keep in mind that we have no route to the Internet yet. There is no way to access the Internet even with the NACLs. You will change this soon. The following NACLs are needed:

- Allow inbound ephemeral ports from 0.0.0.0/0.
- Allow outbound port 80 (HTTP) to 0.0.0.0/0.
- Allow outbound port 443 (HTTPS) to 0.0.0.0/0.

Find the CloudFormation implementation of the above NACLs below:

```

NetworkAclEntryInPrivateBackendHTTP: ①
  Type: 'AWS::EC2::NetworkAclEntry'
  Properties:
    NetworkAclId: !Ref NetworkAclPrivateBackend
    RuleNumber: '110'
    Protocol: '6'
    PortRange:
      From: '80'
      To: '80'
    RuleAction: 'allow'
    Egress: 'false'
    CidrBlock: '10.0.1.0/24'
NetworkAclEntryInPrivateBackendEphemeralPorts: ②
  Type: 'AWS::EC2::NetworkAclEntry'
  Properties:
    NetworkAclId: !Ref NetworkAclPrivateBackend
    RuleNumber: '200'
    Protocol: '6'
    PortRange:
      From: '1024'
      To: '65535'
    RuleAction: 'allow'
    Egress: 'false'
    CidrBlock: '0.0.0.0/0'
NetworkAclEntryOutPrivateBackendHTTP: ③
  Type: 'AWS::EC2::NetworkAclEntry'
  Properties:
    NetworkAclId: !Ref NetworkAclPrivateBackend
    RuleNumber: '100'
    Protocol: '6'
    PortRange:
      From: '80'
      To: '80'
    RuleAction: 'allow'
    Egress: 'true'
    CidrBlock: '0.0.0.0/0'
NetworkAclEntryOutPrivateBackendHTTPS: ④
  Type: 'AWS::EC2::NetworkAclEntry'
  Properties:
    NetworkAclId: !Ref NetworkAclPrivateBackend
    RuleNumber: '110'
    Protocol: '6'
    PortRange:
      From: '443'
      To: '443'
    RuleAction: 'allow'
    Egress: 'true'
    CidrBlock: '0.0.0.0/0'
NetworkAclEntryOutPrivateBackendEphemeralPorts: ⑤
  Type: 'AWS::EC2::NetworkAclEntry'
  Properties:
    NetworkAclId: !Ref NetworkAclPrivateBackend
    RuleNumber: '200'
    Protocol: '6'
    PortRange:
      From: '1024'
      To: '65535'
    RuleAction: 'allow'
    Egress: 'true'
    CidrBlock: '10.0.1.0/24'

```

- ① Allow inbound HTTP from proxy subnet
- ② Ephemeral ports
- ③ Allow outbound HTTP to everywhere

- ④ Allow outbound HTTPS to everywhere
- ⑤ Ephemeral ports

5.5.4 Launching virtual machines in the subnets

Your subnets are ready, and you can continue with the EC2 instances. First you describe the proxy:

```
SecurityGroup:
  Type: 'AWS::EC2::SecurityGroup'
  Properties:
    GroupDescription: 'Allowing all incoming and outgoing traffic.'
    VpcId: !Ref VPC
    SecurityGroupIngress:
      - IpProtocol: '-1'
        FromPort: '-1'
        ToPort: '-1'
        CidrIp: '0.0.0.0/0'
    SecurityGroupEgress:
      - IpProtocol: '-1'
        FromPort: '-1'
        ToPort: '-1'
        CidrIp: '0.0.0.0/0'
  Proxy:
    Type: AWS::EC2::Instance
    Properties:
      ImageId: 'ami-061ac2e015473fbe2'
      InstanceType: 't2.micro'
      IamInstanceProfile: 'ec2-ssm-core'
      SecurityGroupIds:
        - !Ref SecurityGroup ①
      SubnetId: !Ref SubnetPublicProxy ②
      Tags:
        - Key: Name
          Value: Proxy
      UserData: # [...]
    DependsOn: VPCGatewayAttachment ③
```

- ① This security group allows everything.
- ② Launches in the proxy subnet
- ③ We need to manually tell CloudFormation about a dependency here. Without the attachment being created, the instance could launch but without access to the Internet.

The private backend has a different configuration:

```

Backend:
  Type: 'AWS::EC2::Instance'
  Properties:
    ImageId: 'ami-061ac2e015473fbe2'
    InstanceType: 't2.micro'
    IamInstanceProfile: 'ec2-ssm-core'
    SecurityGroupIds:
      - !Ref SecurityGroup
    SubnetId: !Ref SubnetPrivateBackend ①
  Tags:
    - Key: Name
      Value: Backend
  UserData:
    'Fn::Base64': |
      #!/bin/bash -ex
      yum -y install httpd ②
      systemctl start httpd ③
      echo '<html>...</html>' > /var/www/html/index.html

```

- ① Launches in the private backend subnet
- ② Installs Apache from the internet
- ③ Starts Apache web server

You're now in serious trouble: installing Apache won't work because your private subnet has no route to the internet. Therefore, the `yum install` command will fail, as the public Yum repository is not reachable without access to the Internet.

5.5.5 Accessing the internet from private subnets via a NAT gateway

Public subnets have a route to the internet gateway. You can use a similar mechanism to provide outbound internet connectivity for EC2 instances running in private subnets without having a direct route to the internet: create a NAT gateway in a public subnet, and create a route from your private subnet to the NAT gateway. This way, you can reach the internet from private subnets, but the internet can't reach your private subnets. A NAT gateway is a managed service provided by AWS that handles network address translation. Internet traffic from your private subnet will access the internet from the public IP address of the NAT gateway.

SIDE BAR**Reducing costs for NAT gateway**

You have to pay for the traffic processed by a NAT gateway (see VPC Pricing at aws.amazon.com/vpc/pricing/ for more details). If your EC2 instances in private subnets will have to transfer huge amounts of data to the Internet, there are two options to decrease costs.

- Moving your EC2 instances from the private subnet to a public subnet allows them to transfer data to the internet without utilizing the NAT gateway. Use firewalls to strictly restrict incoming traffic from the internet.
- If data is transferred over the internet to reach AWS services (such as Amazon S3 and Amazon DynamoDB), use gateway VPC endpoints. These endpoints allow your EC2 instances to communicate with S3 and DynamoDB directly and at no additional charge. Furthermore, most other services are accessible from private subnets via interface VPC endpoints (offered by AWS PrivateLink) with an hourly and bandwidth fee.

WARNING

NAT gateways are finite resources. A NAT Gateway process up to 100 Gbit/s of traffic and up to 10 million packets per second. A NAT gateway is also bound to an availability zone (introduced in chapter 16).

To keep concerns separated, you'll create a subnet for the NAT gateway.

```

SubnetPublicNAT:
  Type: 'AWS::EC2::Subnet'
  Properties:
    AvailabilityZone: !Select [0, !GetAZs '']
    CidrBlock: '10.0.0.0/24' ①
    MapPublicIpOnLaunch: true
    VpcId: !Ref VPC
    Tags:
      - Key: Name
        Value: 'Public NAT'
RouteTablePublicNAT:
  Type: 'AWS::EC2::RouteTable'
  Properties:
    VpcId: !Ref VPC
RouteTableAssociationPublicNAT:
  Type: 'AWS::EC2::SubnetRouteTableAssociation'
  Properties:
    SubnetId: !Ref SubnetPublicNAT
    RouteTableId: !Ref RouteTablePublicNAT
RoutePublicNATTToInternet: ②
  Type: 'AWS::EC2::Route'
  Properties:
    RouteTableId: !Ref RouteTablePublicNAT
    DestinationCidrBlock: '0.0.0.0/0'
    GatewayId: !Ref InternetGateway
    DependsOn: VPCGatewayAttachment
NetworkAclPublicNAT:
  Type: 'AWS::EC2::NetworkAcl'
  Properties:
    VpcId: !Ref VPC
SubnetNetworkAclAssociationPublicNAT:
  Type: 'AWS::EC2::SubnetNetworkAclAssociation'
  Properties:
    SubnetId: !Ref SubnetPublicNAT
    NetworkAclId: !Ref NetworkAclPublicNAT

```

- ① 10.0.0.0/24 is the NAT subnet.
- ② The NAT subnet is public with a route to the internet.

We need a bunch of NACL rules to make the NAT gateway work.

To allow all VPC subnets to use the NAT Gateway for HTTP and HTTPS:

- Allow inbound port 80 (HTTP) and 443 (HTTPS) from 10.0.0.0/16.
- Allow outbound ephemeral ports to 10.0.0.0/16.

To allow the NAT gateway to reach out to the Internet on HTTP and HTTPS:

- Allow outbound port 80 (HTTP) and 443 (HTTPS) to 0.0.0.0/0.
- Allow inbound ephemeral ports from 0.0.0.0/0.

Find the CloudFormation implementation of the above NACL rules below:

```

NetworkAclEntryInPublicNATHHTTP: ①
  Type: 'AWS::EC2::NetworkAclEntry'
  Properties:
    NetworkAclId: !Ref NetworkAclPublicNAT
    RuleNumber: '100'
    Protocol: '6'

```

```

PortRange:
  From: '80'
  To: '80'
  RuleAction: 'allow'
  Egress: 'false'
  CidrBlock: '10.0.0.0/16'
NetworkAclEntryInPublicNATHHTPS: ②
  Type: 'AWS::EC2::NetworkAclEntry'
  Properties:
    NetworkAclId: !Ref NetworkAclPublicNAT
    RuleNumber: '110'
    Protocol: '6'
    PortRange:
      From: '443'
      To: '443'
      RuleAction: 'allow'
      Egress: 'false'
      CidrBlock: '10.0.0.0/16'
NetworkAclEntryInPublicNATEphemeralPorts: ③
  Type: 'AWS::EC2::NetworkAclEntry'
  Properties:
    NetworkAclId: !Ref NetworkAclPublicNAT
    RuleNumber: '200'
    Protocol: '6'
    PortRange:
      From: '1024'
      To: '65535'
      RuleAction: 'allow'
      Egress: 'false'
      CidrBlock: '0.0.0.0/0'
NetworkAclEntryOutPublicNATHHTTP: ④
  Type: 'AWS::EC2::NetworkAclEntry'
  Properties:
    NetworkAclId: !Ref NetworkAclPublicNAT
    RuleNumber: '100'
    Protocol: '6'
    PortRange:
      From: '80'
      To: '80'
      RuleAction: 'allow'
      Egress: 'true'
      CidrBlock: '0.0.0.0/0'
NetworkAclEntryOutPublicNATHHTPS: ⑤
  Type: 'AWS::EC2::NetworkAclEntry'
  Properties:
    NetworkAclId: !Ref NetworkAclPublicNAT
    RuleNumber: '110'
    Protocol: '6'
    PortRange:
      From: '443'
      To: '443'
      RuleAction: 'allow'
      Egress: 'true'
      CidrBlock: '0.0.0.0/0'
NetworkAclEntryOutPublicNATEphemeralPorts: ⑥
  Type: 'AWS::EC2::NetworkAclEntry'
  Properties:
    NetworkAclId: !Ref NetworkAclPublicNAT
    RuleNumber: '200'
    Protocol: '6'
    PortRange:
      From: '1024'
      To: '65535'

```

```
RuleAction: 'allow'
Egress: 'true'
CidrBlock: '10.0.0.0/16'
```

- ① Allow inbound HTTP from all VPC subnets
- ② Allow inbound HTTPS from all VPC subnets
- ③ Ephemeral ports
- ④ Allow outbound HTTP to everywhere
- ⑤ Allow outbound HTTPS to everywhere
- ⑥ Ephemeral ports

Finally, you can add the NAT gateway itself. The NAT gateway comes with a fixed public IP address (also called Elastic IP address). All traffic that is routed through the NAT gateway will come from this IP address. We also add a route to the backend's route table to route traffic to 0.0.0.0/0 via the NAT gateway.

```
EIPNatGateway: ①
  Type: 'AWS::EC2::EIP'
  Properties:
    Domain: 'vpc'
NatGateway: ②
  Type: 'AWS::EC2::NatGateway'
  Properties:
    AllocationId: !GetAtt 'EIPNatGateway.AllocationId'
    SubnetId: !Ref SubnetPublicNAT
RoutePrivateBackendToInternet:
  Type: 'AWS::EC2::Route'
  Properties:
    RouteTableId: !Ref RouteTablePrivateBackend
    DestinationCidrBlock: '0.0.0.0/0'
    NatGatewayId: !Ref NatGateway ③
```

- ① A static public IP address is used for the NAT gateway.
- ② The NAT gateway is placed into the private subnet and associated with the static public IP address.
- ③ Route from the Apache subnet to the NAT gateway

Lastly, one small tweak to the already covered backend EC2 instance is needed to keep dependencies up-to-date:

```
Backend:
  Type: 'AWS::EC2::Instance'
  Properties:
    # [...]
  DependsOn: RoutePrivateBackendToInternet ①
```

- ① Help CloudFormation to understand that the route is needed before the EC2 instance can be created.

WARNING The NAT gateway included in the example is not covered by the Free Tier. The NAT gateway will cost you \$0.045 USD per hour and \$0.045 per GB of data processed when creating the stack in the US East (N. Virginia) region. Go to aws.amazon.com/vpc/pricing/ to have a look at the current prices.

Now you're ready to create the CloudFormation stack with the template located at s3.amazonaws.com/awsinaction-code3/chapter05/vpc.yaml by clicking on the [CloudFormation Quick-Create Link](#). Once you've done so, copy the `ProxyPublicIpAddress` output and open it in your browser. You'll see an Apache test page.

SIDE BAR

Cleaning up



cleanup

Don't forget to delete your stack after finishing this section, to clean up all used resources. Otherwise, you'll likely be charged for the resources you use.

5.6 Summary

- AWS is a shared-responsibility environment in which security can be achieved only if you and AWS work together. You're responsible for securely configuring your AWS resources and your software running on EC2 instances, while AWS protects buildings and host systems.
- Keeping your software up-to-date is key, and can be automated.
- The Identity and Access Management (IAM) service provides everything needed for authentication and authorization with the AWS API. Every request you make to the AWS API goes through IAM to check whether the request is allowed. IAM controls who can do what in your AWS account. To protect your AWS account, grant only those permissions that your users and roles need.
- Traffic to or from AWS resources like EC2 instances can be filtered based on protocol, port, and source or destination.
- A VPC is a private network in AWS where you have full control. With VPCs, you can control routing, subnets, NACLs, and gateways to the internet or your company network via VPN. A NAT gateway enables access to the internet from private subnets.
- You should separate concerns in your network to reduce potential damage if, for example, one of your subnets is hacked. Keep every system in a private subnet that doesn't need to be accessed from the public internet, to reduce your attackable surface.



Automating operational tasks with Lambda

This chapter covers

- Creating a Lambda function to perform periodic health checks of a website
- Triggering a Lambda function with EventBridge events to automate DevOps tasks
- Searching through your Lambda function's logs with CloudWatch
- Monitoring Lambda functions with CloudWatch alarms
- Configuring IAM roles so Lambda functions can access other services

This chapter is about adding a new tool to your toolbox. The tool we're talking about, AWS Lambda, is as flexible as a Swiss Army Knife. You don't need a virtual machine to run your own code anymore, as AWS Lambda offers execution environments for C#/.NET Core, Go, Java, JavaScript/Node.js, Python, and Ruby. All you have to do is to implement a function, upload your code, and configure the execution environment. Afterward, your code is executed within a fully managed computing environment. AWS Lambda is well-integrated with all parts of AWS, enabling you to easily automate operations tasks within your infrastructure. We use AWS to automate our infrastructure regularly. For example, we use it to add and remove instances to a container cluster based on a custom algorithm, and to process and analyze log files.

AWS Lambda offers a maintenance-free and highly available computing environment. You no longer need to install security updates, replace failed virtual machines, or manage remote access (such as SSH or RDP) for administrators. On top of that, AWS Lambda is billed by invocation. Therefore, you don't have to pay for idling resources that are waiting for work (for example, for a task triggered once a day).

In our first example, you will create a Lambda function that performs periodic health checks for your website. This will teach you how to use the Management Console to get started with AWS Lambda quickly. In our second example, you will learn how to write your own Python code and

deploy a Lambda function in an automated way using CloudFormation—which we introduced in chapter 4. Your Lambda function will automatically add a tag to newly launched EC2 instances. At the end of the chapter, we’ll show you additional use cases like building web applications, Internet of Things (IoT) back ends, or processing data with AWS Lambda.

SIDE BAR**Examples are almost covered by the Free Tier**

The examples in this chapter are mostly covered by the Free Tier. There is one exception: AWS CloudTrail. In case you already created a trail in your account additional charges, most likely just a few cents will apply. For details please see aws.amazon.com/cloudtrail/pricing/.

You will find instructions on how to clean up the examples at the end of each section.

But what is AWS Lambda? Before diving into our first real-world example, we will start with a short introduction.

6.1 Executing your code with AWS Lambda

Computing capacity is available at different layers of abstraction on AWS: virtual machines, containers, and functions. You learned about the virtual machines offered by Amazon’s EC2 service in chapter 3. Containers offer another layer of abstraction on top of virtual machines; you will learn about containers in chapter 18. *AWS Lambda* provides computing power as well but in a fine-grained manner: an execution environment for small functions, rather than a full-blown operating system or container.

6.1.1 What is serverless?

When reading about AWS Lambda, you might have stumbled upon the term *serverless*. Peter Sbarski, in his book *Serverless Architectures on AWS* (Manning) summarizes the confusion created by this catchy and provocative phrase:

*[...] the word *serverless* is a bit of a misnomer. Whether you use a compute service such as AWS Lambda to execute your code, or interact with an API, there are still servers running in the background. The difference is that these servers are hidden from you. There's no infrastructure for you to think about and no way to tweak the underlying operating system. Someone else takes care of the nitty-gritty details of infrastructure management, freeing your time for other things.*

— Peter Sbarski _*Serverless Architectures on AWS* (Manning)

We define a serverless system as one that meets the following criteria:

- No need to manage and maintain virtual machines.
- Fully managed service offering scalability and high availability.
- Billed per request and by resource consumption.

AWS is not the only provider offering a serverless platform. Google (Cloud Functions) and Microsoft (Azure Functions) are other competitors in this area.

6.1.2 Running your code on AWS Lambda

AWS Lambda is the basic building block of the serverless platform provided by AWS. The first step in the process is to run your code on Lambda instead of on your own server.

As shown in figure 6.1, to execute your code with AWS Lambda, follow these steps:

1. Write the code.
2. Upload your code and its dependencies (such as libraries or modules).
3. Create a function determining the runtime environment and configuration.
4. Invoke the function to execute your code in the cloud.

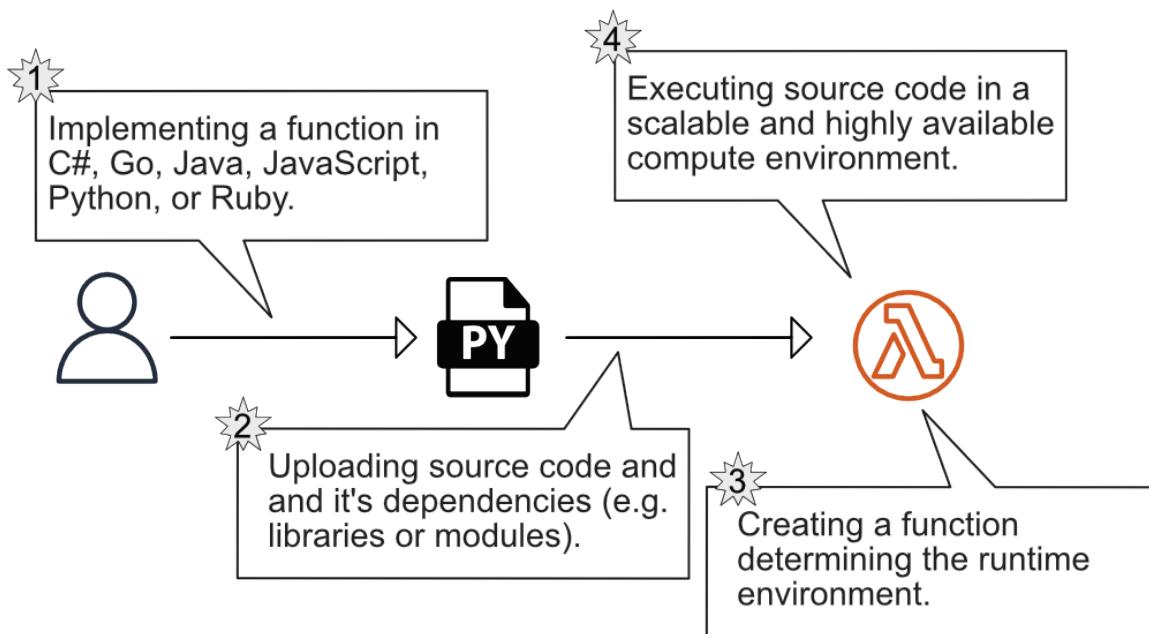


Figure 6.1 Executing code with AWS Lambda

Currently, AWS Lambda offers runtime environments for the following languages:

- C#/.NET Core
- Go
- Java
- JavaScript/Node.js
- Python
- Ruby

Besides that, you can bring your own runtime by using a custom runtime. In theory, a custom

runtime supports any programming language. You need to bring your own container image and follow conventions for initializing the function and processing tasks. Doing so adds extra complexity, so we recommend to go with one of the available runtimes.

Next, we will compare AWS Lambda with EC2 virtual machines.

6.1.3 Comparing AWS Lambda with virtual machines (Amazon EC2)

What is the difference between using AWS Lambda and virtual machines? First, there is the granularity of virtualization. Virtual machines provide a full operating system for running one or multiple applications. In contrast, AWS Lambda offers an execution environment for a single function, a small part of an application.

Furthermore, Amazon EC2 offers virtual machines as a service, but you are responsible for operating them in a secure, scalable and highly available way. Doing so requires you to put a substantial amount of effort into maintenance. By contrast, when building with Lambda, AWS manages the underlying infrastructure for you and provides a production-ready infrastructure.

Beyond that, AWS Lambda is billed per execution, and not per second like when a virtual machine is running. You don't have to pay for unused resources that are waiting for requests or tasks. For example, running a script to check the health of a website every 5 minutes on a virtual machine would cost you about \$3.71 USD. Executing the same health check with AWS Lambda will cost about \$0.04 USD.

Table [6.1](#) compares AWS Lambda and virtual machines in detail. You'll find a discussion of AWS Lambda's limitations at the end of the chapter.

Table 6.1 AWS Lambda compared to Amazon EC2

	AWS Lambda	Amazon EC2
Granularity of virtualization	Small piece of code (a function)	An entire operating system
Scalability	Scales automatically. A throttling limit prevents you from creating unwanted charges accidentally and can be increased by AWS support if needed.	As you will learn in chapter 17, using an Auto Scaling Group allows you to scale the number of EC2 instances serving requests automatically. But configuring and monitoring the scaling activities is your responsibility.
High availability	Fault tolerant by default. The computing infrastructure spans multiple machines and data centers.	Virtual machines are not highly available by default. Nevertheless, as you will learn in chapter 13 it is possible to set up a highly available infrastructure based on EC2 instances as well.
Maintenance effort	Almost zero. You need only to configure your function.	You are responsible for maintaining all layers between your virtual machine's operating system and your application's runtime environment.
Deployment effort	Almost zero due to a well-defined API	Rolling out your application to a fleet of virtual machines is a challenge that requires tools and know-how.
Pricing model	Pay per request as well as execution time and allocated memory	Pay for operating hours of the virtual machines, billed per second.

Looking for limitations and pitfalls of AWS Lambda? Stay tuned: you will find a discussion of Lambda's limitations at the end of the chapter.

That's all you need to know about AWS Lambda to be able to go through the first real-world example. Are you ready?

6.2 Building a website health check with AWS Lambda

Are you responsible for the uptime of a website or application? We do our best to make sure our blog clondonaut.io is accessible 24/7. An external health check acts as a safety net making sure we, and not our readers, are the first to know when our blog goes down. AWS Lambda is the perfect choice for building a website health check, as you do not need computing resources constantly, but only every few minutes for a few milliseconds. This section guides you through setting up a health check for your website based on AWS Lambda.

In addition to AWS Lambda, we are using the Amazon CloudWatch service for this example. Lambda functions publish metrics to CloudWatch by default. Typically you inspect metrics using charts, and create alarms by defining thresholds. For example, a metric could count failures during the function's execution. On top of that, EventBridge provides events that can be used to trigger Lambda functions as well. We are using a rule to publish an event every 5 minutes here.

As shown in figure [6.2](#), your website health check will consist of three parts:

1. *Lambda function*—Executes a Python script that sends an HTTP request to your website (for example GET clondonaut.io) and verifies that the response includes specific text (such as `clondonaut`).
2. *EventBridge rule*—Triggers the Lambda function every 5 minutes. This is comparable to the cron service on Linux.
3. *Alarm* --Monitors the number of failed health checks and notifies you via email whenever your website is unavailable.

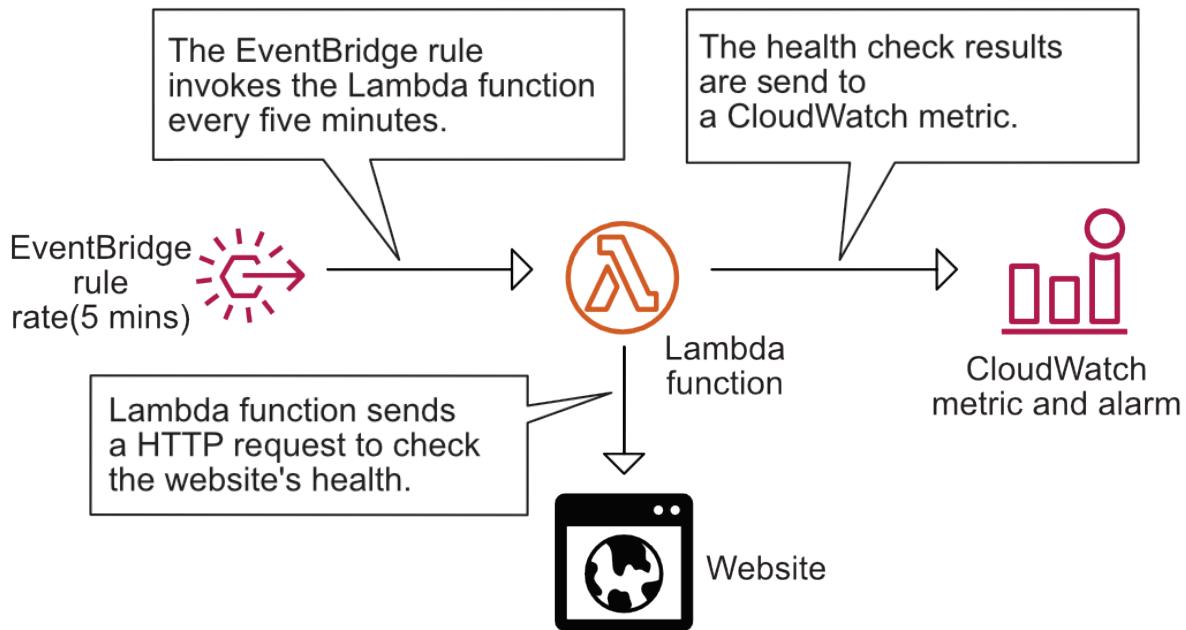


Figure 6.2 The Lambda function performing the website health check is executed every five minutes by a scheduled event. Errors are reported to CloudWatch

You will use the Management Console to create and configure all the necessary parts manually. In our opinion this is a simple way to get familiar with AWS Lambda. You will learn how to deploy a Lambda function in an automated way in section 7.3.

6.2.1 Creating a Lambda function

The following step-by-step instructions guide you through setting up a website health check based on AWS Lambda. Open AWS Lambda in the Management Console: console.aws.amazon.com/lambda/home. Click *Create a function* to start the Lambda function wizard, as shown in figure 6.3.

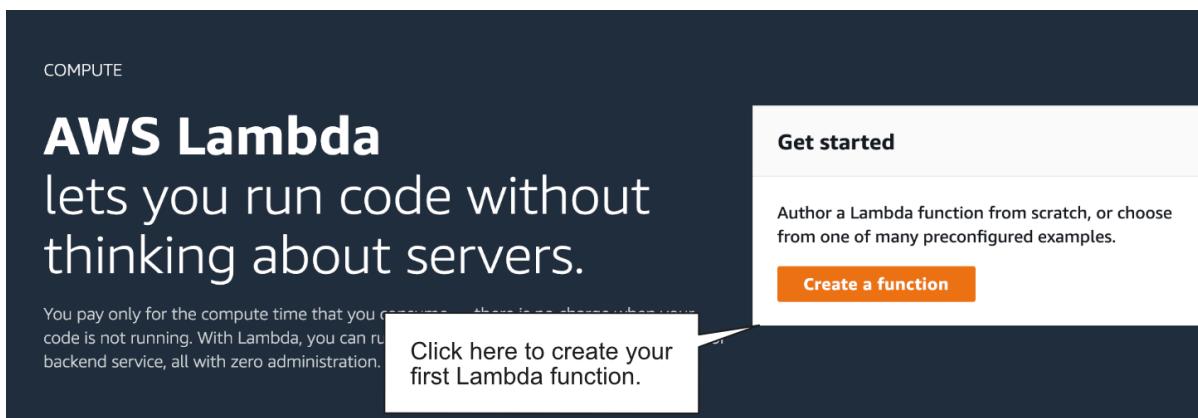


Figure 6.3 Welcome screen: ready to create your first Lambda function

AWS provides blueprints for various use cases, including the code and the Lambda function configuration. We will use one of these blueprints to create a website health check. Select *Use a blueprint* and search for canary. Next, choose the lambda-canary blueprint. Figure 6.4 illustrates the details.

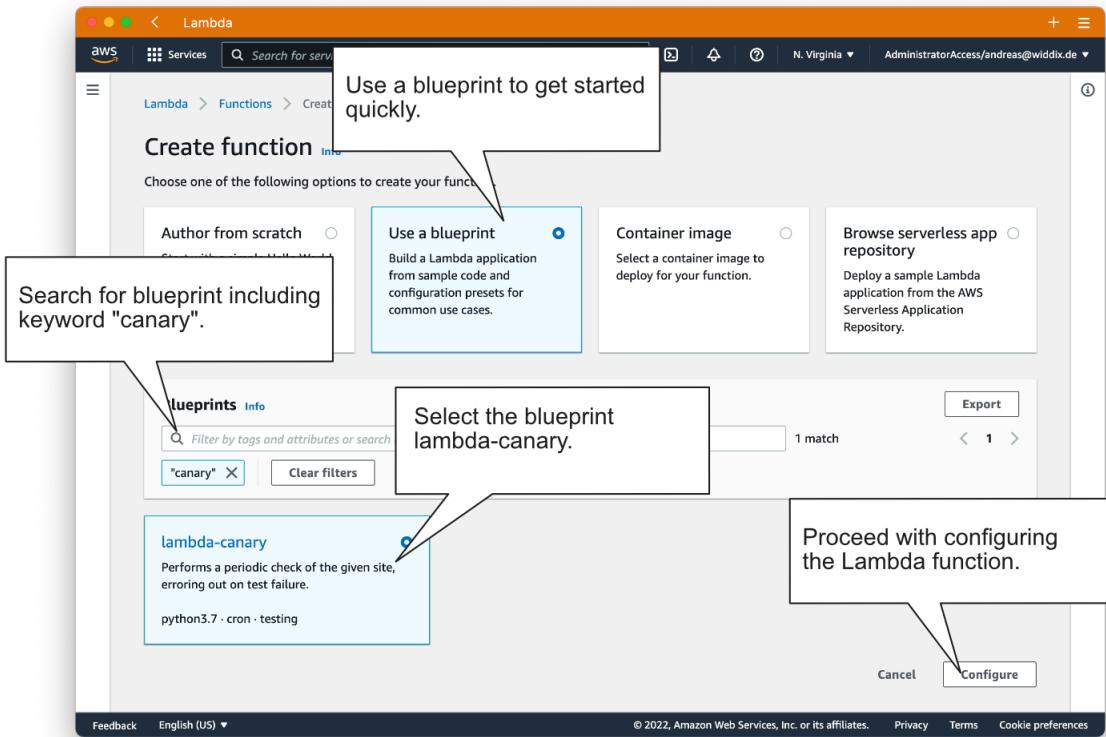


Figure 6.4 Creating a Lambda function based on a blueprint provided by AWS

In the next step of the wizard, you need to specify a name for your Lambda function, as shown in figure 6.5. The function name needs to be unique within your AWS account and the current region US East (N. Virginia), also the name is limited to 64 characters. To invoke a function via the API you need to provide the function name, for example. Type in `website-health-check` as the name for your Lambda function.

Continue with creating an IAM role. Select *Create a new role with basic Lambda permissions* as shown in figure 6.5 to create an IAM role for your Lambda function. You will learn how your Lambda function makes use of the IAM role in section 7.3.

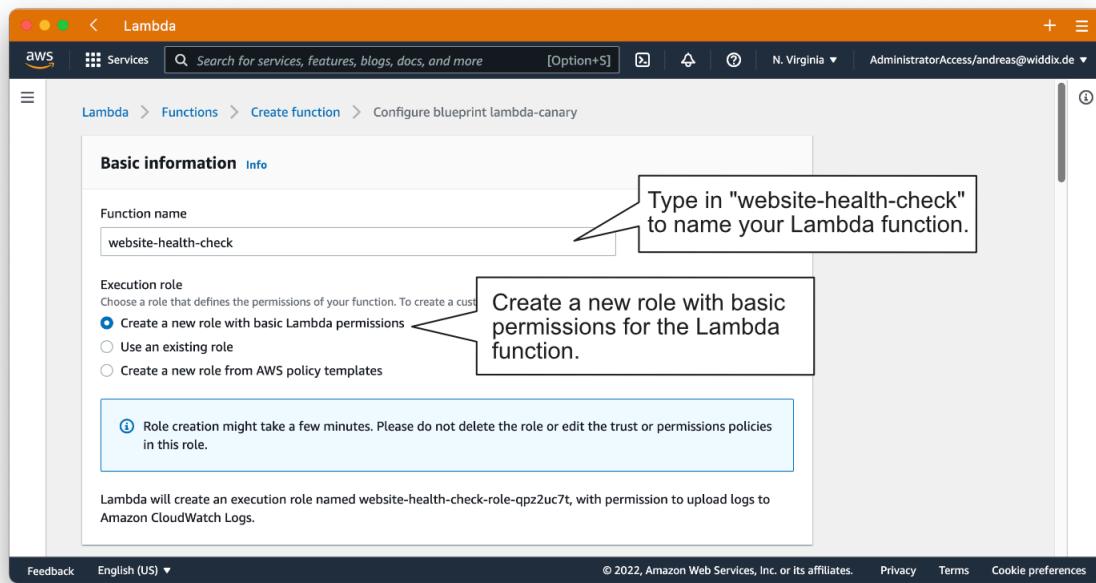


Figure 6.5 Creating a Lambda function: choose a name and define an IAM role

Next, configure the scheduled event that will trigger your health check repeatedly. We will use an interval of 5 minutes in this example. Figure 6.6 shows the settings you need.

1. Select *Create a new rule* to create a scheduled event rule.
2. Type in `website-health-check` as the name for the rule.
3. Enter a description that will help you to understand what is going on if you come back later.
4. Select *Schedule expression* as the rule type. You will learn about the other option, *Event pattern*, at the end of this chapter.
5. Use `rate(5 minutes)` as the schedule expression.

Define recurring tasks using a *schedule expression* in form of `rate($value $unit)`. For example, you could trigger a task every 5 minutes, every hour, or once a day. `$value` needs to be a positive integer value. Use minute, minutes, hour, hours, day, or days as the unit. For example, instead of triggering the website health check every 5 minutes, you could use `rate(1 hour)` as the schedule expression to execute the health check every hour. Note that frequencies of less than one minute are not supported.

It is also possible to use the crontab format when defining a schedule expression.

```

cron($minutes $hours $dayOfMonth $month $dayOfWeek $year)

# Invoke a Lambda function at 08:00am (UTC) everyday
cron(0 8 * * ? *)

# Invoke a Lambda function at 04:00pm (UTC) every monday to friday
cron(0 16 ? * MON-FRI *)

```

See “Schedule expressions using rate or cron” at

docs.aws.amazon.com/lambda/latest/dg/tutorial-scheduled-events-schedule-expressions.html for more details.

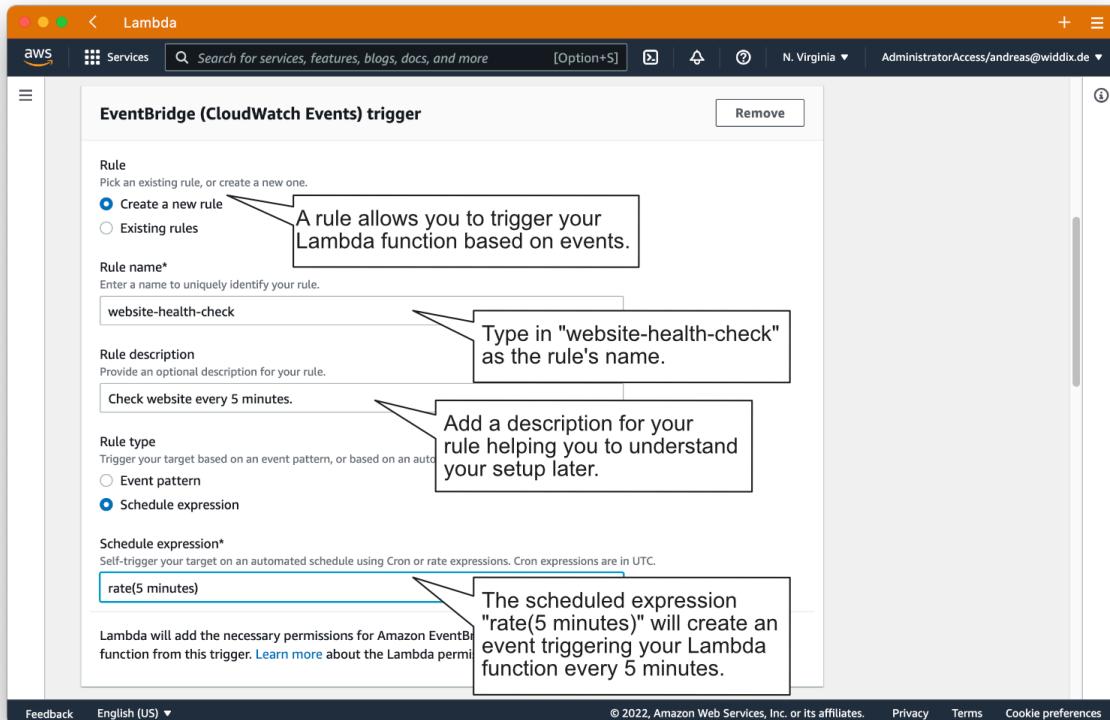


Figure 6.6 Configuring a scheduled event triggering your Lambda function every 5 minutes

Your Lambda function is missing an integral part: the code. As you are using a blueprint, AWS has inserted the Python code implementing the website health check for you, as shown in figure [6.7](#).

The Python code references two environment variables: `site` and `expected`. Environment variables are commonly used to dynamically pass settings to your function.

An environment variable consists of a key and a value. Specify the following environment variables for your Lambda function:

1. `site`—Contains the URL of the website you want to monitor. Use clondonaut.io if you do not have a website to monitor yourself.
2. `expected`—Contains a text snippet that must be available on your website. If the function doesn't find this text, the health check fails. Use `clondonaut` if you are using clondonaut.io as site.

The Lambda function is reading the environment variables during its execution:

```
SITE = os.environ['site']
EXPECTED = os.environ['expected']
```

After defining the environment variables for your Lambda function, click the *Create function* button at the bottom of the screen.

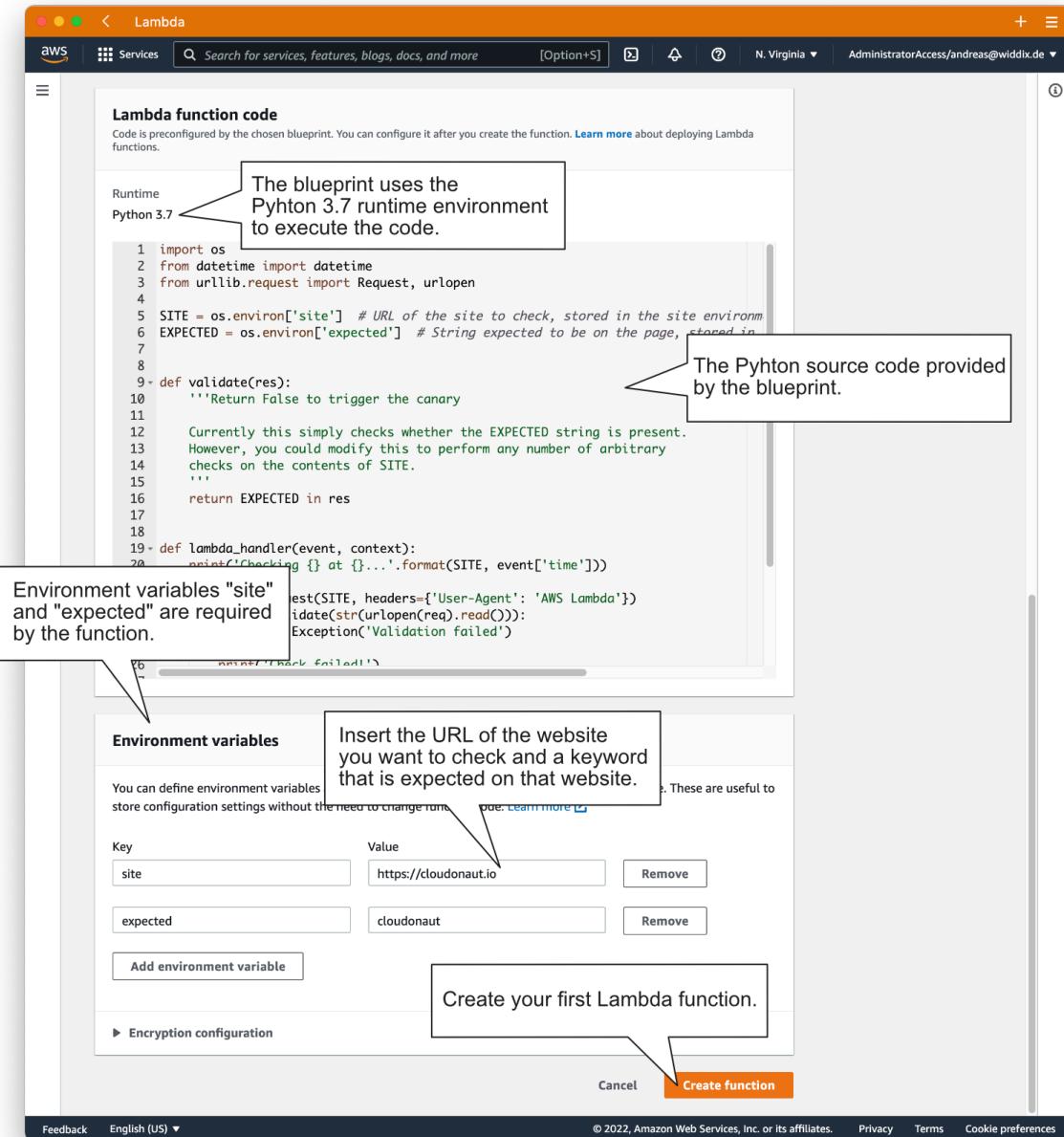


Figure 6.7 The predefined code implementing the website health check and environment variables to pass settings to the Lambda function

Congratulations, you have successfully created a Lambda function. Every 5 minutes, the function is invoked automatically and executes a health check for your website.

You could use this approach to automate recurring tasks, like checking the status of a system, cleaning up unused resources like EBS snapshots, or to send recurring reports.

So far, you used a pre-defined template. Lambda gets much more powerful, when you write your

own code. But before we look into that, you will learn how to monitor your Lambda function and get notified via email whenever the health check fails.

6.2.2 Use CloudWatch to search through your Lambda function's logs

How do you know whether your website health check is working correctly? How do you even know if your Lambda function has been executed? It is time to look at how to monitor a Lambda function. You will learn how to access your Lambda function's log messages first. Afterward, you will create an alarm notifying you if your function fails.

Open the *Monitor* tab in the details view of your Lambda function. You will find a chart illustrating the number of times your function has been invoked. Reload the charts after a few minutes, in case the chart isn't showing any invocations. To go to your Lambda function's logs, click *View logs in CloudWatch* as shown in figure 6.8.

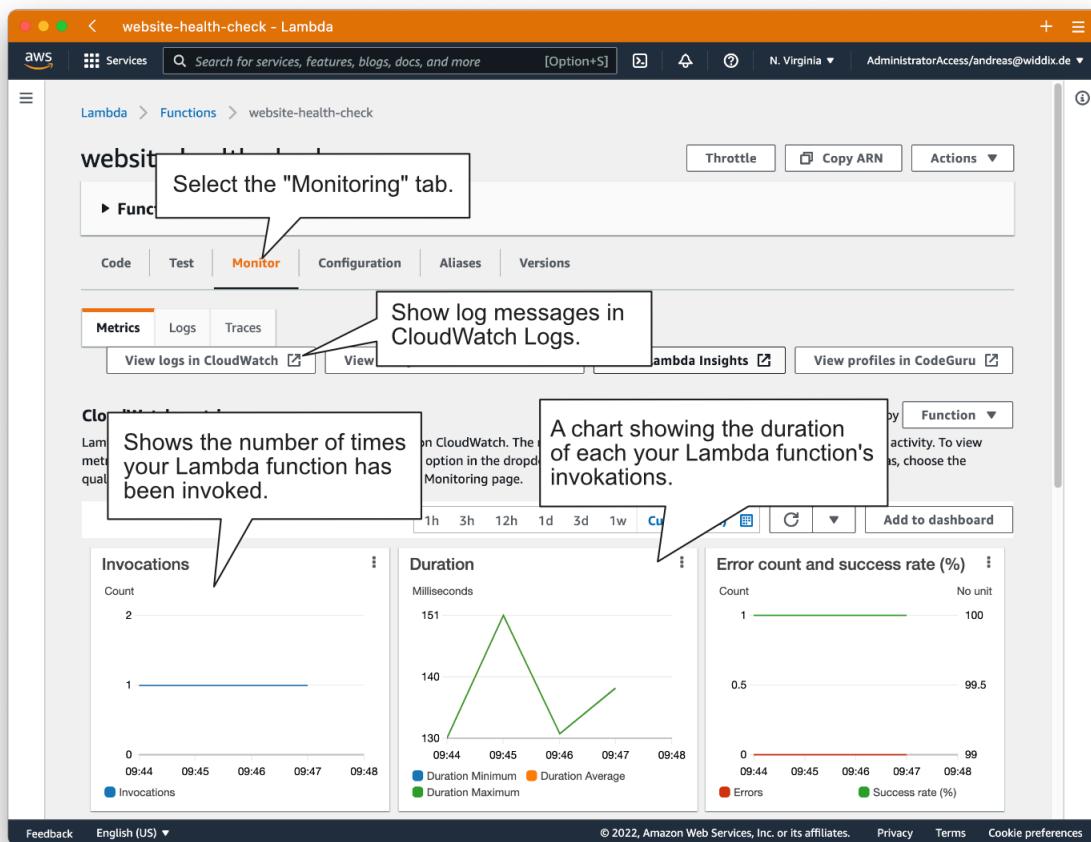


Figure 6.8 Monitoring overview: get insights into your Lambda function's invocations.

By default, your Lambda function sends log messages to CloudWatch. Figure 6.9 shows the log group named `/aws/lambda/website-health-check` that was created automatically and

collects the logs from your function. Typically, a log group contains multiple log streams, allowing the log group to scale. Click *Search log group* to view the log messages from all streams in one view.

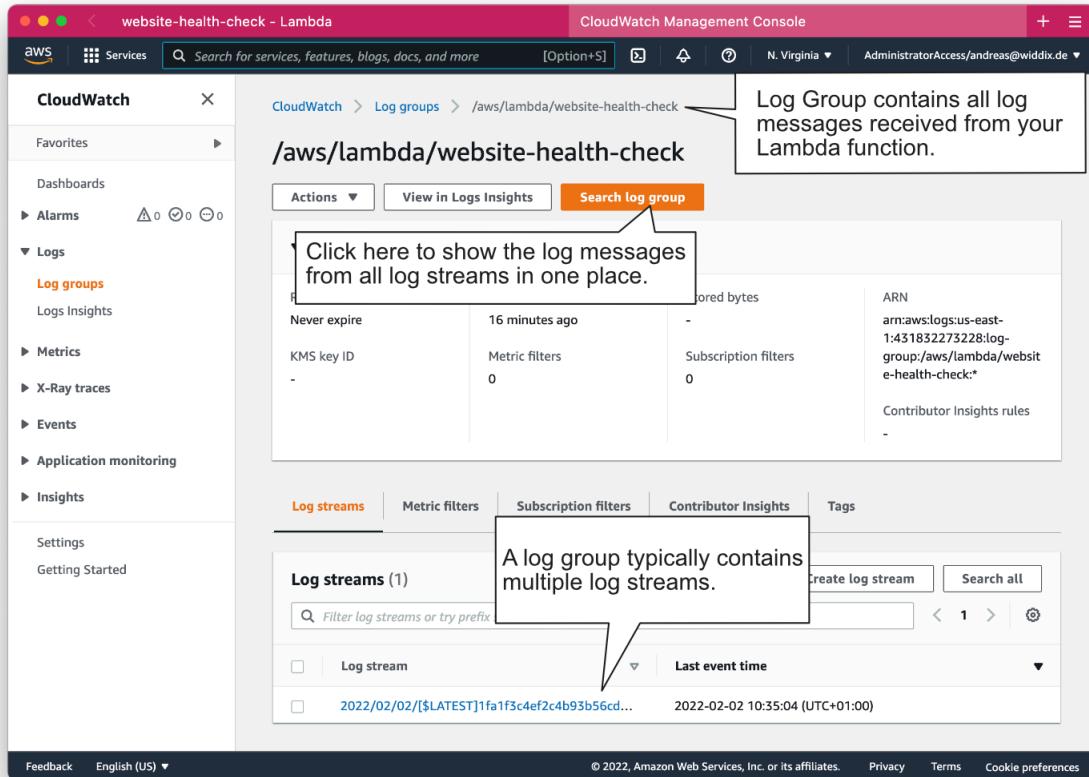


Figure 6.9 A log group collects log messages from a Lambda function stored in multiple log streams.

All log messages are presented in the overview of log streams, as shown in figure 6.10. You should be able to find a log message `Check passed!` indicating that the website health check was executed and passed successfully, for example.

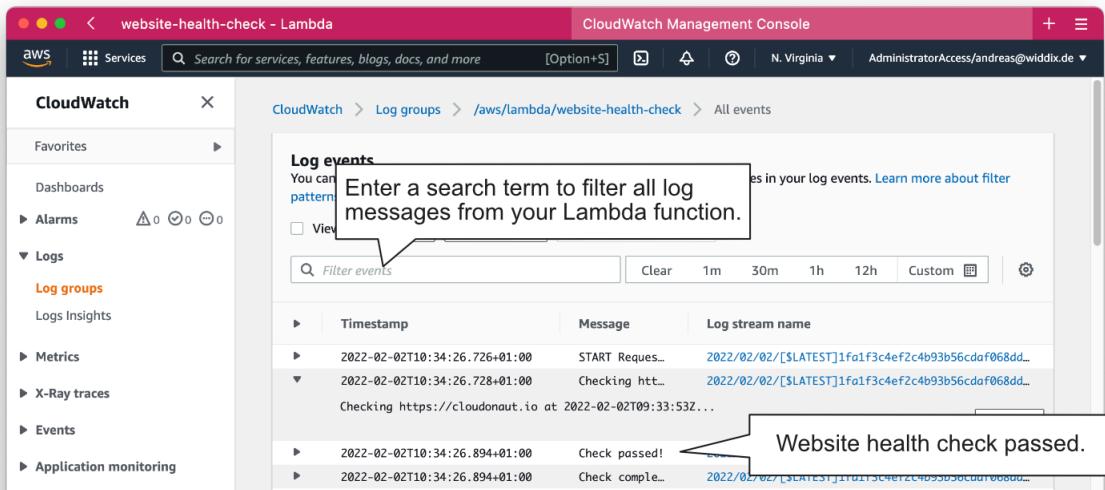


Figure 6.10 CloudWatch shows the log messages of your Lambda function.

The log messages show up after a delay of a few minutes. Reload the table if you are missing any log messages.

Being able to search through log messages in a centralized place is handy when debugging Lambda functions, especially if you are writing your own code. When using Python, you can use `print` statements or use the `logging` module to send log messages to CloudWatch out-of-the-box.

6.2.3 Monitoring a Lambda function with CloudWatch metrics and alarms

The Lambda function now checks the health of your website every 5 minutes. And a log message with the result of each health check is written to CloudWatch. But how do you get notified via email if the health check fails? Each Lambda function publishes metrics to CloudWatch by default. Table 6.2 shows important metrics. Check out docs.aws.amazon.com/lambda/latest/dg/monitoring-metrics.html#monitoring-metrics-types for a complete list of metrics.

Table 6.2 The CloudWatch metrics published by each Lambda function

Name	Description
Invocations	Counts the number of times a function is invoked. Includes successful and failed invocations.
Errors	Counts the number of times a function failed due to errors inside the function. For example, exceptions or timeouts.
Duration	Measures how long the code takes to run, from the time when the code starts executing to when it stops executing.
Throttles	As discussed at the beginning of the chapter, there is a limit for how many copies of your Lambda function are running at one time. This metric counts how many invocations have been throttled due to reaching this limit. Contact AWS support to increase the limit if needed.

Whenever the website health check fails, the Lambda function returns an error, which increases the count of the *Errors* metric. You will create an alarm notifying you via email whenever this

metric counts more than 0 errors. In general, we recommend creating an alarm on the following metrics to monitor your Lambda functions: *Errors* and *Throttles*.

The following steps guide you through creating a CloudWatch alarm to monitor your website health checks. Your Management Console still shows the CloudWatch service. Select *Alarms* from the sub-navigation menu. Did you create a billing alarm in chapter 1? If so, the alarm will be listed here. Next, click *Create alarm* as shown in figure 6.11.

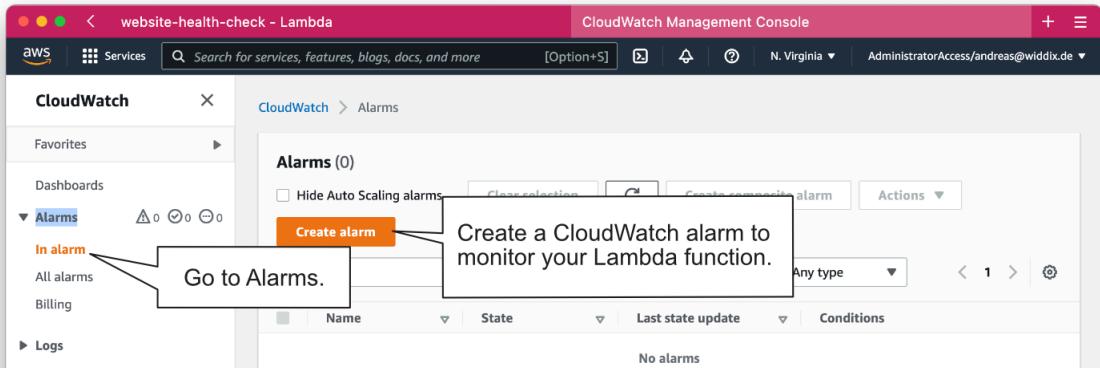


Figure 6.11 Starting the wizard to create a CloudWatch alarm to monitor a Lambda function

The following three steps guide you through the process of selecting the `Error` metric of your `website-health-check` Lambda function as illustrated in figure 6.12.

1. Click *Select metric*.
2. Choose the *Lambda* namespace.
3. Select metrics with dimension *function name*.

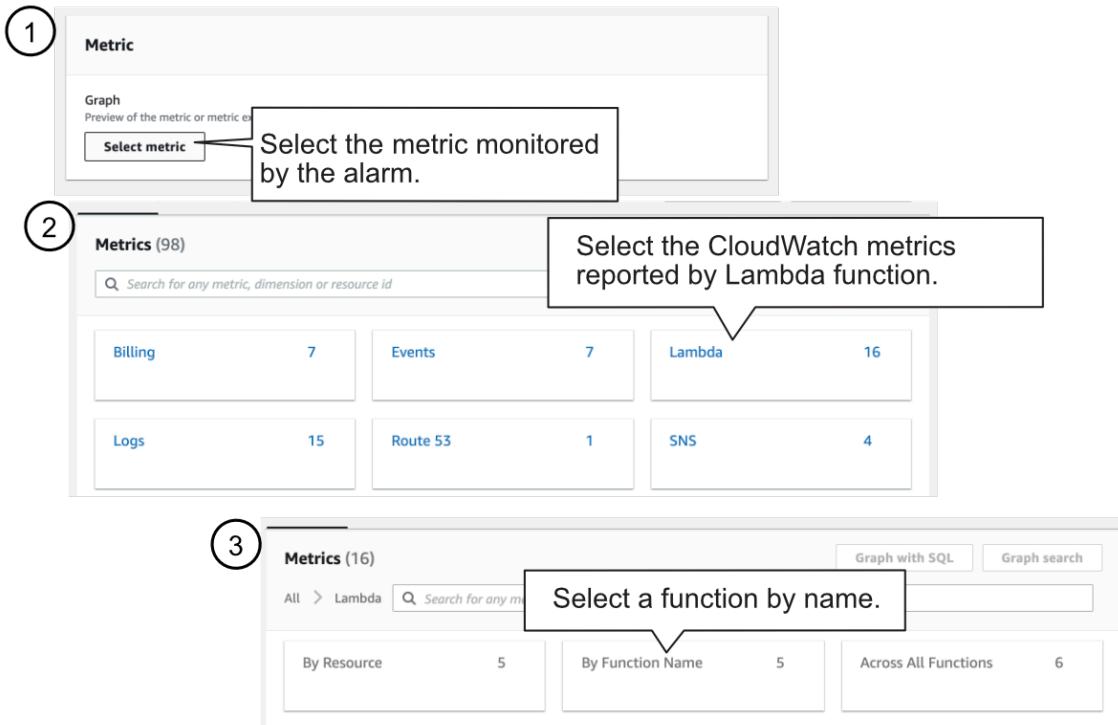


Figure 6.12 Searching the CloudWatch metric to create an alarm

Last but not least, select the `Error` metric belonging to the Lambda function `website-health-check` as shown in figure [6.13](#). Proceed by clicking the *Select metric* button.

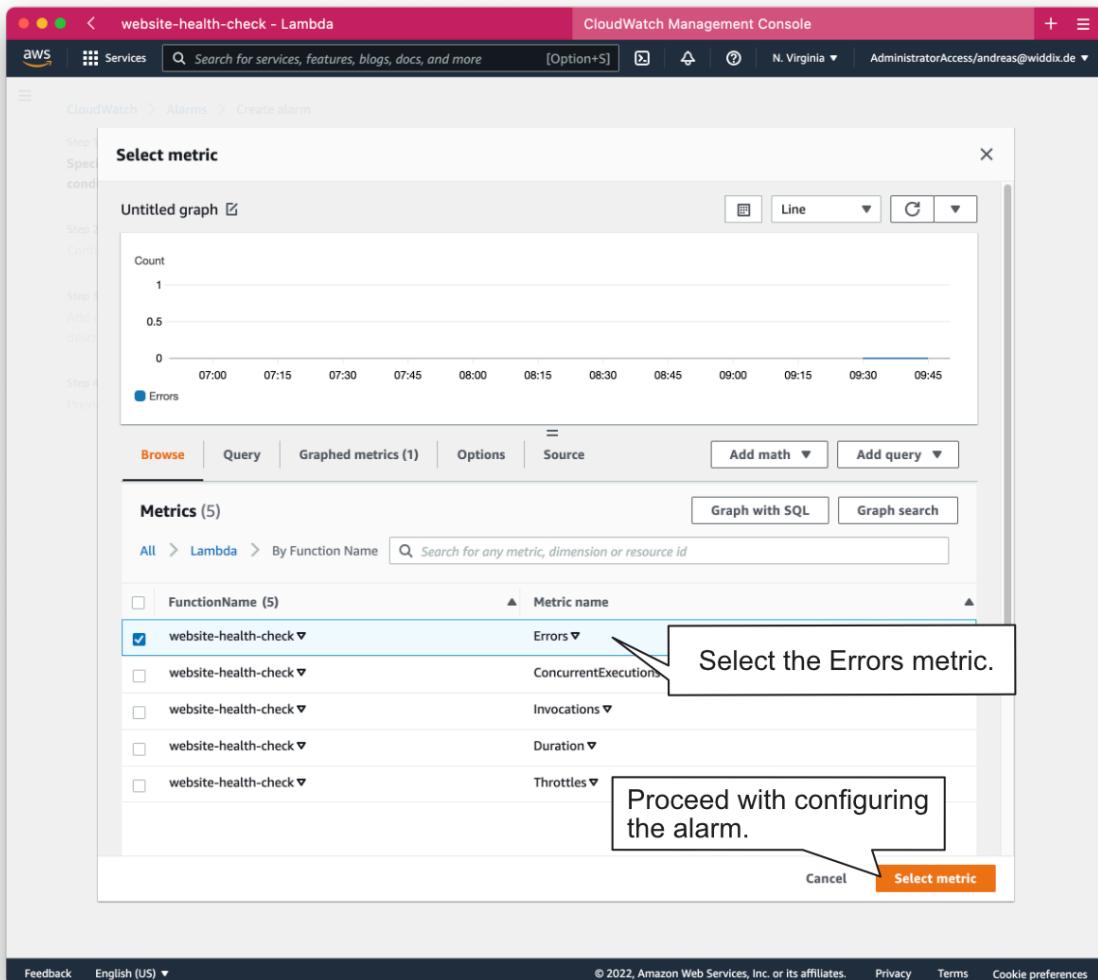


Figure 6.13 Selecting the Error metric of the Lambda function

Next, you need to configure the alarm. To do so, you need to specify the statistic, period and threshold.

1. Choose statistic *Sum* to add up all the errors that occurred during the evaluation period.
2. Define an evaluation period of 5 minutes.
3. Select the static threshold option.
4. Choose the *Greater* operator.
5. Define a threshold of 0.
6. Click the *Next* button to proceed.

In other words, the alarm state will change to `ALARM` when the Lambda function reported any errors during the evaluation period of 5 minutes. Otherwise, the alarm state will be `OK`.

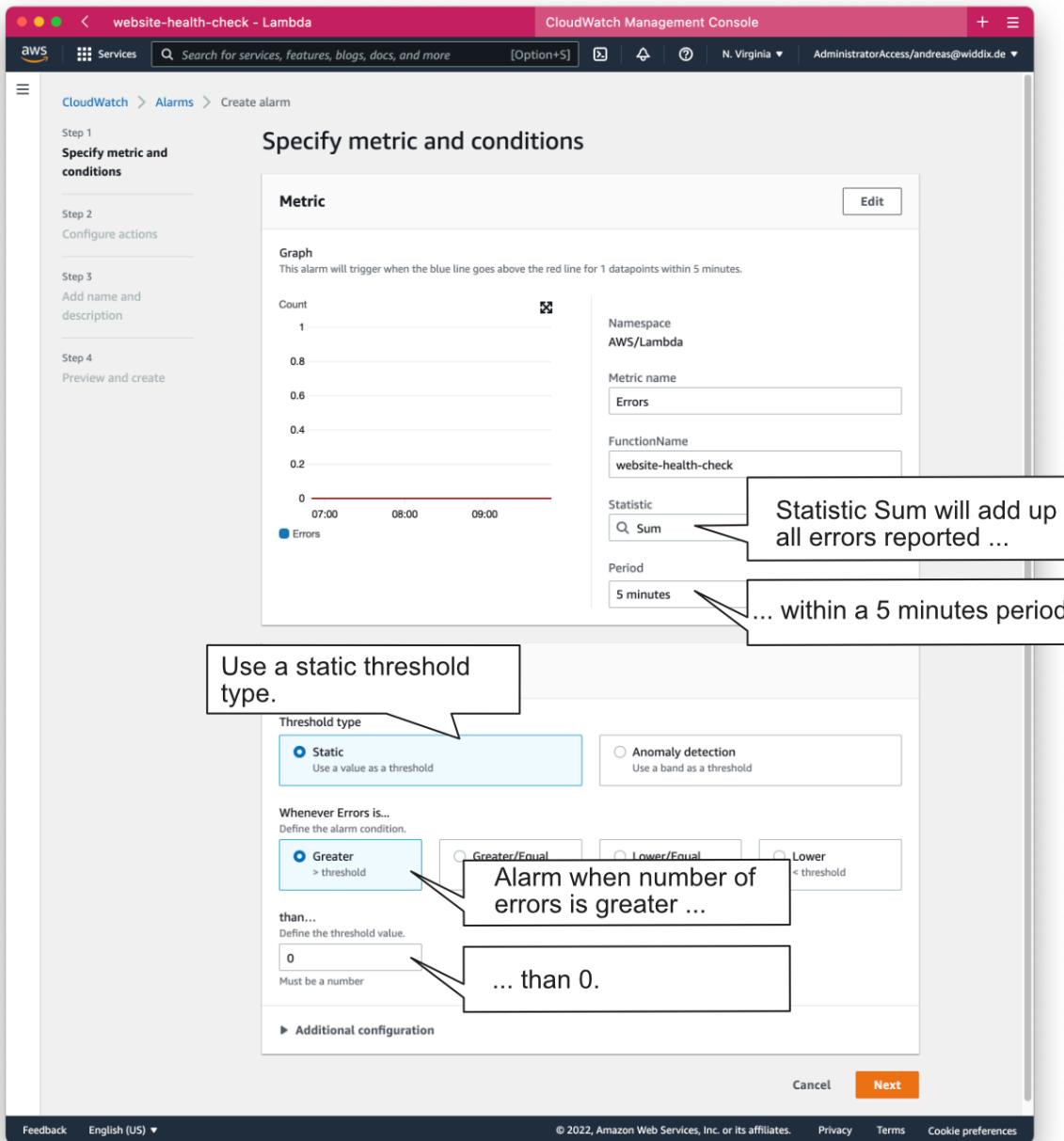


Figure 6.14 Selecting and preparing the metric view for the alarm

The next step, configures the alarm actions. All you need to get notified via e-mail is this.

1. Select *In alarm* as the state trigger.
2. Create a new SNS topic by choosing *Create new topic*.
3. Type in `website-health-check` as the name for the SNS topic.
4. Enter your email address.
5. Click the *Next* button to proceed.

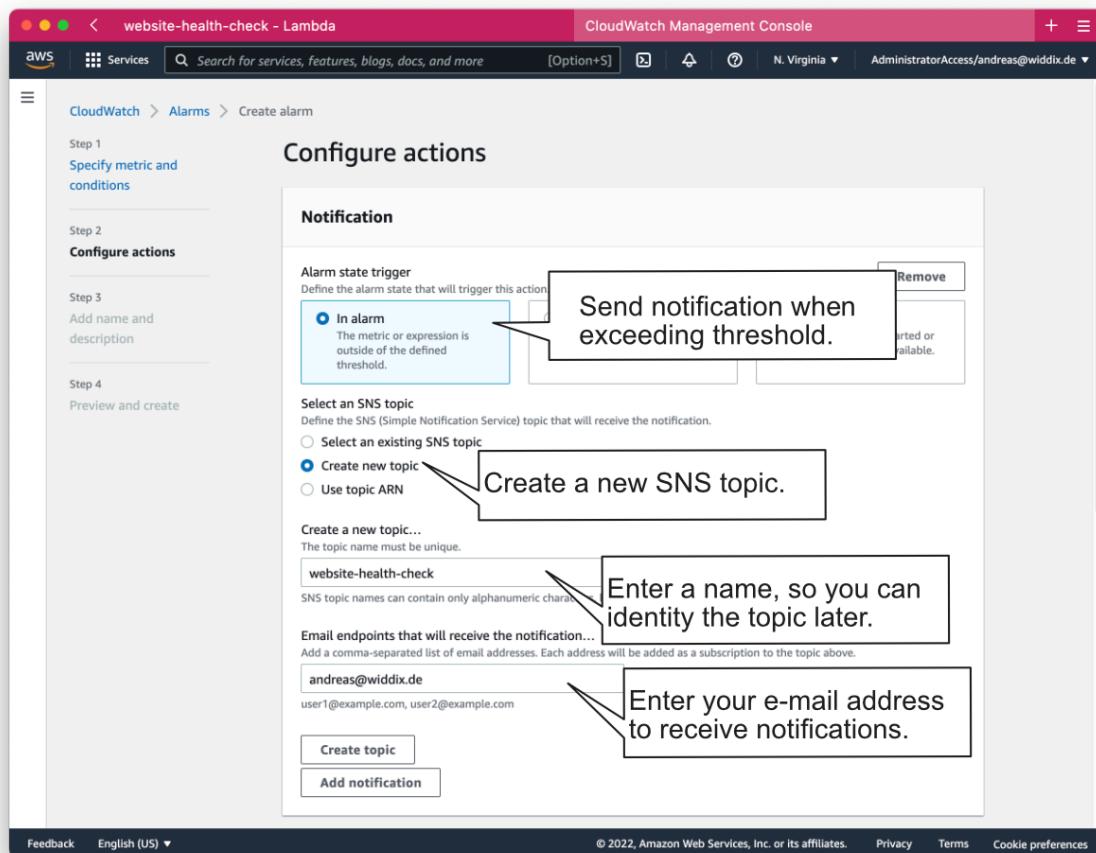


Figure 6.15 Creating an alarm by defining a threshold and defining an alarm action to send notifications via email

In the following step, you need to enter a name for the CloudWatch alarm. Type in `website-health-check-error`. Afterwards, click the *Next* button to proceed.

After reviewing the configuration, click the *Create alarm* button. Shortly after creating the alarm you will receive an email including a confirmation link for SNS. Check your inbox and click the link to confirm your subscription to the notification list.

To test the alarm, go back to the Lambda function and modify the environment variable `expected`. For example, modify the value from `clondonaut` to `FAILURE`. This will cause the health check to fail, as the word `FAILURE` does not appear on the website. It might take up to 15 minutes before you receive a notification about the failed website health check via e-mail.

SIDE BAR**Cleaning up**

cleanup

Open your Management Console and follow these steps to delete all the resources you have created during this section.

- 1. Go to the AWS Lambda service and delete the function named website-health-check.**
- 2. Open the AWS CloudWatch service, select Logs from the sub-navigation, and delete the log group /aws/lambda/website-health-check.**
- 3. Go to the EventBridge service, select Rules from the sub-navigation, and delete the rule website-health-check.**
- 4. Open the CloudWatch service, select Alarms from the sub-navigation, and delete the alarm website-health-check-error.**
- 5. Jump to the AWS IAM service, select Roles from the sub-navigation menu, and delete the role whose name starts with website-health-check-role-.**
- 6. Go to the SNS service, select Topics from the sub-navigation, and delete the rule website-health-check.**

6.2.4 Accessing endpoints within a VPC

As illustrated in figure [6.16](#), Lambda functions run outside your networks defined with VPC by default. However, Lambda functions are connected to the internet and therefore able to access other services. That's exactly what you have been doing when creating a website health check: the Lambda function was sending HTTP requests over the internet.

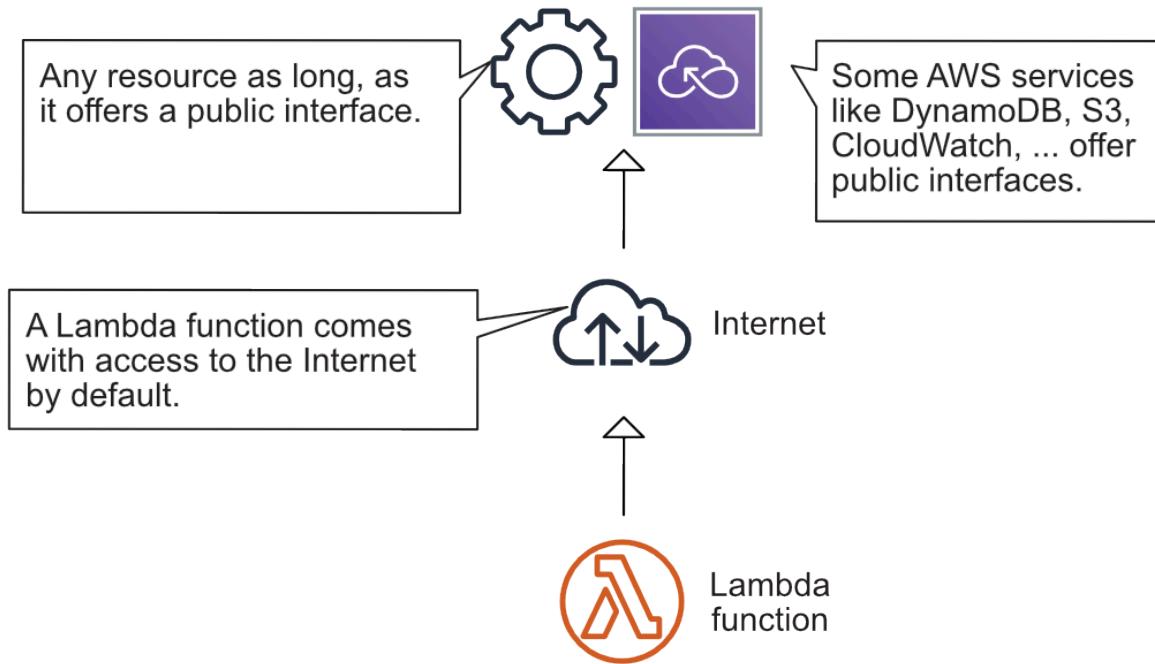


Figure 6.16 By default a Lambda function is connected to the internet and running outside your VPCs.

So what do you do when you have to reach a resource running in a private network within your VPC? For example, if you want to run a health check for an internal website? If you add network interfaces to your Lambda function, the function can access resources within your VPCs as shown in figure [6.17](#).

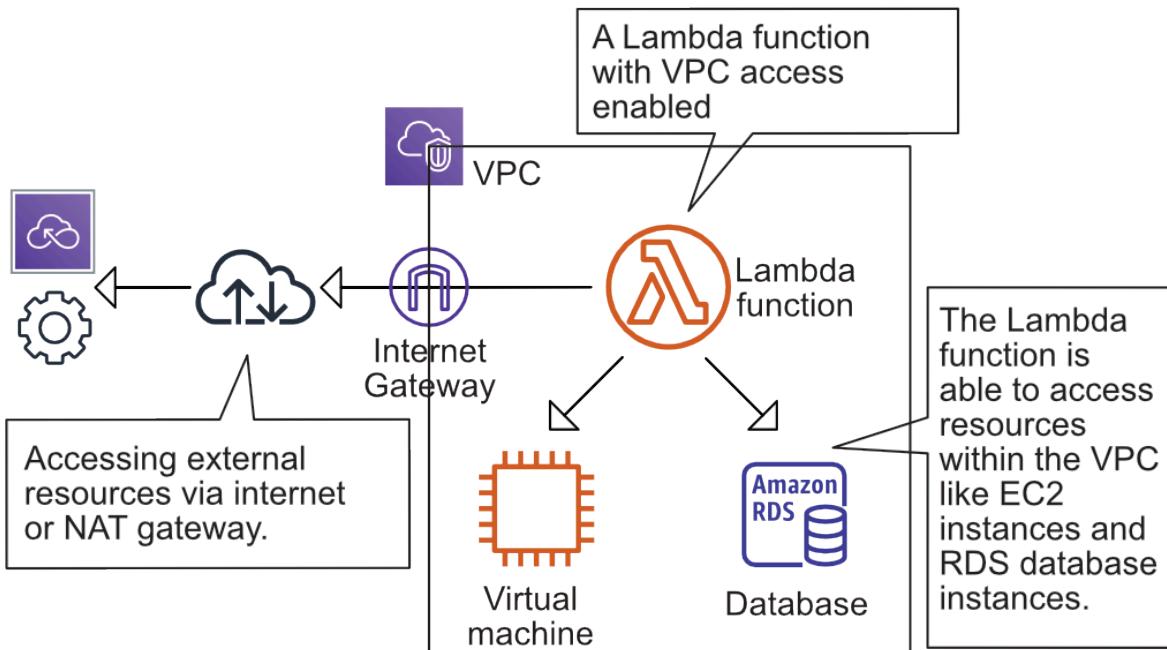


Figure 6.17 Deploying a Lambda function into your VPC allows you to access internal resources (such as database, virtual machines, and so on)

To do so you have to define the VPC, the subnets, as well as security groups for your Lambda function. See “Configuring a Lambda Function to Access Resources in an Amazon VPC” at docs.aws.amazon.com/lambda/latest/dg/vpc.html for more details. We have been using the ability to access resources within a VPC to access databases in various projects.

We do recommend to connect a Lambda function to a VPC only when absolutely necessary as it introduces additional complexity. However, being able to connect with resources within your private networks is very interesting, especially when integrating with legacy systems.

6.3 Adding a tag containing the owner of an EC2 instance automatically

After using one of AWS’s predefined blueprints to create a Lambda function, you will implement a Lambda function from scratch in this section. We are strongly focused on setting up your cloud infrastructure in an automated way. That’s why you will learn how to deploy a Lambda function and all its dependencies without needing the Management Console.

Are you working in an AWS account together with your colleagues? Have you ever wondered who launched a certain EC2 instance? Sometimes you need to find out the owner of an EC2 instance for the following reasons:

- Double-checking if it is safe to terminate an unused instance without losing relevant data.
- Reviewing changes to an instance’s configuration with its owner (such as making changes to the firewall configuration).
- Attributing costs to individuals, projects, or departments.
- Restricting access to an instance (for example, so only the owner is allowed to terminate an instance).

Adding a tag that states who owns an instance solves all these use cases. A tag can be added to an EC2 instance or almost any other AWS resource, and consists of a key and a value. Tags can be used to add information to a resource, filter resources, attribute costs to resources, as well as to restrict access. See “Tag your Amazon EC2 resources” at docs.aws.amazon.com/AWSEC2/latest/UserGuide/Using_Tags.html for more details.

It is possible to add tags specifying the owner of an EC2 instance manually. But sooner or later, someone will forget to add the owner tag. There is a better solution for that! You will implement and deploy a Lambda function that adds a tag containing the name of the user who launched an EC2 instance automatically in the following section. But how do you execute a Lambda function every time an EC2 instance is launched, so that you can add the tag?

6.3.1 Event-driven: Subscribing to EventBridge events

EventBridge is an event bus used by AWS, 3rd party vendors, and customers like you, to publish and subscribe to events. In this example, you will create a rule to listen for events from AWS. Whenever something changes in your infrastructure, an event is generated in near real-time:

- CloudTrail emits an event for every call to the AWS API if CloudTrail is enabled in the AWS account and region.
- EC2 emits events whenever the state of an EC2 instances changes (such as when the state changes from pending to running).
- AWS emits an event to notify you of service degradations or downtimes.

Whenever you launch a new EC2 instance, you are sending a call to the AWS API. Subsequently, CloudTrail sends an event to EventBridge. Our goal is to add a tag to every new EC2 instance. Therefore, we are executing a function for every event that indicates the launch of a new EC2 instance. To trigger a Lambda function whenever such an event occurs, you need a rule. As illustrated in figure 6.18, the rule matches incoming events and routes them to a target, a Lambda function in our case.

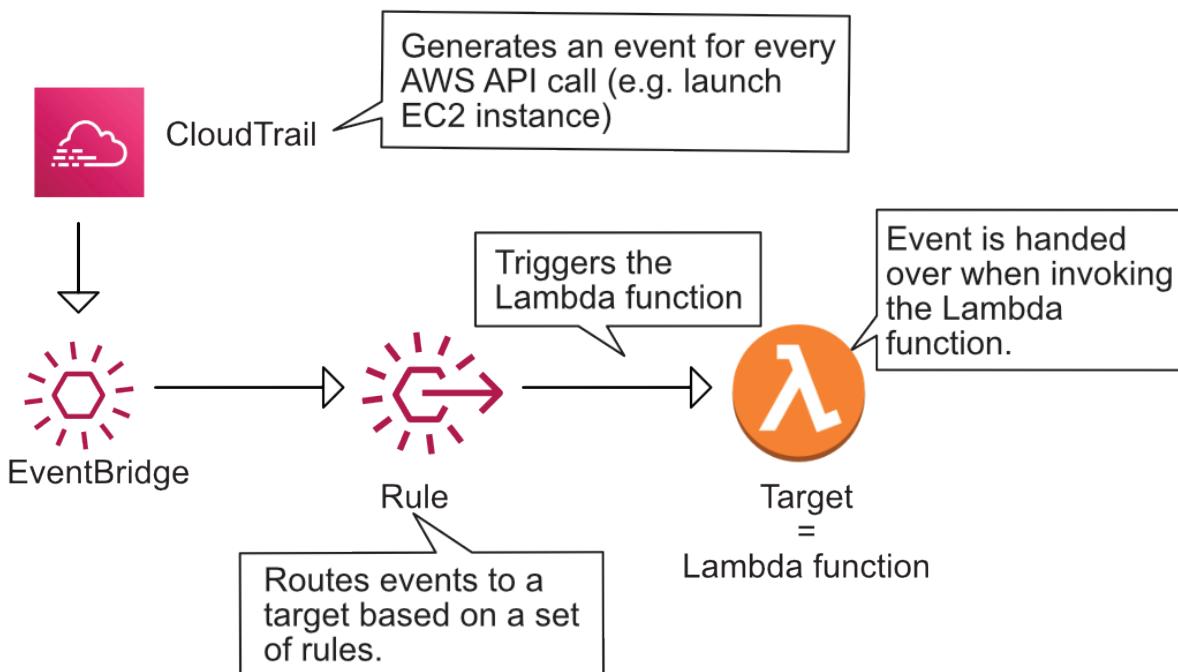


Figure 6.18 CloudTrail generates an event for every AWS API call, a rule routes the event to the Lambda function

Listing 6.1 shows some of the event details generated by CloudTrail whenever someone launches an EC2 instance. For our case, we're interested in the following information:

- `detail-type`—The event has been created by CloudTrail.
- `source`—The EC2 service is the source of the event.

- `eventName`—The event name `RunInstances` indicates that the event was generated because of an AWS API call launching an EC2 instance.
- `userIdentity`—Who called the AWS API to launch an instance?
- `responseElements`—The response from the AWS API when launching an instance. This includes the ID of the launched EC2 instance that we will need to add a tag to the instance later.

Listing 6.1 Event generated by CloudTrail when launching an EC2 instance

```
{
  "version": "0",
  "id": "2db486ef-6775-de10-1472-ecc242928abe",
  "detail-type": "AWS API Call via CloudTrail", ①
  "source": "aws.ec2", ②
  "account": "XXXXXXXXXXXXXX",
  "time": "2022-02-03T11:42:25Z",
  "region": "us-east-1",
  "resources": [],
  "detail": {
    "eventVersion": "1.08",
    "userIdentity": { ③
      "type": "IAMUser",
      "principalId": "XXXXXXXXXXXXXX",
      "arn": "arn:aws:iam::XXXXXXXXXXXX:user/myuser", ④
      "accountId": "XXXXXXXXXXXXXX",
      "accessKeyId": "XXXXXXXXXXXXXX",
      "userName": "myuser"
    },
    "eventTime": "2022-02-03T11:42:25Z",
    "eventSource": "ec2.amazonaws.com",
    "eventName": "RunInstances", ⑤
    "awsRegion": "us-east-1",
    "sourceIPAddress": "109.90.107.17",
    "userAgent": "aws-cli/2.4.14 Python/3.9.10 Darwin/21.2.0
                  [CA]source/arm64 prompt/off command/ec2.run-instances",
    "requestParameters": {
      [...]
    },
    "responseElements": { ⑥
      "requestId": "d52b86c7-5bf8-4d19-86e8-112e59164b21",
      "reservationId": "r-08131583e8311879d",
      "ownerId": "166876438428",
      "groupSet": {},
      "instancesSet": {
        "items": [
          {
            "instanceId": "i-07a3c0d78dc1cb505", ⑦
            "imageId": "ami-01893222c83843146",
            [...]
          }
        ]
      }
    },
    "requestID": "d52b86c7-5bf8-4d19-86e8-112e59164b21",
    "eventID": "8225151b-3a9c-4275-8b37-4a317dfe9ee2",
    "readOnly": false,
    "eventType": "AwsApiCall",
    "managementEvent": true,
    "recipientAccountId": "XXXXXXXXXXXXXX",
    "eventCategory": "Management",
    "tlsDetails": {
      [...]
    }
  }
}
```

- ① CloudTrail generated the event
- ② Someone sent a call to the AWS API affecting the EC2 service.
- ③ Information about the user who launched the EC2 instance
- ④ ID of the user who launched the EC2 instance
- ⑤ Event was generated because a RunInstances call (used to launch an EC2 instance) was processed by the AWS API.
- ⑥ Response of the AWS API when launching the instance
- ⑦ ID of the launched EC2 instance

A rule consists of an event pattern for selecting events, along with a definition of one or multiple targets. The following pattern selects all events from CloudTrail generated by an AWS API call affecting the EC2 service. The pattern matches four attributes from the event described in listing 6.2: `source`, `detail-type`, `eventSource`, and `eventName`.

Listing 6.2 The pattern to filter events from CloudTrail

```
{
  "source": [
    "aws.ec2" ①
  ],
  "detail-type": [
    "AWS API Call via CloudTrail" ②
  ],
  "detail": {
    "eventSource": [
      "ec2.amazonaws.com" ③
    ],
    "eventName": [
      "RunInstances" ④
    ]
  }
}
```

- ① Filter events from the EC2 service.
- ② Filter events from CloudTrail caused by AWS API calls.
- ③ Filter events from the EC2 service.
- ④ Filter events with event name RunInstances, which is the AWS API call to launch an EC2 instance.

Defining filters on other event attributes is possible as well, in case you are planning to write another rule in the future. The rule format stays the same.

When specifying an event pattern, we typically use the following fields, which are included in every event:

- `source`—The namespace of the service which generated the event. See “Amazon Resource Names (ARNs)” at

docs.aws.amazon.com/general/latest/gr/aws-arns-and-namespaces.html#genref-aws-servicename for details.

- `detail-type`—Categorizes the event in more detail.

See “Event Patterns in EventBridge events” at docs.aws.amazon.com/AmazonCloudWatch/latest/events/CloudWatchEventsandEventPatterns.html for more detailed information.

You have now defined the events that will trigger your Lambda function. Next, you will implement the Lambda function.

6.3.2 Implementing the Lambda function in Python

Implementing the Lambda function to tag an EC2 instance with the owner’s user name is simple. You will need to write no more than 10 lines of Python code. The programming model for a Lambda function depends on the programming language you choose. Although we are using Python in our example, you will be able to apply what you’ve learned when implementing a Lambda function in C#/.NET Core, Go, Java, JavaScript/Node.js, Python, or Ruby. As shown in the next listing, your function written in Python needs to implement a well-defined structure.

Listing 6.3 Lambda function written in Python

```
def lambda_handler(event, context):      ①
    # Insert your code                  ②
    return                               ③
```

- ① The name of the Python function, which is referenced by the AWS Lambda as the function handler. The `event` parameter is used to pass the CloudWatch event and the `context` parameter includes runtime information.
- ② It is your job to implement the function.
- ③ Use `return` to end the function execution. It is not useful to hand over a value in this scenario, as the Lambda function is invoked asynchronously by a CloudWatch event.

SIDE BAR

Where is the code located?

As usual, you’ll find the code in the book’s code repository on GitHub: github.com/AWSinAction/code3. Switch to the `chapter06` directory, which includes all files needed for this example.

Time to write some Python code! Listing 6.3 for `lambda_function.py` shows the function, which receives an event from CloudTrail indicating that an EC2 has been launched recently, and adds a tag including the name of the instance’s owner. The AWS SDK for Python 3.9, named `boto3`, is provided out-of-the-box in the Lambda runtime environment for Python 3.9. In this example you are using the AWS SDK to create a tag for the EC2 instance `ec2.create_tags(...)`

. See the Boto 3 Documentation at boto3.readthedocs.io/en/latest/index.html if you are interested in the details of boto3.

Listing 6.4 Lambda function adding a tag to EC2 instance

```
import boto3
ec2 = boto3.client('ec2') ①

def lambda_handler(event, context): ②
    print(event) ③
    if "/" in event['detail']['userIdentity']['arn']: ④
        userName = event['detail']['userIdentity']['arn'].split('/')[1]
    else:
        userName = event['detail']['userIdentity']['arn']
    instanceId = event['detail']['responseElements']['instancesSet'] ⑤
    [CA]['items'][0]['instanceId']
    print("Adding owner tag " + userName + " to instance " + instanceId + ".")
    ec2.create_tags(Resources=[instanceId], ⑥
    [CA]Tags=[{'Key': 'Owner', 'Value': userName}],)
    return
```

- ① Creates an AWS SDK client for EC2
- ② The name of the function used as entry point for the Lambda function
- ③ Logs the incoming event for debugging
- ④ Extracts the user's name from the CloudTrail event
- ⑤ Extracts the instance's DI from the CloudTrail event
- ⑥ Adds a tag to the EC2 instance using the key owner and the user's name as value

After implementing your function in Python, the next step is to deploy the Lambda function with all its dependencies.

6.3.3 Setting up a Lambda function with the Serverless Application Model (SAM)

You have probably noticed that we are huge fans of automating infrastructures with CloudFormation. Using the Management Console is a perfect way to take the first step when learning about a new service on AWS. But leveling up from manually clicking through a web interface to fully automating the deployment of your infrastructure should be your second step.

AWS released the *Serverless Application Model (SAM)* in 2016. SAM provides a framework for serverless applications, extending plain CloudFormation templates to make it easier to deploy Lambda functions.

Listing [6.4](#) shows how to define a Lambda function using SAM and a CloudFormation template.

Listing 6.5 Defining a Lambda function with SAM within a CloudFormation template

```

---
AWSTemplateFormatVersion: '2010-09-09'      ①
Transform: AWS::Serverless-2016-10-31       ②
Description: Adding an owner tag to EC2 instances automatically
Resources:
# [...]
EC2OwnerTagFunction:
  Type: AWS::Serverless::Function          ③
  Properties:
    Handler: lambda_function.lambda_handler ④
    Runtime: python3.9                      ⑤
    Architectures:
      - arm64                            ⑥
    CodeUri: '.'                           ⑦
    Policies:
      - Version: '2012-10-17'
        Statement:
          - Effect: Allow
            Action: 'ec2:CreateTags'
            Resource: '*'
    Events:
      EventBridgeRule:           ⑨
        Type: EventBridgeRule     ⑩
        Properties:
          Pattern:               ⑪
            source:
              - 'aws.ec2'
            detail-type:
              - 'AWS API Call via CloudTrail'
            detail:
              eventSource:
                - 'ec2.amazonaws.com'
              eventName:
                - 'RunInstances'

```

- ① The CloudFormation template version, not the version of your code.
- ② Transforms are used to process your template. We're using the SAM transformation.
- ③ A special resource provided by SAM allows us to define a Lambda function in a simplified way. CloudFormation will generate multiple resources out of this declaration during the transformation phase.
- ④ The handler is a combination your script's filename and Python function name.
- ⑤ Use the Python 3.9 runtime environment.
- ⑥ Choose the ARM architecture for better price-performance.
- ⑦ The current directory shall be bundled, uploaded, and deployed. You will learn more about that soon.
- ⑧ Authorizes the Lambda function to call other AWS services. More on that next.
- ⑨ The definition of the event invoking the Lambda function.
- ⑩ Creates an EventBridge rule.
- ⑪ Configures the filter pattern.

Please note, this example uses CloudTrail which records all the activity within your AWS account. The CloudFormation template creates a trail to store an audit log on S3. That's needed, because the EventBridge rule does not work without an active trail.

6.3.4 Authorizing a Lambda function to use other AWS services with an IAM role

Lambda functions typically interact with other AWS services. For instance, they might write log messages to CloudWatch allowing you to monitor and debug your Lambda function. Or they might create a tag for an EC2 instance, as in the current example. Therefore, calls to the AWS APIs need to be authenticated and authorized. Figure 6.19 shows a Lambda function assuming an IAM role to be able to send authenticated and authorized requests to other AWS services.

Temporary credentials are generated based on the IAM role and injected into each invocation via environment variables (such as `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, `AWS_SESSION_TOKEN`). Those environment variables are used by the AWS SDK to sign requests automatically.

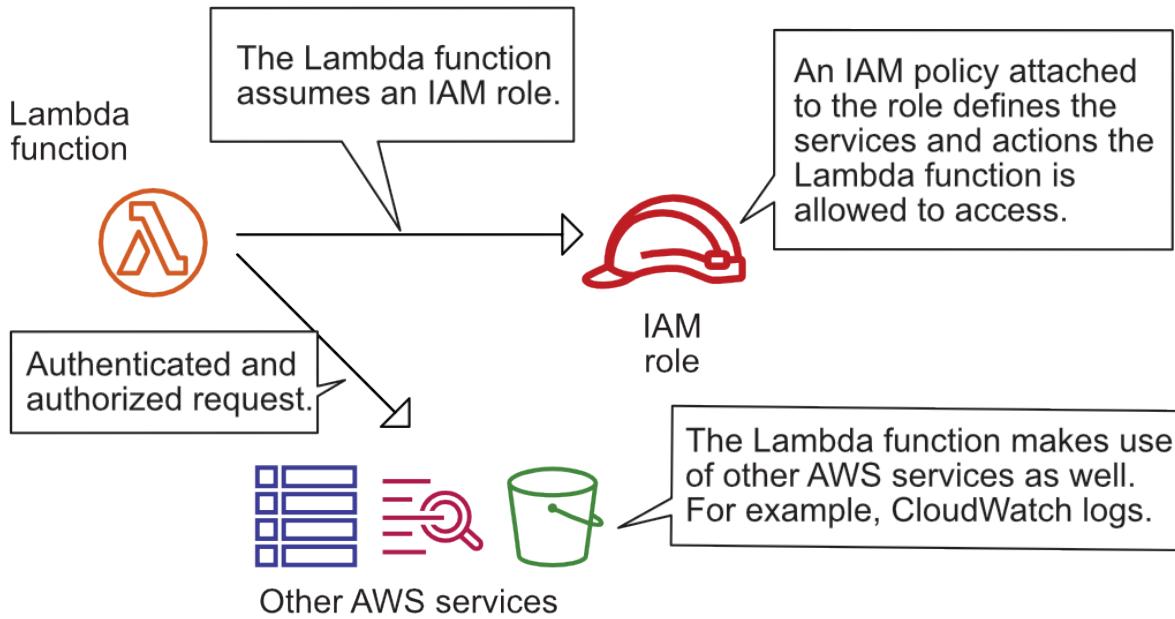


Figure 6.19 A Lambda function assumes an IAM role to authenticate and authorize requests to other AWS services.

You should follow the least-privilege principle: your function should only be allowed to access services and actions that are needed to perform the function's task. You should specify a detailed IAM policy granting access to specific actions and resources.

Listing 6.5 shows an excerpt from the Lambda function's CloudFormation template based on SAM. When using SAM, an IAM role is created for each Lambda function by default. A

managed policy that grants write access to CloudWatch logs is attached to the IAM role by default as well. Doing so allows the Lambda function to write to CloudWatch logs.

So far the Lambda function is not allowed to create a tag for the EC2 instance. You need a custom policy granting access to the `ec2:CreateTags`.

Listing 6.6 A custom policy for adding tags to EC2 instances

```
# [...]
EC2OwnerTagFunction:
  Type: AWS::Serverless::Function
  Properties:
    Handler: lambda_function.lambda_handler
    Runtime: python3.9
    CodeUri: '..'
    Policies:          ①
      - Version: '2012-10-17'
        Statement:
          - Effect: Allow          ②
            Action: 'ec2:CreateTags' ③
            Resource: '*'           ④
    # [...]
```

- ① Defines a custom IAM policy that will be attached to the Lambda function's IAM role
- ② The statement allows ...
- ③ ...creating tags
- ④ ...for all resources.

In case you implement another Lambda function in the future, make sure you create an IAM role granting access to all the services your function needs to access (such as reading objects from S3, writing data to a DynamoDB database, and so on). Revisit section 6.3 if you want to recap the details of IAM.

6.3.5 Deploying a Lambda function with SAM

To deploy a Lambda function, you need to upload the deployment package to S3. The deployment package is a zip file including your code as well as additional modules. Afterward, you need to create and configure the Lambda function as well as all the dependencies (the IAM role, event rule, and so on). Using SAM in combination with the AWS CLI allows you to accomplish both tasks.

First of all you, need to create an S3 bucket to store your deployment packages. Use the following command, replacing `$yourname` with your name to avoid name conflicts with other readers.

```
$ aws s3 mb s3://ec2-owner-tag-$yourname
```

Execute the following command to create a deployment package and upload the package to S3. Please note, the command creates a copy of your template at `output.yaml`, with a reference to the deployment package uploaded to S3. Make sure your working directory is the code directory `chapter06` containing the `template.yaml` and `lambda_function.py` files.

```
$ aws cloudformation package --template-file template.yaml \
[CA] --s3-bucket ec2-owner-tag-$yourname --output-template-file output.yaml
```

By typing in the following command in your terminal, you are deploying the Lambda function. This results in a CloudFormation stack named `ec2-owner-tag`.

```
$ aws cloudformation deploy --stack-name ec2-owner-tag \
[CA] --template-file output.yaml --capabilities CAPABILITY_IAM
```

You are a genius! Your Lambda function is up and running. Launch an EC2 instance and you will find a tag with your user name `myuser` attached after a few minutes.

SIDE BAR

Cleaning up



`cleanup`

If you have launched an EC2 instance to test your Lambda function, don't forget to terminate the instance afterward.

Otherwise it is quite simple to delete the Lambda function and all its dependencies. Just execute the following command in your terminal. Replace `$yourname` with your name.

```
$ CURRENT_ACCOUNT=$(aws sts get-caller-identity --query Account --output text)
$ aws s3 rm --recursive s3://ec2-owner-tag-$CURRENT_ACCOUNT/
$ aws cloudformation delete-stack --stack-name ec2-owner-tag
$ aws s3 rb s3://ec2-owner-tag-$yourname --force
```

6.4 What else can you do with AWS Lambda?

In the last part of the chapter, we would like to share what else is possible with AWS Lambda, starting with Lambda's limitations and insights into the serverless pricing model. We will end with three use cases for serverless applications we have built for our consulting clients.

6.4.1 What are the limitations of AWS Lambda?

Each invocation of your Lambda function needs to complete within a maximum of 900 seconds. This means the problem you are solving with your function needs to be small enough to fit into the 900-second limit. It is probably not possible to download 10 GB of data from S3, process the data, and insert parts of the data into a database within a single invocation of a Lambda function. But even if your use case fits into the 900-second constraint, make sure that it does under all circumstances. Here's a short anecdote from one of our first serverless projects: We built a serverless application that pre-processed analytics data from news sites. The Lambda functions typically processed the data within less than 180 seconds. But when the 2017 U.S. elections came, the volume of the analytics data exploded in a way no one expected. Our Lambda functions were no longer able to complete within 300 seconds - which was the maximum back then. A show stopper for our serverless approach.

AWS Lambda provisions and manages the resources needed to run your function. A new execution context is created in the background every time you deploy a new version of your code, go a long time without any invocations, or when the number of concurrent invocations increases. Starting a new execution context requires AWS Lambda to download your code, initialize a runtime environment, and load your code. This process is called a *cold-start*. Depending on the size of your deployment package, the runtime environment, and your configuration, a cold-start could take from a few milliseconds to a few seconds.

In many scenarios, the increased latency caused by a cold start is not an issue at all. The examples demonstrated in this chapter - a website health check and automated tags for EC2 instances - are not negatively affected by a cold start. To minimize cold-start times, you should keep the size of your deployment package as small as possible, provision additional memory, and use a runtime environment like JavaScript/Node.js or Python.

However, when processing realtime data or user interactions, a cold start is undesirable. For those scenarios, you could enable provisioned concurrency for a Lambda function. With provisioned concurrency, you tell AWS to keep a certain amount of execution contexts warm, even when the Lambda function is not processing any requests. As long as the provisioned concurrency exceeds the required number of execution contexts, you will not experience cold starts. The downside is, you pay USD 0.0000041667 for every provisioned GB-second. No matter if you use the capacity or not. However, you will get a discount on the cost incurred for the actual term of the Lambda function.

Another limitation is the maximum amount of memory you can provision for a Lambda function: 10,240 MB. If your Lambda function uses more memory, its execution will be terminated.

It is also important to know that CPU and networking capacity are allocated to a Lambda function based on the provisioned memory as well. So if you are running computing or

network-intensive work within a Lambda function, increasing the provisioned memory will probably improve performance.

At the same time, the default limit for the maximum size of the compressed deployment package (zip file) is 250 MB. When executing your Lambda function, you can use up to 512 MB non-persistent disk space mounted to `/tmp`.

Look at “AWS Lambda Limits” at docs.aws.amazon.com/lambda/latest/dg/limits.html if you want to learn more about Lambda’s limitations.

6.4.2 Impacts of the serverless pricing model

When launching a virtual machine, you have to pay AWS for every operating hour, billed in second-intervals. You are paying for the machines no matter if you are using the resource they provide. Even when nobody is accessing your website or using your application, you are paying for the virtual machine.

That’s totally different with AWS Lambda. Lambda is billed per request. Costs occur only when someone accesses your website or uses your application. That’s a game changer, especially for applications with uneven access patterns, or for applications that are used rarely. Table 6.3 explains the Lambda pricing model in detail. Please note, when creating a Lambda function, you select the architecture. As usual, the Arm architecture based on AWS Graviton is the cheaper option.

Table 6.3 AWS Lambda pricing model

	Free Tier	x86	Arm
Number of Lambda function invocations	First 1 million requests every month are free	\$0.0000002 USD per request	\$0.0000002 USD per request
Duration billed in 1 ms increments based on the amount of memory you provisioned for your Lambda function	Using the equivalent of 400,000 seconds of a Lambda function with 1 GB is free of charge provisioned memory every month	\$0.0000166667 USD for using 1 GB for one second	\$0.0000133334 USD for using 1 GB for one second

SIDE BAR

Free Tier for AWS Lambda

The Free Tier for AWS Lambda does not expire after 12 months. That’s a huge difference compared to the Free Tier of other AWS services (such as EC2) where you are only eligible for the Free Tier within the first 12 month after creating an AWS account.

Sounds complicated? Figure 6.20 shows an excerpt of an AWS bill. The bill is from November 2017, and belongs to an AWS account we are using to run a chatbot (see marbot.io). Our chatbot implementation is 100% serverless. The Lambda functions were executed 1.2 million times in

November 2017, which results in a charge of \$0.04 USD. All our Lambda functions are configured to provision 1536 MB memory. In total, all our Lambda functions have been running for 216,000 seconds, roundabout 60 hours in November 2017. That's still within the Free Tier of 400,000 seconds with 1 GB provisioned memory every month. So in total we had to pay \$0.04 for using AWS Lambda in November 2017, which allowed us to serve around 400 customers with our chatbot.

This is only a small piece of our AWS bill, as other services used together with AWS Lambda, for example to store data, add more significant costs to our bill.

▼ Lambda		\$0.04
▼ US East (Northern Virginia) Region		\$0.04
AWS Lambda Lambda-GB-Second		\$0.00
AWS Lambda - Compute Free Tier - 400,000 GB-Seconds - US East (Northern Virginia)	331,906.500 seconds	\$0.00
AWS Lambda Request		\$0.04
0.0000000367 USD per AWS Lambda - Requests Free Tier - 1,000,000 Requests - US East (Northern Virginia) (blended price)*	1,209,096 Requests	\$0.04

Figure 6.20 Excerpt from our AWS bill from November 2017 showing costs for AWS Lambda

Don't forget to compare costs between AWS Lambda and EC2. Especially in a high-load scenario with more than 10 million requests per day, using AWS Lambda will probably result in higher costs compared to using EC2. But comparing infrastructure costs is only one part of what you should be looking at. Consider the total cost of ownership (TOC), including costs for managing your virtual machines, performing load and resilience tests, and automating deployments as well.

Our experience has shown that the total cost of ownership is typically lower when running an application on AWS Lambda compared to Amazon EC2.

The last part of the chapter focuses on additional use cases for AWS Lambda besides automating operational tasks, as you have done thus far.

6.4.3 Use case: Web application

A common use case for AWS Lambda is building a back end for a web or mobile application. As illustrated in figure 6.21, an architecture for a serverless web application typically consists of the following building blocks:

- *Amazon API Gateway*—Offers a scalable and secure REST API that accepts HTTPS requests from your web application's front end or mobile application.
- *AWS Lambda*—Lambda functions are triggered by the API gateway. Your Lambda function receives data from the request and returns the data for the response.

- *Object store and NoSQL database*—For storing and querying data, your Lambda functions typically use additional services offering object storage or NoSQL databases, for example.

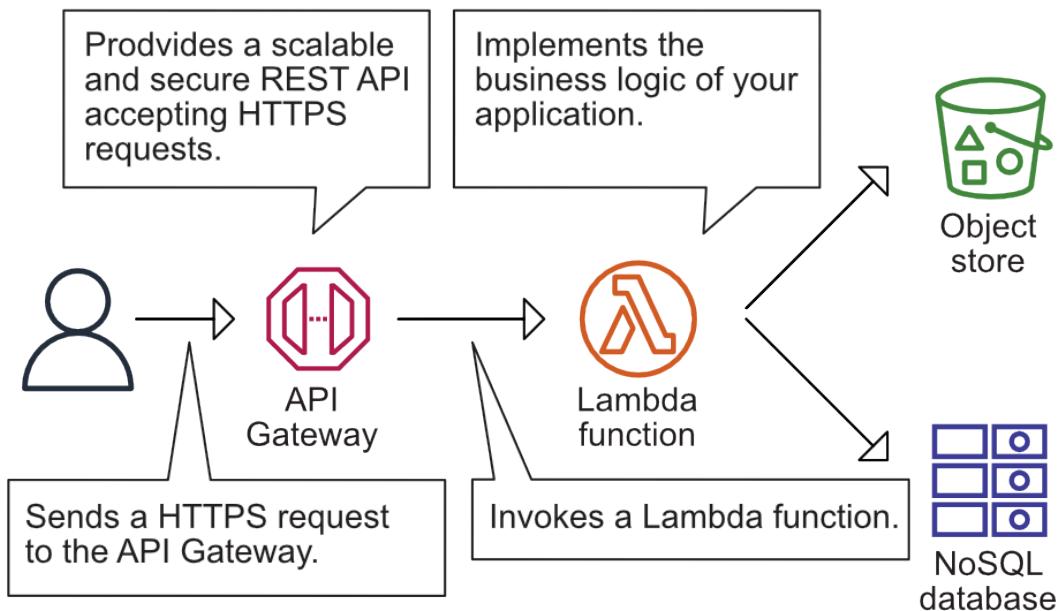


Figure 6.21 A web application build with API Gateway and Lambda

Do you want to get started building web applications based on AWS Lambda? We recommend *AWS Lambda in Action* from Danilo Poccia, (Manning, 2016).

6.4.4 Use case: Data processing

Another popular use case for AWS Lambda is event-driven data processing. Whenever new data is available, an event is generated. The event triggers the data processing needed to extract or transform the data. Figure 6.22 shows an example.

1. The load balancer collects access logs and uploads them to an object store periodically.
2. Whenever an object is created or modified, the object store triggers a Lambda function automatically.
3. The Lambda function downloads the file including the access logs from the object store, and sends the data to an Elasticsearch database to be available for analytics.

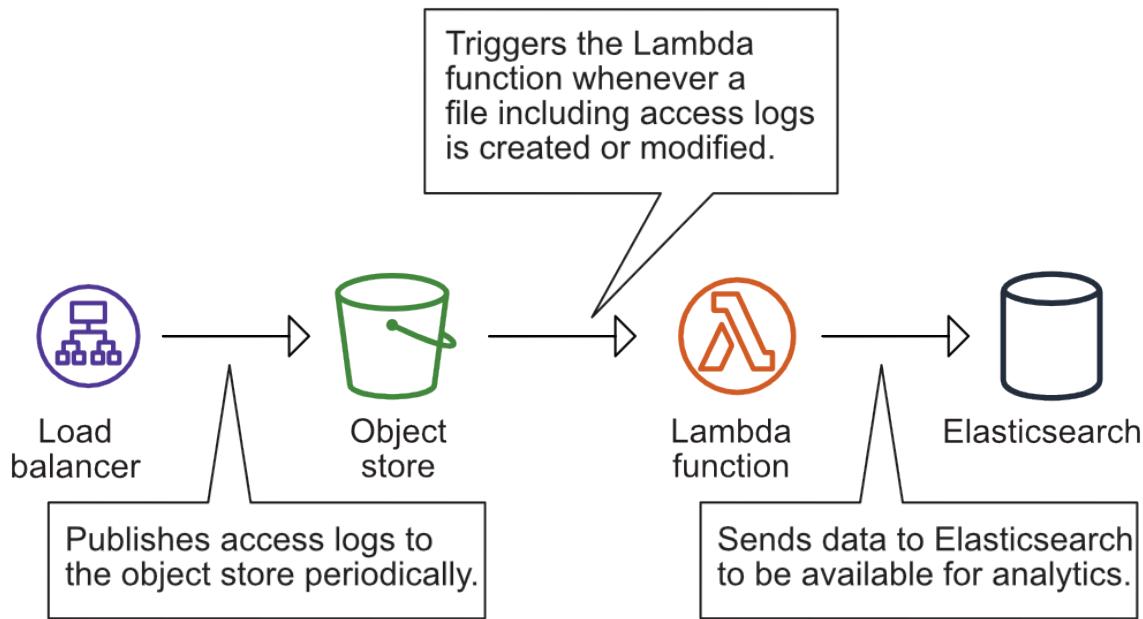


Figure 6.22 Processing access logs from a load balancer with AWS Lambda

We have successfully implemented this scenario in various projects. Keep in mind the maximum execution limit of 900 seconds when implementing data processing jobs with AWS Lambda.

6.4.5 Use case: IoT back end

The AWS IoT service provides building blocks needed to communicate with various devices (things) and build event-driven applications. Figure 6.23 shows an example. Each thing publishes sensor data to a message broker. A rule filters the relevant messages and triggers a Lambda function. The Lambda function processes the event and decides what steps are needed based on business logic you provide.

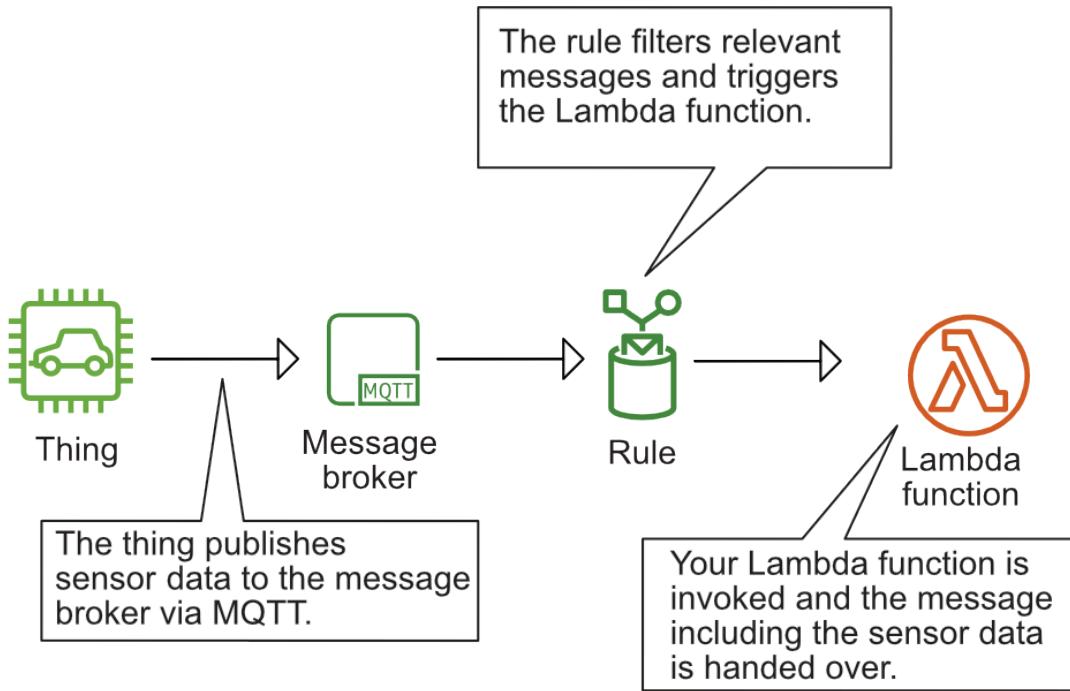


Figure 6.23 Processing events from an IoT device with AWS Lambda

We built a proof-of-concept for collecting sensor data and publishing metrics to a dashboard with AWS IoT and AWS Lambda, for example.

We have gone through three possible use cases for AWS Lambda, but we haven't covered all of them. AWS Lambda is integrated with many other services as well. If you want to learn much more about AWS Lambda, we recommend the following books:

- *AWS Lambda in Action* by Danilo Poccia (Manning, 2016) is an example-driven tutorial that teaches how to build applications using an event-based approach on the back end.
- *Serverless Architectures on AWS* by Peter Sbarski (Manning, 2017) teaches how to build, secure, and manage serverless architectures that can power the most demanding web and mobile apps.

6.5 Summary

- AWS Lambda allows you to run your C#/.NET Core, Go, Java, JavaScript/Node.js, Python and Ruby code within a fully managed, highly available, and scalable environment.
- The Management Console and blueprints offered by AWS help you to get started quickly.
- By using a schedule expression, you can trigger a Lambda function periodically. This is comparable to triggering a script with the help of a cron job.
- CloudWatch is the place to go when it comes to monitoring and debugging your Lambda functions.
- The Serverless Application Model (SAM) enables you to deploy a Lambda function in an automated way with AWS CloudFormation.
- There are many event sources for using Lambda functions in an event-driven way. For example, you can subscribe to events triggered by CloudTrail for every request you send to the AWS API.
- The most important limitation of a Lambda function is the maximum duration of 900 seconds per invocation.
- AWS Lambda can be used to build complex services as well. From typical web applications, to data analytics, and IoT backends, for example.

7 *Storing your objects: S3*

This chapter covers

- Transferring files to S3 using the terminal
- Integrating S3 into your applications with SDKs
- Archiving data at low costs
- Hosting a static website with S3
- Diving into the internals of the S3 object store

Storing data comes with two challenges: ever-increasing volumes of data and ensuring durability. Solving the challenges is hard or even impossible if using disks connected to a single machine. For this reason, this chapter covers a revolutionary approach: a distributed data store consisting of a large number of machines connected over a network. This way, you can store near-unlimited amounts of data by adding additional machines to the distributed data store. And since your data is always stored on more than one machine, you reduce the risk of losing that data dramatically.

You will learn about how to store images, videos, documents, executables, or any other kind of data on Amazon S3 in this chapter. Amazon S3 is a simple-to-use, fully managed distributed data store provided by AWS. Data is managed as objects, so the storage system is called an *object store*. We will show you how to use S3 to back up your data, how to archive data at low costs, how to integrate S3 into your own application for storing user-generated content, as well as how to host static websites on S3.

SIDE BAR

Not all examples are covered by the Free Tier

The examples in this chapter are not all covered by the Free Tier. A special warning message appears when an example incurs costs. As for the other examples, as long as you follow the instructions and don't run them longer than a few days, you won't pay anything.

7.1 What is an object store?

Back in the old days, data was managed in a hierarchy consisting of folders and files. The file was the representation of the data. In an *object store*, data is stored as objects. Each object consists of a globally unique identifier, some metadata, and the data itself, as figure 7.1 illustrates. An object's *globally unique identifier (GUID)* is also known as its *key*; you can address the object from different devices and machines in a distributed system using the GUID.

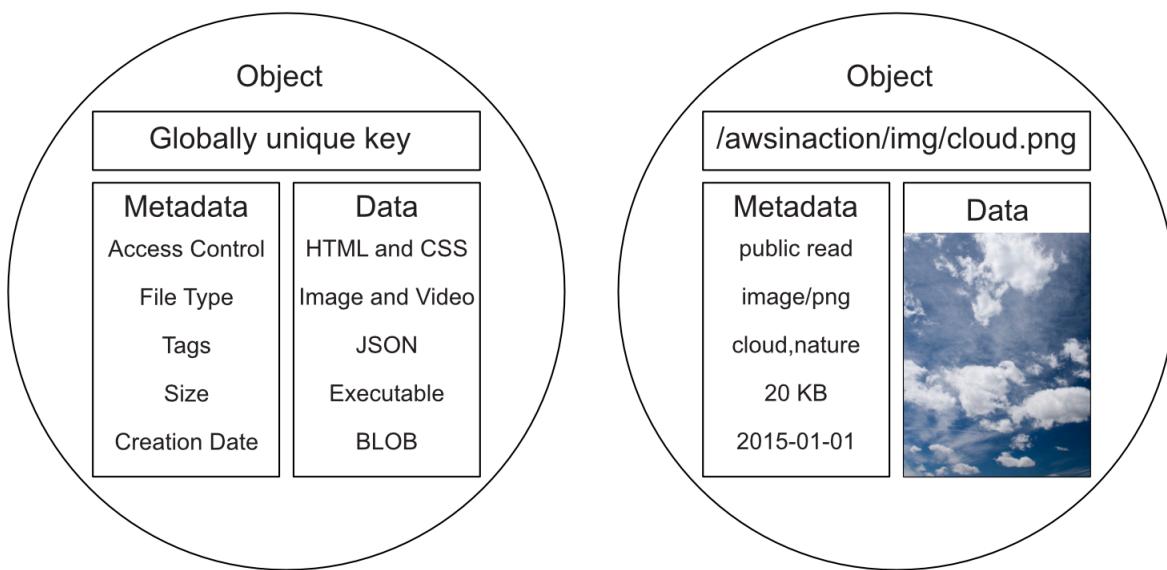


Figure 7.1 Objects stored in an object store have three parts: a unique ID, metadata describing the content, and the content itself (such as an image).

Typical examples for object metadata are:

- Date of last modification
- Object size
- Object's owner
- Object's content type

It is possible to request only an object's metadata without requesting the data itself. This is useful

if you want to list objects and their metadata before accessing a specific object's data.

7.2 Amazon S3

Amazon S3 is a distributed data store, and one of the oldest services provided by AWS. *Amazon S3* is an acronym for *Amazon Simple Storage Service*. It's a typical web service that lets you store and retrieve data organized as objects via an API reachable over HTTPS.

Here are some typical use cases:

- Storing and delivering static website content. For example, our blog clondonaut.io is hosted on S3.
- Backing up data. For example, Andreas backs up his photo library from his computer to S3 using the AWS CLI.
- Storing structured data for analytics, also called a *data lake*. For example, I use S3 to store JSON files containing the results of performance benchmarks.
- Storing and delivering user-generated content. For example, I built a web application—with the help of the AWS SDK—that stores user uploads on S3.

Amazon S3 offers unlimited storage space, and stores your data in a highly available and durable way. You can store any kind of data, such as images, documents, and binaries, as long as the size of a single object doesn't exceed 5 TB. You have to pay for every GB you store in S3, and you also incur costs for every request and for all transferred data. As figure 7.2 shows, you can access S3 via the internet using HTTPS to upload and download objects. To access S3 you can use the Management Console, the CLI, SDKs, or third-party tools.



Figure 7.2 Uploading and downloading an object to S3 via HTTPS

S3 uses *buckets* to group objects. A bucket is a container for objects. It is up to you, to create multiple buckets, each of which has a globally unique name, to separate data for different scenarios. By *unique* we really mean unique—you have to choose a bucket name that isn't used by any other AWS customer in any other region. Figure 7.3 shows the concept.

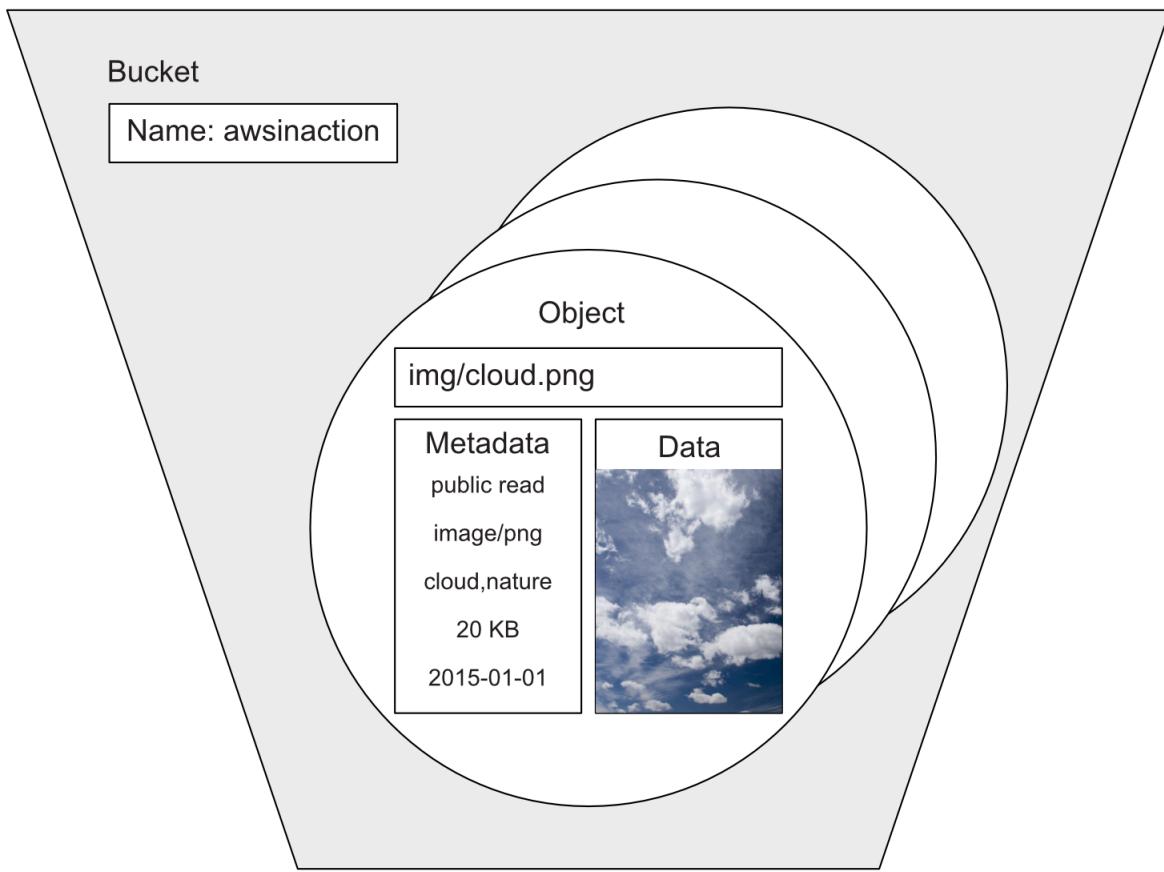


Figure 7.3 S3 uses buckets with a globally unique name to group objects.

You will learn how to upload and download data to S3 using the AWS CLI next.

7.3 Backing up your data on S3 with AWS CLI

Critical data needs to be backed up to avoid loss. Backing up data at an off-site location decreases the risk of losing data even during extreme conditions like natural disaster. But where should you store your backups? S3 allows you to store any data in the form of objects. The AWS object store is a perfect fit for your backup, allowing you to choose a location for your data as well as storing any amount of data with a pay-per-use pricing model.

In this section, you'll learn how to use the AWS CLI to upload data to and download data from S3. This approach isn't limited to off-site backups; you can use it in many other scenarios as well:

- Sharing files with your coworkers or partners, especially when working from different locations.
- Storing and retrieving artifacts needed to provision your virtual machines (such as application binaries, libraries, configuration files, and so on).
- Outsourcing storage capacity to lighten the burden on local storage systems—in

particular, for data that is accessed infrequently.

First you need to create a bucket for your data on S3. As we mentioned earlier, the name of the bucket must be unique among all other S3 buckets, even those in other regions and those of other AWS customers. To find a unique bucket name, it's useful to use a prefix or suffix that includes your company's name or your own name. Run the following command in the terminal, replacing `$yourname` with your name:

```
$ aws s3 mb s3://awsinaction-$yourname
```

Your command should look similar to this one.

```
$ aws s3 mb s3://awsinaction-awittig
```

If your bucket name conflicts with an existing bucket, you'll get an error like this one:

```
[...] An error occurred (BucketAlreadyExists) [...]
```

In this case, you'll need to use a different value for `$yourname`.

Everything is ready to upload your data. Choose a folder you'd like to back up, such as your Desktop folder. Try to choose a folder with a total size less than 1 GB, to avoid long waiting times and exceeding the Free Tier. The following command uploads the data from your local folder to your S3 bucket. Replace `$path` with the path to your folder and `$yourname` with your name. `sync` compares your folder with the `/backup` folder in your S3 bucket and uploads only new or changed files:

```
$ aws s3 sync $path s3://awsinaction-$yourname/backup
```

Your command should look similar to this one.

```
$ aws s3 sync /Users/andreas/Desktop s3://awsinaction-awittig/backup
```

Depending on the size of your folder and the speed of your internet connection, the upload can take some time.

After uploading your folder to your S3 bucket to back it up, you can test the restore process. Execute the following command in your terminal, replacing `$path` with a folder you'd like to use for the restore (don't use the folder you backed up) and `$yourname` with your name. Your Downloads folder would be a good place to test the restore process:

```
$ aws s3 cp --recursive s3://awsinaction-$yourname/backup $path
```

Your command should look similar to this one:

```
$ aws s3 cp --recursive s3://awsinaction-awittig/backup/ \
[CA] /Users/andreas/Downloads/restore
```

Again, depending on the size of your folder and the bandwidth of your internet connection, the download may take a while.

SIDE BAR

Versioning for objects

By default, S3 versioning is disabled for every bucket. Suppose you use the following steps to upload two objects:

1. Add an object with key A and data 1.
2. Add an object with key A and data 2.

If you download the object with key A, you'll download data 2. The old data 1 doesn't exist any more.

You can change this behavior by turning on versioning for a bucket. The following command activates versioning for your bucket. Don't forget to replace \$yourname:

```
$ aws s3api put-bucket-versioning --bucket awsinaction-$yourname \
[CA] --versioning-configuration Status=Enabled
```

If you repeat the previous steps, the first version of object A consisting of data 1 will be accessible even after you add an object with key A and data 2. The following command retrieves all objects and versions:

```
$ aws s3api list-object-versions --bucket awsinaction-$yourname
```

You can now download all versions of an object.

Versioning can be useful for backup and archiving scenarios. Keep in mind that the size of the bucket you'll have to pay for will grow with every new version.

You no longer need to worry about losing data. S3 is designed for 99.99999999% durability of objects over a year. For instance, when storing 100,000,000,000 objects on S3, you will lose only a single object per year on average.

After you've successfully restored your data from the S3 bucket, it's time to clean up. Execute the following command to remove the S3 bucket containing all the objects from your backup. You'll have to replace \$yourname with your name to select the right bucket. rb removes the bucket; the force option deletes every object in the bucket before deleting the bucket itself:

```
$ aws s3 rb --force s3://awsinaction-$yourname
```

Your command should look similar to this one:

```
$ aws s3 rb --force s3://awsinaction-awittig
```

You're finished—you've uploaded and downloaded files to S3 with the help of the CLI.

SIDE BAR**Removing bucket causes BucketNotEmpty error**

If you turn on versioning for your bucket, removing the bucket will cause a `BucketNotEmpty` error. Use the Management Console to delete the bucket in this case:

1. Open the Management Console with your browser.
2. Go to the S3 service using the main navigation menu.
3. Select the bucket you want to delete.
4. Press the Empty button and confirm permanently deleting all objects.
5. Wait until objects and versions have been deleted and click the Exit button.
6. Select the bucket you want to delete.
7. Press the Delete button and confirm deleting the bucket.

7.4 Archiving objects to optimize costs

In the previous section, you learned about backing up your data to S3. Storing 1 TB of data on S3, costs about 23 USD per month. Wouldn't it be nice, to reduce the costs for storing data by 95%? Besides, the default, S3 comes with storage classes designed to archive data for long time spans.

Table [7.1](#) compares the storage class *S3 Standard* with storage classes intended for data archival.

Table 7.1 Differences between storing data with S3 and Glacier

	S3 Standard	S3 Glacier Instant Retrieval	S3 Glacier Flexible Retrieval	S3 Glacier Deep Archive
Storage costs for 1 GB per month in US East (N. Virginia)	0.023 USD	0.004 USD	0.0036 USD	0.00099 USD
Costs for 1,000 write requests	0.005 USD	0.02 USD	0.03 USD	0.05 USD
Costs for retrieving data	Low	High	High	Very High
Accessibility	Milliseconds	Milliseconds	1-5 minutes / 3-5 hours / 5-12 hours	12 hours / 48 hours
Durability objective	99.999999999%	99.999999999%	99.999999999%	99.999999999%
Availability objective	99.99%	99.9%	99.99%	99.99%

The potential savings for storage costs are enormous. So what's the catch?

First, accessing data stored on S3 by using the storage classes *S3 Glacier Instant Retrieval*, *S3 Glacier Flexible Retrieval*, and *S3 Glacier Deep Archive* is expensive. Let's assume, you are

storing 1 TB of data on S3 and decided to use storage type *S3 Glacier Deep Archive*. It will cost you about 120 USD to restore 1 TB of data stored when stored in 1,000 files.

Second, fetching data from *S3 Glacier Flexible Retrieval* and *S3 Glacier Deep Archive* takes something between 1 minute and 48 hours depending on the storage class and retrieval option.

Using the following example, we would like to explain what it means not to be able to access archived data immediately. Let's say you want to archive a document for five years. You do not expect to access the document more than 5 times during this period.

SIDE BAR
Example not covered by Free Tier

The following example is not covered by the Free Tier. Archiving and restoring data as shown in the example will cost you less than \$1. You will find information on how to delete all resources at the end of the section. Therefore, we recommend to complete the section within a few days.

Start by creating an S3 bucket, that you will use to archive documents. Replace `$yourname` with your name to get a unique bucket name.

```
$ aws s3 mb s3://awsinaction-archive-$yourname
```

Next, copy a document from your local machine to S3. The `--storage-class` parameter overrides the default storage class with `GLACIER` which maps to the *S3 Glacier Flexible Retrieval* storage class. Replace `$path` with the path to a document, and `$yourname` with your name. Note down, the key of the object.

```
$ aws s3 cp --storage-class GLACIER $path \
[CA] s3://awsinaction-archive-$yourname/
```

For instance, I run the following command.

```
$ aws s3 cp --storage-class GLACIER \
[CA] /Users/andreas/Desktop/taxstatement-2022-07-01.pdf \
[CA] s3://awsinaction-archive-awittig/
```

The key point is that you can't download the object. Replace `$objectkey` with the object's key that you noted down after uploading the document, and `$path` with the *Downloads* folder on your local machine.

```
$ aws s3 cp s3://awsinaction-archive-$yourname/$objectkey $path
```

For example, I'm getting the following error when trying to do download my document `taxstatement-2022-07-01.pdf`.

```
$ aws s3 cp s3://awsinaction-archive-awittig/taxstatement-2022-07-01.pdf \
[CA] ~/Downloads
warning: Skipping file s3://awsinaction-archive-awittig/
[CA]taxstatement-2022-07-01.pdf. Object is of storage class GLACIER.
[CA]Unable to perform download operations on GLACIER objects. You must
[CA]restore the object to be able to perform the operation.
```

As mentioned in the error message, you need to restore the object before downloading it. By default, doing so will take 3 to 5 hours, that's why we will pay a little extra -just a few cents- for expedited retrieval. Execute the following command after replacing \$yourname with your name, and \$objectkey with the object's key.

```
$ aws s3api restore-object --bucket awsinaction-archive-$yourname \
[CA] --key $objectkey \
[CA] --restore-request Days=1,,GlacierJobParameters={"Tier":"Expedited"}
```

That's resulting in the following command in my scenario.

```
$ aws s3api restore-object --bucket awsinaction-archive-awittig \
[CA] --key taxstatement-2022-07-01.pdf
[CA] --restore-request Days=1,,GlacierJobParameters={"Tier":"Expedited"}
```

As you are using expedited retrieval, you need to wait 1 to 5 minutes for the object to become available for download. Use the following command to check the status of the object and its retrieval. Don't forget to replace \$yourname with your name, and \$objectkey with the object's key.

```
$ aws s3api head-object --bucket awsinaction-archive-$yourname \
[CA] --key $objectkey
{
  "AcceptRanges": "bytes",
  "Expiration": "expiry-date=\\"Wed, 12 Jul 2023 ...\\", rule-id=\\"...\\",
  "Restore": "ongoing-request=\\"true\\\"", ①
  "LastModified": "2022-07-11T09:26:12+00:00",
  "ContentLength": 112,
  "ETag": "\\"c25faldf1968993d8e647c9dcd352d39\\",
  "ContentType": "binary/octet-stream",
  "Metadata": {},
  "StorageClass": "GLACIER"
}
```

- ① Restoring the object is still ongoing.

Repeat fetching the status of the object until ongoing-request flips to false.

```
{
  "AcceptRanges": "bytes",
  "Expiration": "expiry-date=\\"Wed, 12 Jul 2023 ...\\", rule-id=\\"...\\",
  "Restore": "ongoing-request=\\"false\\\"", expiry-date=\\"...\\", ①
  "LastModified": "2022-07-11T09:26:12+00:00",
  "ContentLength": 112,
  "ETag": "\\"c25faldf1968993d8e647c9dcd352d39\\",
  "ContentType": "binary/octet-stream",
  "Metadata": {},
  "StorageClass": "GLACIER"
}
```

- ① The restore finished, no ongoing restore requests.

After restoring the object, you are now able to download the document. Replace `$objectkey` with the object's key that you noted down after uploading the document, and `$path` with the *Downloads* folder on your local machine.

```
$ aws s3 cp s3://awsinaction-archive-$yourname/$objectkey $path
```

In summary, the Glacier storage types are intended for archiving data that you need to access seldom which means every few months or years. For example, we are using the *S3 Glacier Deep Archive* to store a remote backup of our MacBooks. As we store another backup of our data on an external hard drive, chances that we need to restore data from S3 are very low.

SIDE BAR

Cleaning up



`fig cleanup`

Execute the following command to remove the S3 bucket containing all the objects from your backup. You'll have to replace `$yourname` with your name to select the right bucket. `rb` removes the bucket; the `force` option deletes every object in the bucket before deleting the bucket itself:

```
$ aws s3 rb --force s3://awsinaction-archive-$yourname
```

You've learned how to use S3 with the help of the CLI. We'll show you how to integrate S3 into your applications with the help of SDKs in the next section.

7.5 Storing objects programmatically

S3 is accessible using an API via HTTPS. This enables you to integrate S3 into your applications by making requests to the API programmatically. Doing so allows your applications to benefit from a scalable and highly available data store. AWS offers free SDKs for common programming languages like C++, Go, Java, JavaScript, .NET, PHP, Python, and Ruby. You can execute the following operations using an SDK directly from your application:

- Listing buckets and their objects.
- Creating, removing, updating, and deleting (CRUD) objects and buckets.
- Managing access to objects.

Here are examples of how you can integrate S3 into your application:

- *Allow a user to upload a profile picture.* Store the image in S3, and make it publicly accessible. Integrate the image into your website via HTTPS.

- *Generate monthly reports (such as PDFs), and make them accessible to users.* Create the documents and upload them to S3. If users want to download documents, fetch them from S3.
- *Share data between applications.* You can access documents from different applications. For example, application A can write an object with the latest information about sales, and application B can download the document and analyze the data.

Integrating S3 into an application is one way to implement the concept of a *stateless server*. We'll show you how to integrate S3 into your application by diving into a simple web application called Simple S3 Gallery next. This web application is built on top of Node.js and uses the AWS SDK for JavaScript and Node.js. You can easily transfer what you learn from this example to SDKs for other programming languages; the concepts are the same.

SIDE BAR

Installing and getting started with Node.js

Node.js is a platform for executing JavaScript in an event-driven environment so you can easily build network applications. To install Node.js, visit nodejs.org and download the package that fits your OS. All examples in this book are tested with Node.js 14.

After Node.js is installed, you can verify that everything works by typing `node --version` into your terminal. Your terminal should respond with something similar to `v14.*`. Now you're ready to run JavaScript examples like the Simple S3 Gallery.

Do you want to get started with Node.js? We recommend Node.js in Action (2nd edition) from Alex Young, et al. (Manning, 2017), or the video course Node.js in Motion from P.J. Evans, (Manning, 2018).

Next, we will dive into a simple web application called the *Simple S3 Gallery*,. The gallary allows you to upload images to S3 and displays all the images you've already uploaded. Figure [7.4](#) shows Simple S3 Gallery in action. Let's set up S3 to start your own gallery.

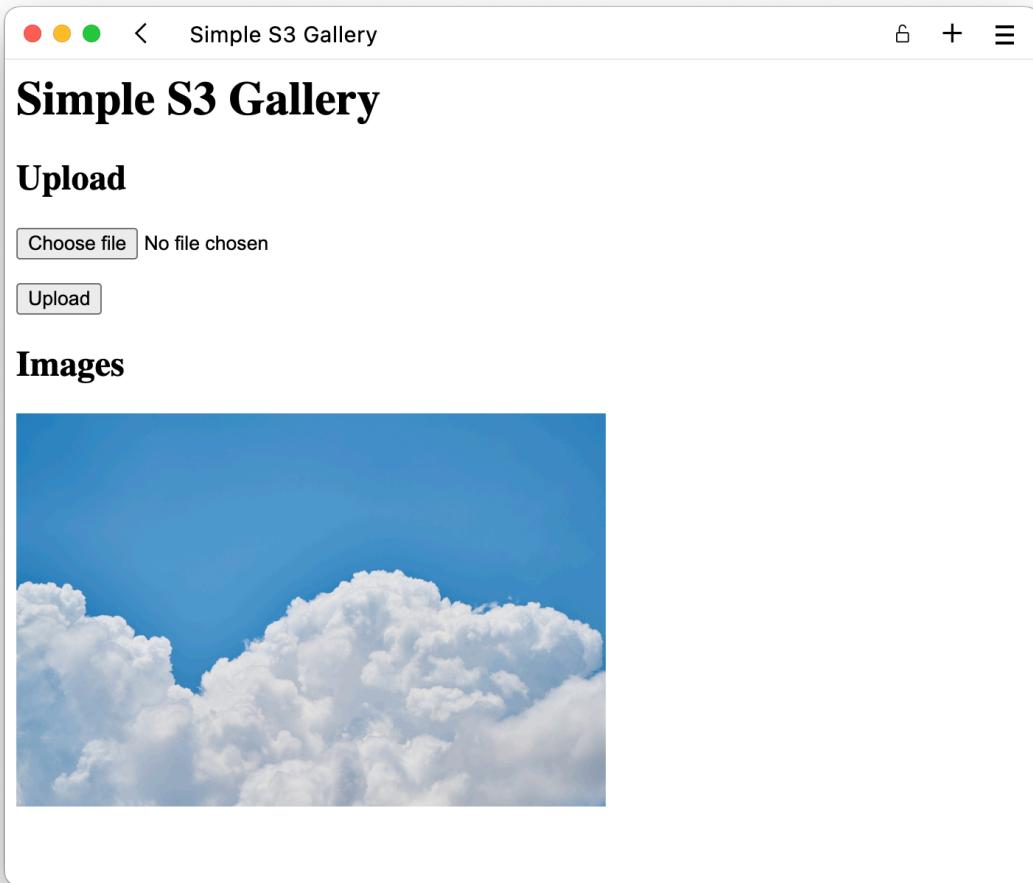


Figure 7.4 The Simple S3 Gallery app lets you upload images to an S3 bucket and then download them from the bucket for display.

7.5.1 Setting up an S3 bucket

To begin, you need to set up an empty bucket. Execute the following command, replacing `$yourname` with your name or nickname:

```
$ aws s3 mb s3://awsinaction-sdk-$yourname
```

Your bucket is now ready to go. Installing the web application is the next step.

7.5.2 Installing a web application that uses S3

You can find the Simple S3 Gallery application in `/chapter07/gallery/` in the book's code folder. Switch to that directory, and run `npm install` in your terminal to install all needed dependencies.

To start the web application, run the following command. Replace `$yourname` with your name;

the name of the S3 bucket is then passed to the web application:

```
$ node server.js awsinaction-sdk-$yourname
```

SIDE BAR
Where is the code located?

You can find all the code in the book's code repository on GitHub: github.com/AWSInAction/code3. You can download a snapshot of the repository at github.com/AWSInAction/code3/archive/main.zip.

After you start the server, you can open the gallery application. To do so, open localhost:8080 with your browser. Try uploading a few images.

7.5.3 Reviewing code access S3 with SDK

You've uploaded images to the *Simple S3 Gallery* and displayed images from S3. Inspecting parts of the code will help you to understand how you can integrate S3 into your own applications. It's not a problem if you don't follow all the details of the programming language (JavaScript) and the Node.js platform; we just want you to get an idea of how to use S3 via SDKs.

UPLOADING AN IMAGE TO S3

You can upload an image to S3 with the SDK's `putObject()` function. Your application will connect to the S3 service and transfer the image via HTTPS. The listing 7.1 shows how to do so.

Listing 7.1 Uploading an image with the AWS SDK for S3

```
const AWS = require('aws-sdk');          ①
const uuid = require('uuid');

const s3 = new AWS.S3({    ②
  'region': 'us-east-1'
});

const bucket = process.argv[2];

async function uploadImage(image, response) {
  try {
    await s3.putObject({    ③
      Body: image,        ④
      Bucket: bucket,    ⑤
      Key: uuid.v4(),    ⑥
      ACL: 'public-read', ⑦
      ContentLength: image.byteCount, ⑧
      ContentType: image.headers['content-type'] ⑨
    }).promise();
    response.redirect('/');
  } catch (err) {    ⑩
    console.error(err);
    response.status(500);
    response.send('Internal server error.');
  }
}
```

- ① Load the AWS SDK.
- ② Instantiate s3 client with additional config.
- ③ Uploads the image to S3
- ④ Image content
- ⑤ Name of the bucket
- ⑥ Generates a unique key for the object
- ⑦ Allows everybody to read the image from bucket
- ⑧ Size of image in bytes
- ⑨ Content type of the object (image/png)
- ⑩ Catching errors
- ⑪ Return error with HTTP status code 500

The AWS SDK takes care of sending all the necessary HTTPS requests to the S3 API in the background.

LISTING ALL THE IMAGES IN THE S3 BUCKET

To display a list of images, the application needs to list all the objects in your bucket. This can be done with the S3 service's `listObjects()` function. Listing 7.2 shows the implementation of the corresponding function in the `server.js` JavaScript file, acting as a web server.

Listing 7.2 Retrieving all the image locations from the S3 bucket

```
const bucket = process.argv[2];      ①

async function listImages(response) {
  try {
    let data = await s3.listObjects({      ②
      Bucket: bucket      ③
    }).promise();
    let stream = mu.compileAndRender(      ④
      'index.html',
      {
        Objects: data.Contents,
        Bucket: bucket
      }
    );
    stream.pipe(response);      ⑤
  } catch (err) {      ⑥
    console.error(err);
    response.status(500);
    response.send('Internal server error.');
  }
}
```

- ① Reads the bucket name from the process arguments
- ② Lists the objects stored in the bucket

- ③ The bucket name is the only required parameter
- ④ Renders a HTML page based on the list of objects
- ⑤ Stream the response
- ⑥ Handle potential errors

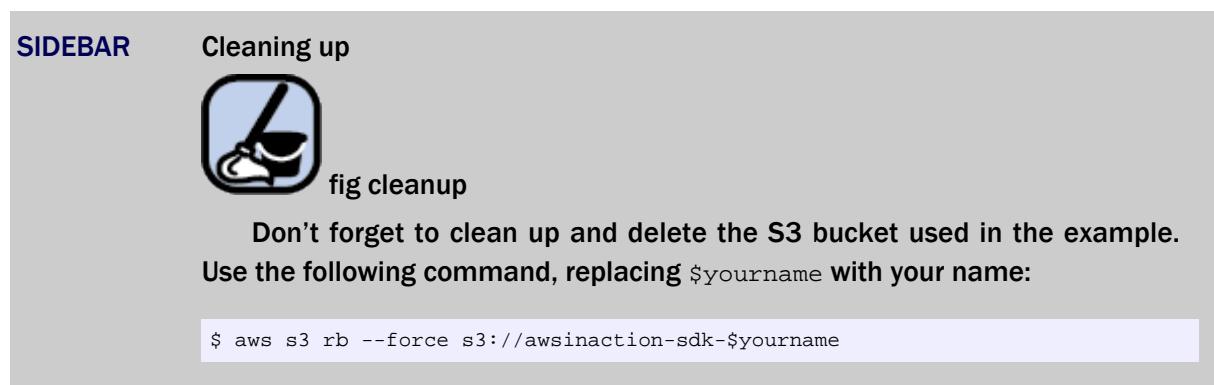
Listing the objects returns the names of all the images from the bucket, but the list doesn't include the image content. During the uploading process, the access rights to the images are set to public read. This means anyone can download the images with the bucket name and a random key. Listing 7.3 shows an excerpt of the index.html template, which is rendered on request. The `Objects` variable contains all the objects from the bucket.

Listing 7.3 Template to render the data as HTML

```
[...]
<h2>Images</h2>
{{#Objects}}
  ①
  <p><img src=      ②
  [CA] "https://s3.amazonaws.com/{{Bucket}}/{{Key}}"
  [CA] width="400px" ></p>
{{/Objects}}
[...]
```

- ① Iterates over all objects
- ② Puts together the URL to fetch an image from the bucket

You've now seen the three important parts of the Simple S3 Gallery integration with S3: uploading an image, listing all images, and downloading an image.



You've learned how to use S3 using the AWS SDK for JavaScript and Node.js. Using the AWS SDK for other programming languages is similar.

The next section is about a different scenario, you will learn how to host static websites on

7.6 Using S3 for static web hosting

We have started our blog cloudonaut.io in May 2015. The most popular blog posts like “ECS vs. Fargate: What’s the difference?” (cloudonaut.io/ecs-vs-fargate-whats-the-difference/), “Advanced AWS Networking: Pitfalls That You Should Avoid” (cloudonaut.io/advanved-aws-networking-pitfalls-that-you-should-avoid/), and “CloudFormation vs Terraform” (cloudonaut.io/cloudformation-vs-terraform/) have been read more than 200,000 times. But we did not need to operate any VMs to publish our blog posts. Instead, we used S3 to host our static website built with a static site generator hexo.io. This approach provides a cost-effective, scalable, and maintenance-free infrastructure for our blog.

You can host a static website with S3 and deliver static content like HTML, JavaScript, CSS, images (such as PNG and JPG), audio, and videos as well. But keep in mind that you can’t execute server-side scripts like PHP or JSP. For example, it’s not possible to host WordPress, a CMS based on PHP, on S3.

SIDE BAR

Increasing speed by using a CDN

Using a content-delivery network (CDN) helps reduce the load time for static web content. A CDN distributes static content like HTML, CSS, and images to nodes all around the world. If a user sends out a request for some static content, the request is answered from the nearest available node with the lowest latency. Various providers offer CDNs. Amazon CloudFront is the CDN offered by AWS. When using CloudFront, users connect to CloudFront to access your content, which is which is fetched from S3 or other sources. See the [CloudFront documentation at docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/DownloadDistS3AndCustomOrigins.html](http://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/DownloadDistS3AndCustomOrigins.html) if you want to set this up; we won’t cover it in this book.

In addition, S3 offers the following features for hosting a static website:

- Defining a custom index document and error documents. For example, you can define *index.html* as the default index document.
- Defining redirects for all or specific requests. For example, you can forward all requests from */img/old.png* to */img/new.png*.
- Setting up a custom domain for S3 bucket. For example, Andreas might want to set up a domain like *mybucket.andreaswittig.info* pointing to his bucket.

7.6.1 Creating a bucket and uploading a static website

First you need to create a new S3 bucket. To do so, open your terminal and execute the following command, replacing `$BucketName` with your own bucket name. As we’ve mentioned, the bucket name has to be globally unique. If you want to link your domain name to S3, you must use your entire domain name as the bucket name.

```
$ aws s3 mb s3://$BucketName
```

The bucket is empty; you'll place an HTML document in it next. We've prepared a placeholder HTML file. Download it to your local machine from the following URL: raw.githubusercontent.com/AWSinAction/code3/master/chapter07/helloworld.html. You can now upload the file to S3. Execute the following command to do so, replacing \$pathToPlaceholder with the path to the HTML file you downloaded in the previous step and \$BucketName with the name of your bucket:

```
$ aws s3 cp $pathToPlaceholder/helloworld.html \
[CA] s3://$BucketName/helloworld.html
```

You've now created a bucket and uploaded an HTML document called helloworld.html. You need to configure the bucket next.

7.6.2 Configuring a bucket for static web hosting

By default, only you, the owner, can access files from your S3 bucket. As you want to use S3 to deliver your static website, so you'll need to allow everyone to view or download the documents included in your bucket. A *bucket policy* helps you control access to bucket objects globally. You already know from chapter 5 that policies are defined in JSON and contain one or more statements that either allow or deny specific actions on specific resources. Bucket policies are similar to IAM policies.

Download our bucket policy from the following URL: raw.githubusercontent.com/AWSinAction/code3/master/chapter07/bucketpolicy.json. You need to edit the bucketpolicy.json file next, as shown in the following listing. Open the file with the editor of your choice, and replace \$BucketName with the name of your bucket.

Listing 7.4 Bucket policy allowing read-only access to every object in a bucket

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AddPerm",
      "Effect": "Allow",      ①
      "Principal": "*",       ②
      "Action": ["s3:GetObject"], ③
      "Resource": ["arn:aws:s3:::$BucketName/*"] ④
    }
  ]
}
```

- ① Allows access
- ② For anyone
- ③ Read objects

④ Your bucket

You can add a bucket policy to your bucket with the following command. Replace `$BucketName` with the name of your bucket and `$pathToPolicy` with the path to the `bucketpolicy.json` file:

```
$ aws s3api put-bucket-policy --bucket $BucketName \
[CA] --policy file://$pathToPolicy/bucketpolicy.json
```

Every object in the bucket can now be downloaded by anyone. You need to enable and configure the static web-hosting feature of S3 next. To do so, execute the following command, replacing `$BucketName` with the name of your bucket:

```
$ aws s3 website s3://$BucketName --index-document helloworld.html
```

Your bucket is now configured to deliver a static website. The HTML document `helloworld.html` is used as index page. You'll learn how to access your website next.

7.6.3 Accessing a website hosted on S3

You can now access your static website with a browser. To do so, you need to choose the right endpoint. The endpoints for S3 static web hosting depend on your bucket's region. For `us-east-1` (US East N. Virginia), the website endpoint looks like this.

```
http://$BucketName.s3-website-us-east-1.amazonaws.com
```

Replace `$BucketName` with your bucket. So if your bucket is called `awesomebucket` and was created in the default region `us-east-1`, your bucket name would be:

```
http://awesomebucket.s3-website-us-east-1.amazonaws.com
```

Open this URL with your browser, and you should be welcomed by a Hello World website.

Please note, that for some regions, the website endpoint looks a little different. Check [S3 endpoints and quotas](#) for details.

SIDE BAR**Linking a custom domain to an S3 bucket**

If you want to avoid hosting static content under a domain like `awsinaction.s3-website-us-east-1.amazonaws.com`, you can link a custom domain to an S3 bucket, such as `awsinaction.example.com`. All you have to do is to add a CNAME record for your domain, pointing to the bucket's S3 endpoint. The domain name system provided by AWS allowing you to create a CNAME record is called Route 53.

The CNAME record will only work if you comply with the following rules:

- Your bucket name must match the CNAME record name. For example, if you want to create a CNAME for `awsinaction.example.com`, your bucket name must be `awsinaction.example.com` as well.
- CNAME records won't work for the primary domain name (such as `example.com`). You need to use a subdomain for CNAMEs like `awsinaction` or `www`, for example. If you want to link a primary domain name to an S3 bucket, you need to use the Route 53 DNS service from AWS.

Linking a custom domain to your S3 bucket only works for HTTP. If you want to use HTTPS (and you probably should), use AWS CloudFront together with S3. AWS CloudFront accepts HTTPS from the client and forwards the request to S3.

SIDE BAR**Cleaning up**

`fig cleanup`

Don't forget to clean up your bucket after you finish the example. To do so, execute the following command, replacing `$BucketName` with the name of your bucket:

```
$ aws s3 rb --force s3://$BucketName
```

7.7 Protecting data from unauthorized access

Not a week goes by without a frightening announcement that an organization has leaked confidential data from Amazon S3 accidentally. Why is that?

While reading through this chapter, you have learned about different scenarios for using S3. For example, you used S3 to backup data from your local machine. Also, you hosted a static website on S3. So S3 is used to store sensitive data as well as public data. This can be a dangerous mix as a misconfiguration might cause a data leak.

To mitigate the risk, we recommend to enable *Block Public Access* for all your buckets as illustrated in figure 7.5. By doing so, you will disable public access to all the buckets belonging to your AWS account. So this will break S3 website hosting or any other form of accessing S3 objects publicly.

1. Open the AWS Management Console and navigate to S3.
2. Select *Block Public Access settings for this account* from the sub navigation.
3. Enable *Block all public access* and click the *Save changes* button.

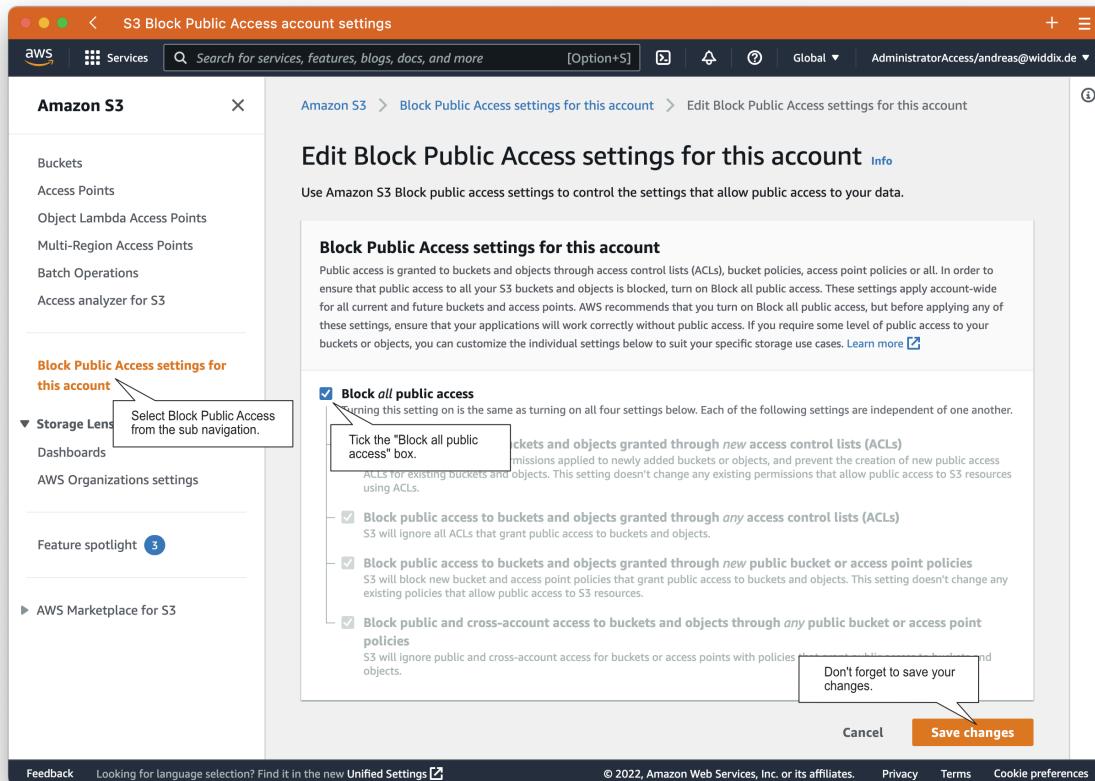


Figure 7.5 Enable block public access for all S3 buckets to avoid data leaks

In case you really need both buckets with sensitive data and buckets with public data, you should not enable *Block Public Access* on the account level but for all buckets with sensitive data individually instead.

Check out our blog post [How to avoid S3 data leaks?](#) in case you are interested in further advice.

7.8 Optimizing performance

By default, S3 handles 3,500 writes and 5,500 reads per second. In case your workload requires higher throughput, you need to consider the following when coming up with the naming scheme for the object keys.

Objects are stored without a hierarchy on S3. There is no such thing as a directory. All you do is

to specify an object key, as discussed at the beginning of the chapter. However, using a prefix allows you to structure the object keys.

By default, the slash character (/) is used as the prefix delimiter. So, `archive` is the prefix in the following example of object keys.

```
archive/image1.png
archive/image2.png
archive/image3.png
archive/image4.png
```

Be aware, that the maximum throughput per partitioned prefix is 3,500 writes and 5,500 reads per second. Therefore, you cannot read more than 5,500 objects from prefix `archive` per second.

To increase, the maximum throughput you need to distribute your objects among additional prefixes. For example, you could organize the objects from the example above like this.

```
archive/2021/image1.png
archive/2021/image2.png
archive/2022/image3.png
archive/2022/image4.png
```

By doing so, you can double the maximum throughput when reading from `archive/2021` and `archive/2022` as illustrated in figure [7.6](#).

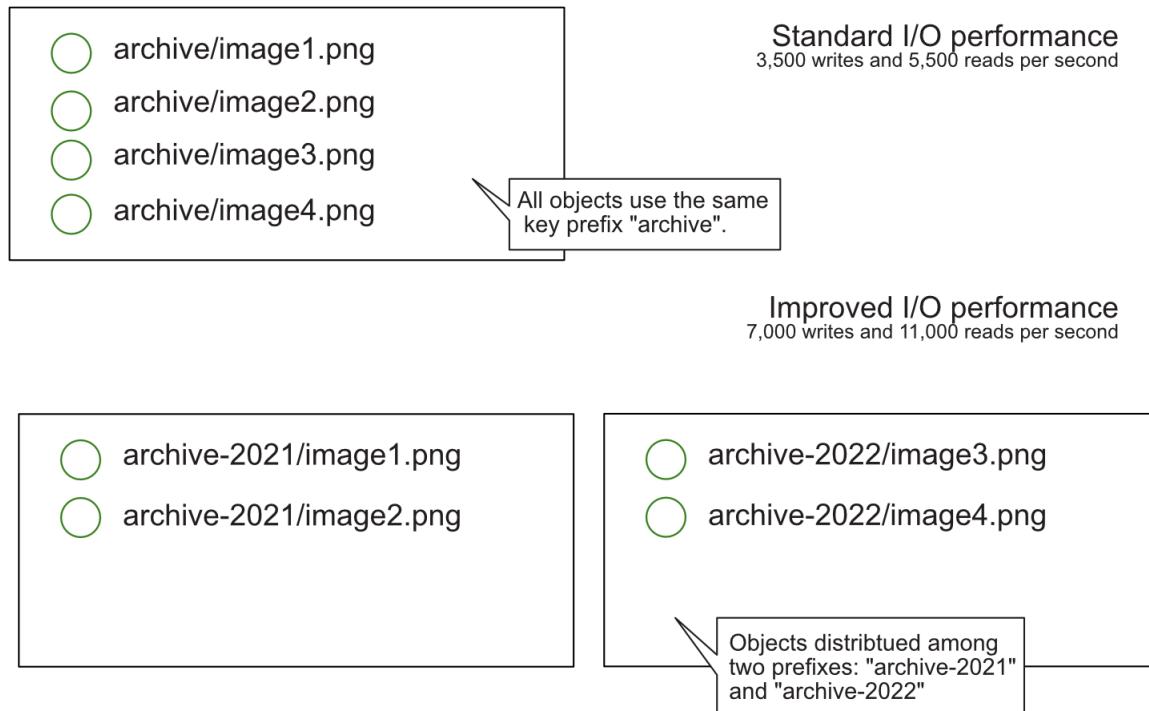


Figure 7.6 To improve I/O performance distribute requests among multiple object key prefixes

In summary, the structure of your object keys has an effect on the maximum read and write throughput.

7.9 Summary

- An object consists of a unique identifier, metadata to describe and manage the object, and the content itself. You can save images, documents, executables, or any other content as an object in an object store.
- Amazon S3 provides endless storage capacity with a pay-per-use model. You are charged for storage as well as read and write requests.
- Amazon S3 is an object store accessible only via HTTP(S). You can upload, manage, and download objects with the CLI, SDKs, or the Management Console. The storage classes *Glacier Instant Retrieval*, *Glacier Flexible Retrieval*, and *Glacier Deep Archive* are designed to archive data at low costs.
- Integrating S3 into your applications will help you implement the concept of a stateless server, because you don't have to store objects locally on the server.
- Enable *Block Public Access* for all buckets, or at least those buckets that contain sensitive information to avoid data leaks.
- When optimizing for high performance make sure to use many different key prefixes instead of similar or the same prefix for all objects.

Notes

1. AWS Customer Success, aws.amazon.com/solutions/case-studies/.

13

Achieving high availability: Availability zones, auto-scaling, and CloudWatch

This chapter covers

- Recovering a failed virtual machine with a CloudWatch alarm
- Using auto-scaling to guarantee your virtual machines keep running
- Understanding availability zones in an AWS region
- Analyzing disaster-recovery requirements

Imagine you run an online shop. During the night, the hardware running your virtual machine fails. Until the next morning when you go into work, your users can no longer access your web shop. During the 8-hour downtime, your users search for an alternative and stop buying from you. That's a disaster for any business. Now imagine a highly available web shop. Just a few minutes after the hardware failed, the system recovers, restarts itself on new hardware, and your e-commerce website is back online again—without any human intervention. Your users can now continue to shop on your site. In this chapter, we'll teach you how to build a high-availability system based on EC2 instances like that.

Virtual machines are not highly available by default, the potential for system failure is always present. The following scenarios could cause an outage of your virtual machine:

- A software issue causes the virtual machine's OS to fail.
- A software issue occurs on the host machine, causing the virtual machine to crash (either the OS of the host machine crashes or the virtualization layer does).
- The computing, storage, or networking hardware of the physical host fails.
- Parts of the data center that the virtual machine depends on fail: network connectivity, the power supply, or the cooling system.

For example, if the computing hardware of a physical host fails, all EC2 instances running on this host will fail. If you're running an application on an affected virtual machine, this

application will fail and experience downtime until somebody—probably you—intervenes by starting a new virtual machine on another physical host. To avoid downtimes and to build a highly available system, you should enable auto-recovery or use multiple virtual machines.

SIDE BAR**Examples are 100% covered by the Free Tier**

The examples in this chapter are totally covered by the Free Tier. As long as you don't run the examples longer than a few days, you won't pay anything for it. Keep in mind that this applies only if you created a fresh AWS account for this book and there is nothing else going on in your AWS account. Try to complete the chapter within a few days, because you'll clean up your account at the end of the chapter.

High availability describes a system that is operating with almost no downtime. Even if a failure occurs, the system can provide its services most of the time. The Harvard Research Group (HRG) defines high availability with the classification AEC-2, which requires an uptime of 99.99% over a year, or not more than 52 minutes and 35.7 seconds of downtime per year. You can achieve 99.99% uptime with EC2 instances if you follow the instructions in the rest of this chapter. Although a short interruption might be necessary to recover from a failure, there is no need for human intervention to instigate the recovery.

SIDE BAR**High availability vs. fault tolerance**

A highly available system can recover from a failure automatically with a short downtime. A fault-tolerant system, in contrast, requires the system to provide its services without interruption in case of a component failure. We'll show you how to build a fault-tolerant system in chapter 16.

AWS provides building blocks for highly available systems based on EC2 instances. Depending on whether or not you can distribute a workload across multiple machines, different tools come into play.

- Building a highly available infrastructure by using groups of isolated data centers, called *availability zones*, within a region.
- Monitoring the health of virtual machines with CloudWatch and triggering recovery automatically if needed. This option fits best for workloads that need to run on a single virtual machine.
- Using auto-scaling to guarantee a certain number of virtual machines is up and running, and replace failed instances automatically. Use this approach when distributing your workload among multiple virtual machines is an option.

You will learn how to automatically replace a failed virtual machine in the first section of this chapter. But what happens in case of a data center outage? You will learn how to recover from a data center outage in the second section of this chapter. At the end of the chapter, we will discuss

how to analyze disaster recovery requirements and translate them into an AWS architecture.

In the following section, you'll learn how to protect a workload that can only run on a single virtual machine at a time from failures.

13.1 Recovering from EC2 instance failure with CloudWatch

AWS does not recover failed virtual machines automatically. An easy countermeasure is to automate the recovery of an EC2 instance by creating a CloudWatch alarm. You will learn how to do so in this section.

Suppose that your team is using an agile development process. To accelerate the process, your team decides to automate the testing, building, and deployment of the software. You've been asked to set up a continuous integration (CI) server which allows you to automate the development process. You've chosen to use Jenkins, an open source application written in Java that runs in a servlet container such as Apache Tomcat. Because you're using infrastructure as code, you're planning to deploy changes to your infrastructure with Jenkins as well.²

A Jenkins server is a typical use case for a high-availability setup. It's an important part of your infrastructure, because your colleagues won't be able to test and deploy new software if Jenkins suffers from downtime. But a short downtime in the case of a failure with automatic recovery won't hurt your business too much, so you don't need a fault-tolerant system. Jenkins is only an example. You can apply the same principles to any other applications where you can tolerate a short amount of downtime but still want to recover from hardware failures automatically. For example, we used the same approach for hosting FTP servers and VPN servers.

In this example, you'll do the following:

1. Create a virtual network in the cloud (VPC).
2. Launch a virtual machine in the VPC, and automatically install Jenkins during bootstrap.
3. Create a CloudWatch alarm to monitor the health of the virtual machine and replace the virtual machine when needed.

SIDE BAR

AWS CloudWatch

AWS CloudWatch is a service offering metrics, events, logs, and alarms for AWS resources. You used CloudWatch to monitor a Lambda function in chapter 7, and gained some insight into the current load of a relational database instance in chapter 11.

We'll guide you through these steps with the help of a CloudFormation template.

You can find the CloudFormation template for this example on GitHub and on S3. Download a snapshot of the repository at <http://github.com/AWSinAction/code3/archive/master.zip>. The file

we're talking about is located at `chapter13/recovery.yaml`. On S3, the same file is located at <https://s3.amazonaws.com/awsinaction-code3/chapter14/recovery.yaml>.

The following command creates a CloudFormation template which launches an EC2 instance with a CloudWatch alarm that triggers a recovery if the virtual machine fails. Replace `$Password` with a password consisting of 8-40 characters. The template automatically installs a Jenkins server while starting the virtual machine:

```
$ aws cloudformation create-stack --stack-name jenkins-recovery \
→--template-url https://s3.amazonaws.com/\
→awsinaction-code3/chapter14/recovery.yaml \
→--parameters "ParameterKey=JenkinsAdminPassword,
→ParameterValue=$Password" \
→--capabilities CAPABILITY_IAM
```

While you are waiting for CloudFormation to launch the EC2 instance, we will have a look into the CloudFormation template.

The CloudFormation template contains the definition of a private network. But the most important parts of the template are these:

- A virtual machine with user data containing a Bash script, which installs a Jenkins server during bootstrapping
- A public IP address assigned to the EC2 instance, so you can access the new instance after a recovery using the same public IP address as before
- A CloudWatch alarm based on the system-status metric published by the EC2 service

Listing 13.1 shows how to create an EC2 instance that runs a script to install Jenkins during bootstrapping. The Elastic IP address ensures a static public IP address for the virtual machine.

Listing 13.1 Starting an EC2 instance running a Jenkins CI server with a recovery alarm

```
# [...]
ElasticIP:
  Type: 'AWS::EC2::EIP'    ①
  Properties:
    InstanceId: !Ref Instance
    Domain: vpc
    DependsOn: VPCGatewayAttachment
Instance: ②
  Type: 'AWS::EC2::Instance'
  Properties:
    ImageId: !FindInMap [RegionMap, !Ref 'AWS::Region', AMI] ③
    InstanceType: 't2.micro' ④
    IamInstanceProfile: !Ref IamInstanceProfile
    NetworkInterfaces:
      - AssociatePublicIpAddress: true
        DeleteOnTermination: true
        DeviceIndex: 0
        GroupSet:
          - !Ref SecurityGroup
        SubnetId: !Ref Subnet
    UserData: ⑤
      'Fn::Base64': !Sub |
        #!/bin/bash -ex
        trap '/opt/aws/bin/cfn-signal -e 1 --stack ${AWS::StackName} \
        ↵--resource Instance --region ${AWS::Region}' ERR

        # Installing Jenkins ⑥
        amazon-linux-extras enable epel=7.11 && yum -y clean metadata
        yum install -y epel-release && yum -y clean metadata
        yum install -y java-11-amazon-corretto-headless daemonize
        wget -q -T 60 http://.../jenkins-2.319.1-1.1.noarch.rpm
        rpm --install jenkins-2.319.1-1.1.noarch.rpm

        # Configuring Jenkins
        # ...

        # Starting Jenkins ⑦
        systemctl enable jenkins.service
        systemctl start jenkins.service
        /opt/aws/bin/cfn-signal -e $? --stack ${AWS::StackName} \
        ↵--resource Instance --region ${AWS::Region}

Tags:
  - Key: Name
    Value: 'jenkins-recovery'
CreationPolicy:
  ResourceSignal:
    Timeout: PT10M
  DependsOn: VPCGatewayAttachment
```

- ① The public IP address stays the same after recovery when using ElasticIP.
- ② Launches a virtual machine to run a Jenkins server
- ③ Selects the AMI (in this case Amazon Linux)
- ④ Recovery is supported for t2 instance types.
- ⑤ User data containing a shell script that is executed during bootstrapping to install a Jenkins server
- ⑥ Downloads and installs Jenkins
- ⑦ Starts Jenkins

In case the EC2 instance fails, AWS will not replace the instance automatically. Instead you need to create a CloudWatch alarm to trigger the recovery of the virtual machine automatically.

A CloudWatch alarm consists of the following:

- A metric that monitors data (health check, CPU usage, and so on)
- A rule defining a threshold based on a statistical function over a period of time
- Actions to trigger if the state of the alarm changes (such as triggering a recovery of an EC2 instance if the state changes to ALARM)

An alarm is in one of the following states:

- OK—Everything is fine; the threshold hasn't been reached.
- INSUFFICIENT_DATA—There isn't enough data to evaluate the alarm.
- ALARM—Something is broken: the threshold has been overstepped.

Listing 13.2 creates a CloudWatch alarm based on a metric called `StatusCheckFailed_System` (referenced by attribute `MetricName`) to monitor a virtual machine's health and recover it in case the underlying host system fails. This metric contains the results of the system status checks performed by the EC2 service every minute. If the check fails, a measurement point with value 1 is added to the metric `StatusCheckFailed_System`. Because the EC2 service publishes this metric, the `Namespace` is called `AWS/EC2` and the `Dimension` of the metric is the ID of a virtual machine.

The CloudWatch alarm checks the metric every 60 seconds, as defined by the `Period` attribute. As defined in `EvaluationPeriods`, the alarm will check the last five periods, the last 5 minutes in this example. The check runs a statistical function specified in `Statistic` on the time periods. The result of the statistical function, a maximum function in this case, is compared against `Threshold` using the chosen `ComparisonOperator`. If the result is negative, the alarm actions defined in `AlarmActions` are executed: in this case, the recovery of the virtual machine—a built-in action for EC2 instances.

In summary, AWS checks the status of the virtual machine every minute. The result of these checks is written to the `StatusCheckFailed_System` metric. The alarm checks this metric. If there are five consecutive failed checks, the alarm trips.

Listing 13.2 Creating a CloudWatch alarm to recover a failed EC2 instance

```

RecoveryAlarm: ①
  Type: 'AWS::CloudWatch::Alarm'
  Properties:
    AlarmDescription: 'Recover EC2 instance when underlying hardware fails.'
    Namespace: 'AWS/EC2' ②
    MetricName: 'StatusCheckFailed_System' ③
    Statistic: Maximum ④
    Period: 60 ⑤
    EvaluationPeriods: 5 ⑥
    ComparisonOperator: GreaterThanThreshold ⑦
    Threshold: 0 ⑧
    AlarmActions:
      - !Sub 'arn:aws:automate:${AWS::Region}:ec2:recover'
    Dimensions:
      - Name: InstanceId ⑩
        Value: !Ref Instance
  
```

- ① Creates a CloudWatch alarm to monitor the health of the virtual machine
- ② The metric to monitor is provided by the EC2 service with namespace AWS/EC2.
- ③ The name of the metric
- ④ Statistical function to apply to the metric. The minimum is to notify you if a single status check failed.
- ⑤ Duration the statistical function is applied, in seconds. Must be a multiple of 60.
- ⑥ Number of time periods over which data is compared to the threshold
- ⑦ Operator for comparing the output of the statistical function with the threshold
- ⑧ Threshold triggering an alarm
- ⑨ Action to perform in case of an alarm. Uses the predefined recovery action for EC2 instances.
- ⑩ The virtual machine is a dimension of the metric.

While you have been reading through the details the CloudFormation stack should have reached the status COMPELTE. Run the following command to get the output of the stack. If the output is null, retry after a few more minutes:

```
$ aws cloudformation describe-stacks --stack-name jenkins-recovery \
  ➔ --query "Stacks[0].Outputs"
```

If the query returns output like the following, containing a URL, a user, and a password, the stack has been created and the Jenkins server is ready to use. If not, retry the command after a few minutes. Next, open the URL in your browser, and log in to the Jenkins server with user `admin` and the password you've chosen:

```
[
  {
    "Description": "URL to access web interface of Jenkins server.",
    "OutputKey": "JenkinsURL",
    "OutputValue": "http://54.152.240.91:8080"      ①
  },
  {
    "Description": "Administrator user for Jenkins.",
    "OutputKey": "User",
    "OutputValue": "admin"                         ②
  },
  {
    "Description": "Password for Jenkins administrator user.",
    "OutputKey": "Password",
    "OutputValue": "*****"                        ③
  }
]
```

- ① Open this URL in your browser to access the web interface of the Jenkins server.
- ② Use this user name to log in to the Jenkins server.
- ③ Use this password to log in to the Jenkins server.

Now you'll want to try out whether the Jenkins server works. You're ready to create your first build job on the Jenkins server. To do so, you have to log in with the username and password from the previous output. Figure 13.1 shows the Jenkins server's login form.

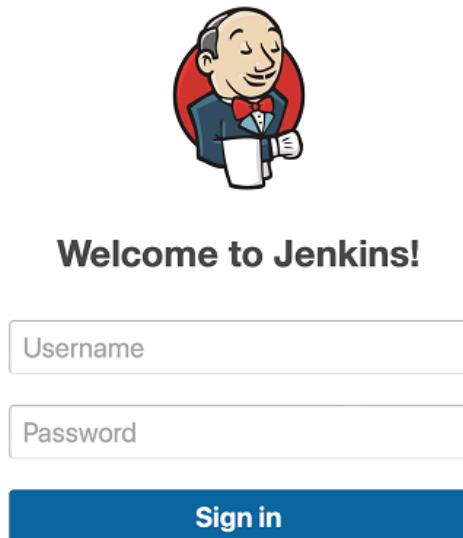


Figure 13.1 Web interface of the Jenkins server

The following steps guide you through the process of creating a Jenkins project.

1. Open `http://$PublicIP:8080/newJob` in your browser, and replace `$PublicIP` with the

- public IP address from the output of the previous describe command.
2. Log in with user `admin` and the password you chose when starting the CloudFormation template.
 3. Select the `Install selected plugins` option.
 4. Keep the default for Jenkins URL and click `Save and Finish`.
 5. Click `Start using Jenkins`.
 6. Click `New Item` to create a new project.
 7. Type `AWS` in `Action` as the name for the new project.
 8. Select `Freestyle project` as the job type, and click `OK`.

The Jenkins server runs on a virtual machine with automated recovery. If the virtual machine fails because of issues with the host system, it will be recovered with all data and the same public IP address. The URL doesn't change because you're using an Elastic IP for the virtual machine. All data is restored because the new virtual machine uses the same EBS volume as the previous virtual machine, so you can find your *AWS in Action* job again.

Unfortunately, you can't test the recovery process. The CloudWatch alarm monitors the health of the host system, which can only be controlled by AWS.

13.1.1 How does a CloudWatch alarm recover an EC2 instance?

After you launched an EC2 instance with self-healing capabilities, it is time to take a look into the details. The EC2 service checks the status of every virtual machine automatically. System status checks are performed every minute and the results are available as CloudWatch metrics.

A *system status check* detects a loss of network connectivity or power, as well as software or hardware issues on the physical host. By default, AWS is not recovering failed EC2 instances automatically. But you can configure a CloudWatch alarm based on the system status check to restart a failed virtual machine on another physical host.

Figure [13.2](#) shows the process in the case of an outage affecting a virtual machine:

1. The physical hardware fails and causes the EC2 instance to fail as well.
2. The EC2 service detects the outage and reports the failure to a CloudWatch metric.
3. A CloudWatch alarm triggers recovery of the virtual machine.
4. The EC2 instance is launched on another physical host.
5. The EBS volume and Elastic IP stay the same, and are linked to the new EC2 instance.

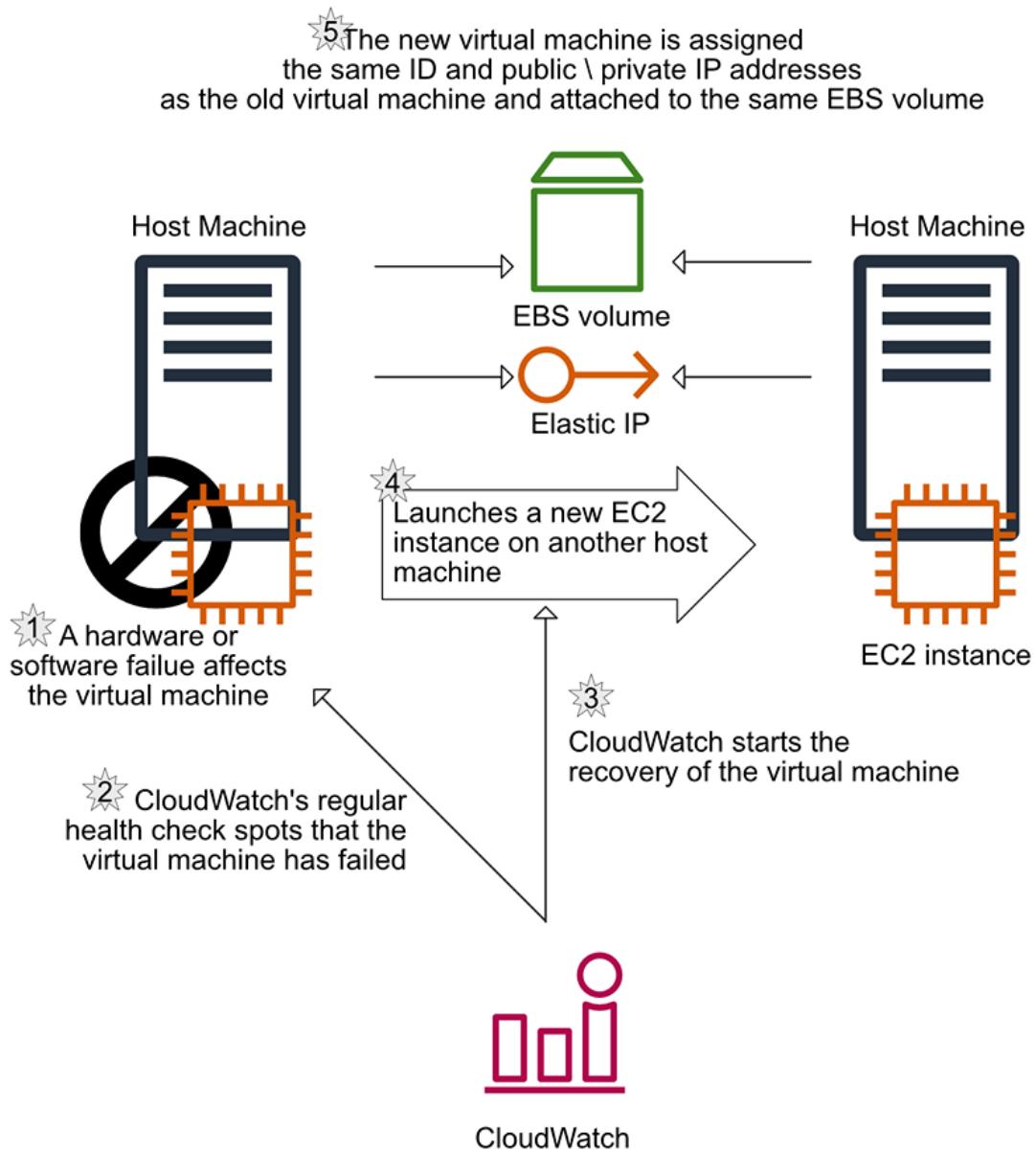


Figure 13.2 In the case of a hardware failure, CloudWatch triggers the recovery of the EC2 instance.

After the recovery, a new EC2 instance is running with the same ID and private IP address. Data on network-attached EBS volumes is available as well. No data is lost because the EBS volume stays the same. EC2 instances with local disks (instance storage) aren't supported for this process. If the old EC2 instance was connected to an Elastic IP address, the new EC2 instance is connected to the same Elastic IP address.

SIDE BAR**Requirements for recovering EC2 instances**

An EC2 instance must meet the following requirements if you want to use the recovery feature:

- It must be running in a VPC network.
- The instance family must be A1, C3, C4, C5, C5a, C5n, C6g, C6gn, Inf1, C6i, M3, M4, M5, M5a, M5n, M5zn, M6g, M6i, P3, R3, R4, R5, R5a, R5b, R5n, R6g, R6i, T2, T3, T3a, T4g, X1, and X1e. Other instance families aren't supported.
- The EC2 instance must use EBS volumes exclusively, because data on instance storage would be lost after the instance was recovered.

SIDE BAR**Cleaning up**

Now that you've finished this example, it's time to clean up to avoid unwanted charges. Execute the following command to delete all resources corresponding to the Jenkins setup:

```
$ aws cloudformation delete-stack --stack-name jenkins-recovery
$ aws cloudformation wait stack-delete-complete \
  --stack-name jenkins-recovery ①
```

① Waits until the stack is deleted

There is an important limitation to the approach you learned about in this section. The CloudWatch alarm will recover a failed instance, but can do so in the same data center only. In case a whole data center fails, your Jenkins server will become unavailable and will not recover automatically.

13.2 Recovering from a data center outage with Auto Scaling Group

Recovering an EC2 instance after underlying software or hardware fails is possible using system status checks and CloudWatch, as described in the previous section. But what happens if the entire data center fails because of a power outage, a fire, or some other disaster? Recovering a virtual machine as described in section 14.1 will fail because it tries to launch an EC2 instance in the *same data center*.

AWS is built for failure, even in the rare case that an entire data center fails. The AWS regions consist of multiple data centers grouped into *availability zones*. By distributing your workload

among multiple availability zones, you are able to recover from an data center outage. There are two pitfalls when building a highly available setup over multiple availability zones:

1. Data stored on network-attached storage (EBS) won't be available after failing over to another availability zone by default. So you can end up having no access to your data (stored on EBS volumes) until the availability zone is back online (you won't lose your data in this case).
2. You can't start a new virtual machine in another availability zone with the same private IP address. That's because, subnets are bound to availability zones, and each subnet has a unique IP address range. By default, you can't keep the same public IP address automatically after a recovery, as was the case in the previous section with a CloudWatch alarm triggering a recovery.

We will deal with those pitfalls at the end of this section.

In this section, you'll improve the Jenkins setup from the previous section, add the ability to recover from an outage of an entire availability zone, and work around the pitfalls afterward.

13.2.1 Availability zones: groups of isolated data centers

As you've learned, AWS operates multiple locations worldwide, called *regions*. You've used region US East (N. Virginia), also called us-east-1, if you've followed the examples so far. In total, there are 22 publicly available regions throughout North America, South America, Europe, Africa, and Asia Pacific.

Each region consists of multiple *availability zones (AZs)*. You can think of an AZ as an isolated group of data centers, and a region as an area where multiple availability zones are located at a sufficient distance. The region us-east-1 consists of six availability zones (us-east-1a to us-east-1f) for example. The availability zone us-east-1a could be one data center, or many. We don't know because AWS doesn't make information about their data centers publicly available. So from an AWS user's perspective, you only know about regions and AZs.

The AZs are connected through low-latency links, so requests between different availability zones aren't as expensive as requests across the internet in terms of latency. The latency within an availability zone (such as from an EC2 instance to another EC2 instance in the same subnet) is lower compared to latency across AZs. The number of availability zones depends on the region. All regions come with three or more availability zones. Keep in mind, that AWS charges \$0.01/GB for network traffic between availability zones. Figure [13.3](#) illustrates the concept of availability zones within a region.

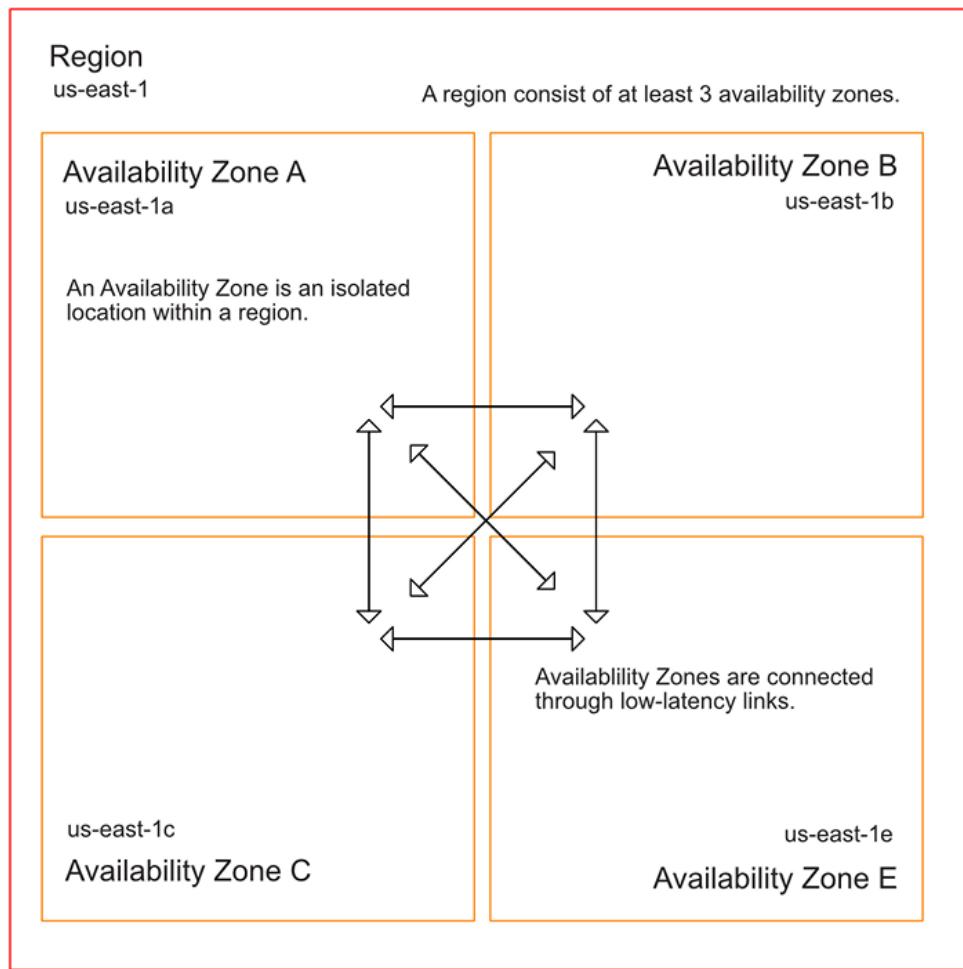


Figure 13.3 A region consists of multiple availability zones connected through low-latency links.

13.2.2 Recovering a failed virtual machine to another availability zone with the help of auto-scaling

In the first part of the chapter, you used a CloudWatch alarm to trigger the recovery of a virtual machine that was running a Jenkins CI server, in case of a failure. This mechanism launches an identical copy of the original virtual machine if necessary. This is only possible in the same availability zone, because the private IP address and the EBS volume of a virtual machine are bound to a single subnet and a single availability zone. But suppose your team isn't happy about the fact that they won't be able to use the Jenkins server to test, build, and deploy new software in case of an unlikely availability zone outage. You need a tool that will let you recover in another availability zone.

Failing over into another availability zone is possible with the help of *auto-scaling*. You can find the CloudFormation template for this example on GitHub and on S3. Download a snapshot of the repository at <https://github.com/AWSinAction/code3/archive/master.zip>. The file we're talking about is located at `chapter14/multiaz.yaml`. On S3, the same file is located at <https://s3.amazonaws.com/awsinaction-code3/chapter13/multiaz.yaml>.

Auto-scaling is part of the EC2 service and helps you to ensure that a specified number of EC2 instances is running even when availability zones become unavailable. You can use auto-scaling to launch a virtual machine and make sure a new instance is started if the original instance fails. You can use it to start virtual machines in multiple subnets. So in case of an outage of an entire availability zone, a new instance can be launched in another subnet in another availability zone.

Execute the following command to create a virtual machine that can recover in another availability zone if necessary. Replace `$Password` with a password consisting of 8-40 characters.

```
$ aws cloudformation create-stack --stack-name jenkins-multiaz \
  --template-url https://s3.amazonaws.com/\
  awsinaction-code3/chapter14/multiaz.yaml \
  --parameters "ParameterKey=JenkinsAdminPassword,
  ParameterValue=$Password" \
  --capabilities CAPABILITY_IAM
```

While you wait for CloudFormation to create the stack, you will learn more about the details.

To configure auto-scaling, you need to create two parts of the configuration:

- A *launch template* contains all information needed to launch an EC2 instance: instance type (size of virtual machine) and image (AMI) to start from.
- An *auto-scaling group* tells the EC2 service how many virtual machines should be started with a specific launch template, how to monitor the instances, and in which subnets EC2 instances should be started.

Figure [13.4](#) illustrates this process.

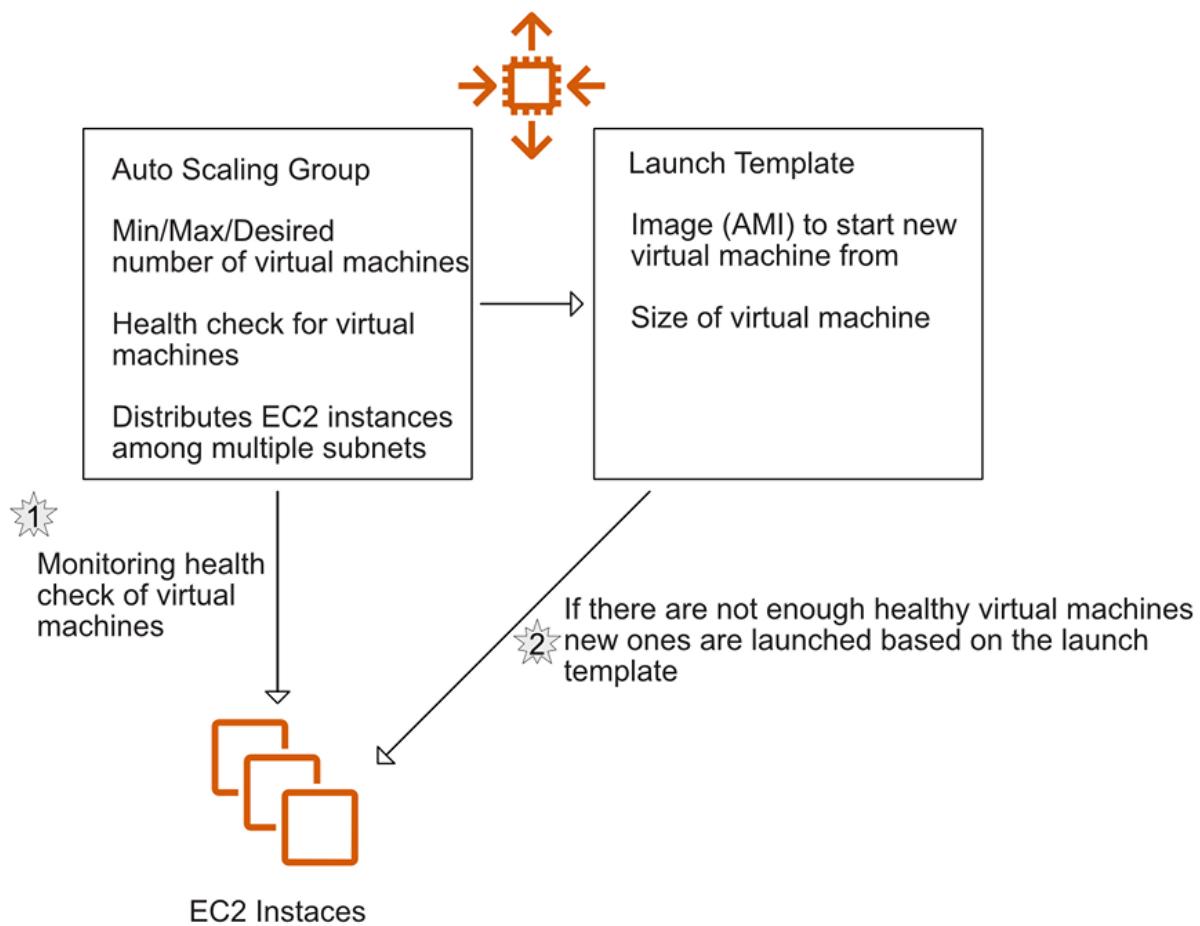


Figure 13.4 Auto-scaling ensures that a specified number of EC2 instances are running.

You'll find both a launch template and an auto-scaling group in the CloudFormation template launching a Jenkins server as shown in listing 13.3. The parameters are explained in table 13.1. You already used the most important parameters for the launch template when starting a single virtual machine with a CloudWatch recovery alarm in the previous section.

Table 13.1 Required parameters for the launch template and auto-scaling group

Context	Property	Description	Values
LaunchTemplate	ImageId	The ID of the AMI the virtual machine should be started from.	Any AMI ID accessible from your account.
LaunchTemplate	InstanceType	The size of the virtual machine.	All available instance sizes, such as t2.micro, m3.medium, and c3.large.
LaunchTemplate	SecurityGroupIds	References the security groups for the EC2 instance.	Any security group belonging to the same VPC.
LaunchTemplate	UserData	Script executed during bootstrap to install the Jenkins CI server	Any bash script.
AutoScalingGroup	MinSize	The minimum value for the DesiredCapacity	Any positive integer. Use 1 if you want a single virtual machine to be started based on the launch template.
AutoScalingGroup	MaxSize	The maximum value for the DesiredCapacity	Any positive integer (greater than or equal to the MinSize value). Use 1 if you want a single virtual machine to be started based on the launch template.
AutoScalingGroup	VPCZoneIdentifier	The subnet IDs you want to start virtual machines in	Any subnet ID from a VPC from your account. Subnets must belong to the same VPC.
AutoScalingGroup	HealthCheckType	The health check used to identify failed virtual machines. If the health check fails, the auto-scaling group replaces the virtual machine with a new one.	EC2 to use the status checks of the virtual machine, or ELB to use the health check of the load balancer (see chapter 16).

There is one important difference between the definition of a single EC2 instance and the launch template: the subnet for the virtual machine isn't defined in the launch template, but rather in the auto-scaling group.

An auto-scaling group is also used if you need to scale the number of virtual machines based on usage of your system. You'll learn how to scale the number of EC2 instances based on current load in chapter 17. In this chapter, you only need to make sure a single virtual machine is always running. Because you need a single virtual machine, set the following parameters for auto-scaling to 1:

- MinSize
- MaxSize

Listing 13.3 Launching a Jenkins virtual machine with auto-scaling in two AZs

```

# [...]
LaunchTemplate:
  Type: 'AWS::EC2::LaunchTemplate' ①
  Properties:
    LaunchTemplateData:
      IamInstanceProfile: ②
        Name: !Ref IamInstanceProfile
      ImageId: !FindInMap [RegionMap, !Ref 'AWS::Region', AMI] ③
      Monitoring: ④
        Enabled: false
      InstanceType: 't2.micro' ⑤
      NetworkInterfaces:
        - AssociatePublicIpAddress: true ⑦
          DeviceIndex: 0
        Groups:
          - !Ref SecurityGroup ⑧
      UserData:
        'Fn::Base64': !Sub |
          #!/bin/bash -ex
          trap '/opt/aws/bin/cfn-signal -e 1 --stack ${AWS::StackName} \
            --resource AutoScalingGroup --region ${AWS::Region}' ERR

        # Installing Jenkins
        amazon-linux-extras enable epel=7.11 && yum -y clean metadata
        yum install -y epel-release && yum -y clean metadata
        yum install -y java-11-amazon-corretto-headless daemonize
        wget -q -T 60 http://ftp-chi.osuosl.org/pub/jenkins/
        ↪redhat-stable/jenkins-2.319.1-1.1.noarch.rpm
        rpm --install jenkins-2.319.1-1.1.noarch.rpm

        # Configuring Jenkins
        # [...]

        # Starting Jenkins
        systemctl enable jenkins.service
        systemctl start jenkins.service
        /opt/aws/bin/cfn-signal -e $? --stack ${AWS::StackName} \
          --resource AutoScalingGroup --region ${AWS::Region}

AutoScalingGroup:
  Type: 'AWS::AutoScaling::AutoScalingGroup' ⑩
  Properties:
    LaunchTemplate: ⑪
      LaunchTemplateId: !Ref LaunchTemplate
      Version: !GetAtt 'LaunchTemplate.LatestVersionNumber'
    Tags: ⑫
      - Key: Name
        Value: 'jenkins-multiaz'
        PropagateAtLaunch: true
    MinSize: 1 ⑬
    MaxSize: 1 ⑭
    VPCZoneIdentifier: ⑮
      - !Ref SubnetA
      - !Ref SubnetB
    HealthCheckGracePeriod: 600 ⑯
    HealthCheckType: EC2 ⑰
# [...]

```

- ① The blueprint used by the auto-scaling group when launching an EC2 instance
- ② Attaches an IAM role to the EC2 instance to grant access for Session Manager
- ③ Select the AMI (in this case Amazon Linux 2).

- ④ By default, EC2 sends metrics to CloudWatch every 5 minutes. But you can enable detailed instance monitoring to get metrics every minute for an additional cost.
- ⑤ The instance type for the virtual machine
- ⑥ Configures the network interface (ENI) of the EC2 instance
- ⑦ Associate a public IP address when launching the instance
- ⑧ Attach a security group allowing ingress on port 8080 to the instance
- ⑨ The EC2 instance will execute the script loaded from user data at the end of the boot process, the script installs and configures Jenkins
- ⑩ Auto-scaling group responsible for launching the virtual machine
- ⑪ Reference the launch template
- ⑫ Add those tags to the auto-scaling group itself as well as to all EC2 instances launched by the auto-scaling group
- ⑬ Minimum number of EC2 instances
- ⑭ Maximum number of EC2 instances
- ⑮ Launches the virtual machines in subnet A (created in availability zone A) and subnet B (created in availability zone B)
- ⑯ Wait 10 minutes before considering the health check of a newly launched instance
- ⑰ Uses the internal health check of the EC2 service to discover issues with the virtual machine

The CloudFormation stack might be already up and running. Execute the following command to grab the public IP address of the virtual machine. If no IP address appears, the virtual machine isn't started yet. Wait another minute, and try again:

```
$ aws ec2 describe-instances --filters "Name=tag:Name,\n"
  Values=jenkins-multiaz" "Name=instance-state-code,Values=16" \
  --query "Reservations[0].Instances[0].\
  [InstanceId, PublicIpAddress, PrivateIpAddress, SubnetId]"
[
  "i-0cff527cda42afbcc",      ①
  "34.235.131.229",          ②
  "172.31.38.173",            ③
  "subnet-28933375"           ④
]
```

- ① Instance ID of the virtual machine
- ② Public IP address of the virtual machine
- ③ Private IP address of the virtual machine
- ④ Subnet ID of the virtual machine

Open `http://$PublicIP:8080` in your browser, and replace `$PublicIP` with the public IP address

from the output of the previous `describe-instances` command. The web interface for the Jenkins server appears.

Execute the following command to terminate the virtual machine and test the recovery process with auto-scaling. Replace `$InstanceId` with the instance ID from the output of the previous `describe` command:

```
$ aws ec2 terminate-instances --instance-ids $InstanceId
```

After a few minutes, the auto-scaling group detects that the virtual machine was terminated and starts a new virtual machine. Rerun the `describe-instances` command until the output contains a new running virtual machine:

```
$ aws ec2 describe-instances --filters "Name=tag:Name,\n  Values=jenkins-multiaz" "Name=instance-state-code,Values=16" \
  --query "Reservations[0].Instances[0].\n  [InstanceId, PublicIpAddress, PrivateIpAddress, SubnetId]"
[
  "i-0293522fad287bdd4",
  "52.3.222.162",
  "172.31.37.78",
  "subnet-45b8c921"
]
```

The instance ID, the public IP address, the private IP address, and probably even the subnet ID have changed for the new instance. Open `http://$PublicIP:8080` in your browser, and replace `$PublicIP` with the public IP address from the output of the previous `describe-instances` command. The web interface from the Jenkins server appears.

You've now built a highly available architecture consisting of an EC2 instance with the help of auto-scaling. There are two issues with the current setup:

- The Jenkins server stores data on disk. When a new virtual machine is started to recover from a failure, this data is lost because a new disk is created.
- The public and private IP addresses of the Jenkins server change after a new virtual machine is started for recovery. The Jenkins server is no longer available under the same endpoint.

You'll learn how to solve these problems in the next part of the chapter.

SIDE BAR**Cleaning up**

It's time to clean up to avoid unwanted costs. Execute the following command to delete all resources corresponding to the Jenkins setup:

```
$ aws cloudformation delete-stack --stack-name jenkins-multiaz
$ aws cloudformation wait stack-delete-complete \
  --stack-name jenkins-multiaz ①
```

① Waits until the stack is deleted

13.2.3 Pitfall: recovering network-attached storage

The EBS service offers network-attached storage for virtual machines. Remember that EC2 instances are linked to a subnet, and the subnet is linked to an availability zone. EBS volumes are also located in a single availability zone only. If your virtual machine is started in another availability zone because of an outage, the EBS volume cannot be accessed from the other availability zone. Let's say your Jenkins data is stored on an EBS volume in availability zone `us-east-1a`. As long as you have an EC2 instance running in the same availability zone, you can attach the EBS volume. But if this availability zone becomes unavailable, and you start a new EC2 instance in availability zone `us-east-1b`, you can't access that EBS volume in `us-east-1a`, which means that you can't recover Jenkins because you don't have access to the data. See figure [13.5](#).

SIDE BAR**Don't mix availability and durability guarantees**

An EBS volume is guaranteed to be available for 99.999% of the time. So in case of an availability zone outage, the volume is no longer available. This does not imply that you lose any data. As soon as the availability zone is back online, you can access the EBS volume again with all its data.

An EBS volume guarantees that you won't lose any data in 99.9% of the time. This guarantee is called the durability of the EBS volume. If you have 1,000 volumes in use, you can expect that you will lose one of the volumes and its data a year.

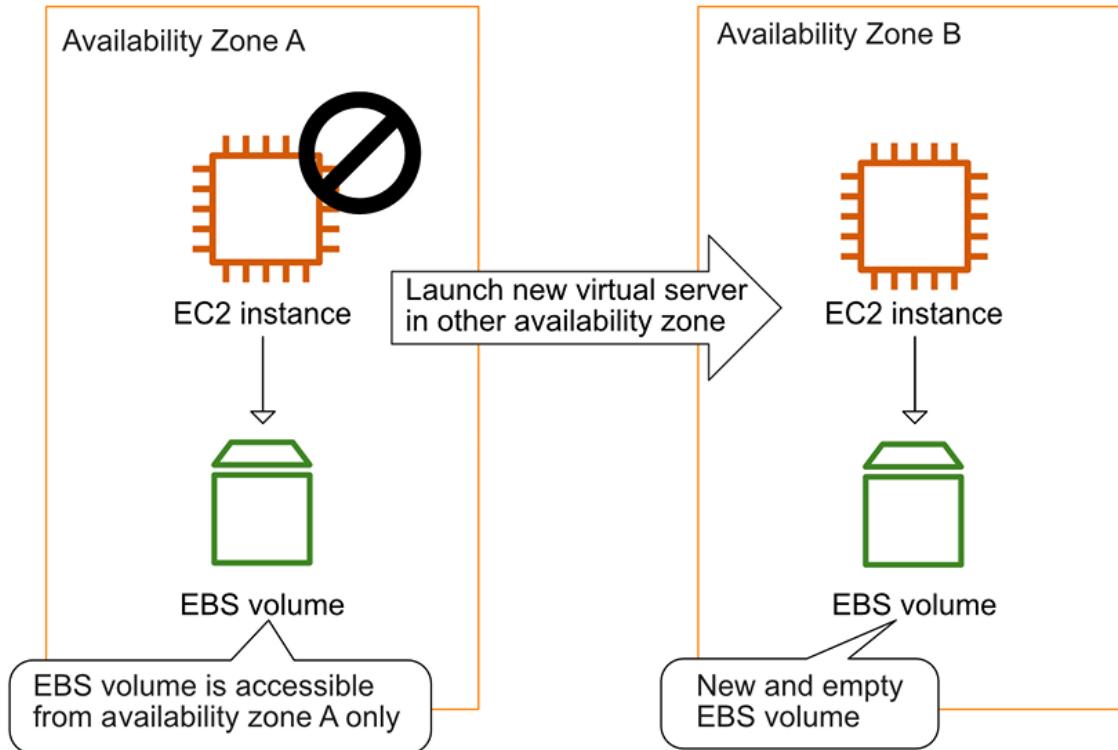


Figure 13.5 An EBS volume is only available in a single availability zone.

There are multiple solutions for this problem:

1. Outsource the state of your virtual machine to a managed service that uses multiple availability zones by default: RDS, DynamoDB (NoSQL database), EFS (NFSv4.1 share), or S3 (object store).
2. Create snapshots of your EBS volumes regularly, and use these snapshots if an EC2 instance needs to recover in another availability zone. EBS snapshots are stored on S3, thus available in multiple availability zones. If the EBS volume is the root volume of the ECS instance, create AMIs to back up the EBS volume instead of a snapshot.
3. Use a distributed third-party storage solution to store your data in multiple availability zones: GlusterFS, DRBD, MongoDB, and so on.

The Jenkins server stores data directly on disk. To outsource the state of the virtual machine, you can't use RDS, DynamoDB, or S3; you need a file-level storage solution instead. As you've learned, an EBS volume is only available in a single availability zone, so this isn't the best fit for the problem. But do you remember EFS from chapter 10? EFS provides network file storage (over NFSv4.1) and replicates your data automatically between availability zones in a region.

To embed EFS into the Jenkins setup, shown in listing 13.4, you have to make three modifications to the Multi-AZ template from the previous section:

1. Create an EFS filesystem.
2. Create EFS mount targets in each availability zone.
3. Adjust the user data to mount the EFS filesystem. Jenkins stores all its data under `/var/lib/jenkins`.

Listing 13.4 Store Jenkins state on EFS

```

# [...]
FileSystem: ①
  Type: 'AWS::EFS::FileSystem'
  Properties: {}
MountTargetSecurityGroup: ②
  Type: 'AWS::EC2::SecurityGroup'
  Properties:
    GroupDescription: 'EFS Mount target'
    SecurityGroupIngress:
      - FromPort: 2049 ③
        IpProtocol: tcp
        SourceSecurityGroupId: !Ref SecurityGroup
        ToPort: 2049
    VpcId: !Ref VPC
MountTargetA: ④
  Type: 'AWS::EFS::MountTarget'
  Properties:
    FileSystemId: !Ref FileSystem
    SecurityGroups:
      - !Ref MountTargetSecurityGroup
    SubnetId: !Ref SubnetA ⑤
MountTargetB: ⑥
  Type: 'AWS::EFS::MountTarget'
  Properties:
    FileSystemId: !Ref FileSystem
    SecurityGroups:
      - !Ref MountTargetSecurityGroup
    SubnetId: !Ref SubnetB
# [...]
LaunchTemplate:
  Type: 'AWS::EC2::LaunchTemplate' ⑦
  Properties:
    LaunchTemplateData:
      # [...]
      UserData:
        'Fn::Base64': !Sub |
          #!/bin/bash -ex
          trap '/opt/aws/bin/cfn-signal -e 1 --stack ${AWS::StackName}' ERR
          #--resource AutoScalingGroup --region ${AWS::Region}' ERR
          # Installing Jenkins
          # [...]
          # Mounting EFS volume
          mkdir -p /var/lib/jenkins ⑧
          echo "${FileSystem}:/ /var/lib/jenkins efs tls,_netdev 0 0"
          #>>> /etc/fstab ⑨
          while ! (echo > /dev/tcp/${FileSystem}.efs.${AWS::Region}...
            #>>> amazonaws.com/2049) >/dev/null 2>&1; do sleep 5; done ⑩
          mount -a -t efs ⑪
          chown -R jenkins:jenkins /var/lib/jenkins ⑫
          # Configuring Jenkins
          # [...]
          # Starting Jenkins
          systemctl enable jenkins.service
          systemctl start jenkins.service
          /opt/aws/bin/cfn-signal -e $? --stack ${AWS::StackName}
          #--resource AutoScalingGroup --region ${AWS::Region}
AutoScalingGroup:
  Type: 'AWS::AutoScaling::AutoScalingGroup' ⑬
  Properties:
    LaunchTemplate: ⑭
      LaunchTemplateId: !Ref LaunchTemplate
      Version: !GetAtt 'LaunchTemplate.LatestVersionNumber'
    Tags:
      - Key: Name
        Value: 'jenkins-multiaz-efs'

```

```

    PropagateAtLaunch: true
    MinSize: 1
    MaxSize: 1
    VPCZoneIdentifier:
      - !Ref SubnetA
      - !Ref SubnetB
    HealthCheckGracePeriod: 600
    HealthCheckType: EC2
# [...]

```

- ① Creates an Elastic File System (EFS) which provides a NFS (network file system)
- ② Creates a security group used to grant network traffic from the EC2 instance to EFS
- ③ Allows incoming traffic on port 2049 used by NFS
- ④ The mount target provides a network interface for the file system
- ⑤ The mount target is attached to a subnet
- ⑥ Therfore, you need a mount target per subnet
- ⑦ The blueprint used by the auto-scaling group to launch virtual machines
- ⑧ Create folder used by Jenkins to store data if it does not exist yet
- ⑨ Add an entry to the configuration file for volumes
- ⑩ Wait until EFS file system becomes available
- ⑪ Mount the EFS file system
- ⑫ Change the ownership of the mounted directory to make sure Jenkins is able to write and read files
- ⑬ Create the auto-scaling group
- ⑭ References the launch template defined above

Execute the following command to create the new Jenkins setup that stores state on EFS. Replace \$Password with a password consisting of 8-40 characters.

```

$ aws cloudformation create-stack --stack-name jenkins-multiaz-efs \
  --template-url https://s3.amazonaws.com/\
  awsinaction-code3/chapter14/multiaz-efs.yaml \
  --parameters "ParameterKey=JenkinsAdminPassword,
  ParameterValue=$Password" \
  --capabilities CAPABILITY_IAM

```

The creation of the CloudFormation stack will take a few minutes. Run the following command to get the public IP address of the virtual machine. If no IP address appears, the virtual machine isn't started yet. In this case, please wait another minute, and try again:

```
$ aws ec2 describe-instances --filters "Name=tag:Name,\n"
  Values=jenkins-multiaz-efs" "Name=instance-state-code,Values=16" \
  --query "Reservations[0].Instances[0].\n
  [InstanceId, PublicIpAddress, PrivateIpAddress, SubnetId]"
[
  "i-0efcd2f01a3e3af1d",      ①
  "34.236.255.218",          ②
  "172.31.37.225",           ③
  "subnet-0997e66d"           ④
]
```

- ① Instance ID of the virtual machine
- ② Public IP address of the virtual machine
- ③ Private IP address of the virtual machine
- ④ Subnet ID of the virtual machine

Next, create a new Jenkins job by following these steps:

1. Open `http://$PublicIP:8080/newJob` in your browser, and replace `$PublicIP` with the public IP address from the output of the previous `describe` command.
2. Log in with user `admin` and the password you chose when starting the CloudFormation template.
3. Select the `Install selected plugins` option.
4. Keep the default for `Jenkins URL` and click `Save` and `Finish`.
5. Click `Start using Jenkins`.
6. Click `New Item` to create a new project.
7. Type in `AWS` in `Action` as the name for the new project.
8. Select `Freestyle project` as the job type, and click `OK`.

You've made some changes to the state of Jenkins stored on EFS. Now, terminate the EC2 instance with the following command and you will see that Jenkins recovers from the failure without data loss. Replace `$InstanceId` with the instance ID from the output of the previous `describe` command:

```
$ aws ec2 terminate-instances --instance-ids $InstanceId
```

After a few minutes, the auto-scaling group detects that the virtual machine was terminated and starts a new virtual machine. Rerun the `describe-instances` command until the output contains a new running virtual machine:

```
$ aws ec2 describe-instances --filters "Name=tag:Name,\n"
  Values=jenkins-multiaz-efs" "Name=instance-state-code,Values=16" \
  --query "Reservations[0].Instances[0].\n
  [InstanceId, PublicIpAddress, PrivateIpAddress, SubnetId]"
[
  "i-07ce0865adf50cccf",
  "34.200.225.247",
  "172.31.37.199",
  "subnet-0997e66d"
]
```

The instance ID, the public IP address, the private IP address, and probably even the subnet ID have changed for the new instance. Open `http://$PublicIP:8080` in your browser, and replace `$PublicIP` with the public IP address from the output of the previous `describe-instances` command. The web interface from the Jenkins server appears and it still contains the `AWS in Action` job you created recently.

You've now built a highly available architecture consisting of an EC2 instance with the help of auto-scaling. State is now stored on EFS and is no longer lost when an EC2 instance is replaced. There is one issue left:

- The public and private IP addresses of the Jenkins server change after a new virtual machine is started for recovery. The Jenkins server is no longer available under the same endpoint.

SIDE BAR Cleaning up 

It's time to clean up to avoid unwanted costs. Execute the following command to delete all resources corresponding to the Jenkins setup:

```
$ aws cloudformation delete-stack --stack-name jenkins-multiaz-efs
$ aws cloudformation wait stack-delete-complete \
  --stack-name jenkins-multiaz-efs
```

① Waits until the stack is deleted

You'll learn how to solve the last issue next.

13.2.4 Pitfall: network interface recovery

Recovering a virtual machine using a CloudWatch alarm in the same availability zone, as described at the beginning of this chapter, is easy because the private IP address and the public IP address stay the same automatically. You can use these IP addresses as an endpoint to access the EC2 instance even after a failover.

When it comes to creating a virtual network in the cloud (VPC) you need to be aware about the following links:

- A VPC is always bound to a region.
- A subnet within a VPC is linked to an availability zone.
- A virtual machine is launched into a single subnet.

Figure [13.6](#) illustrates these dependencies.

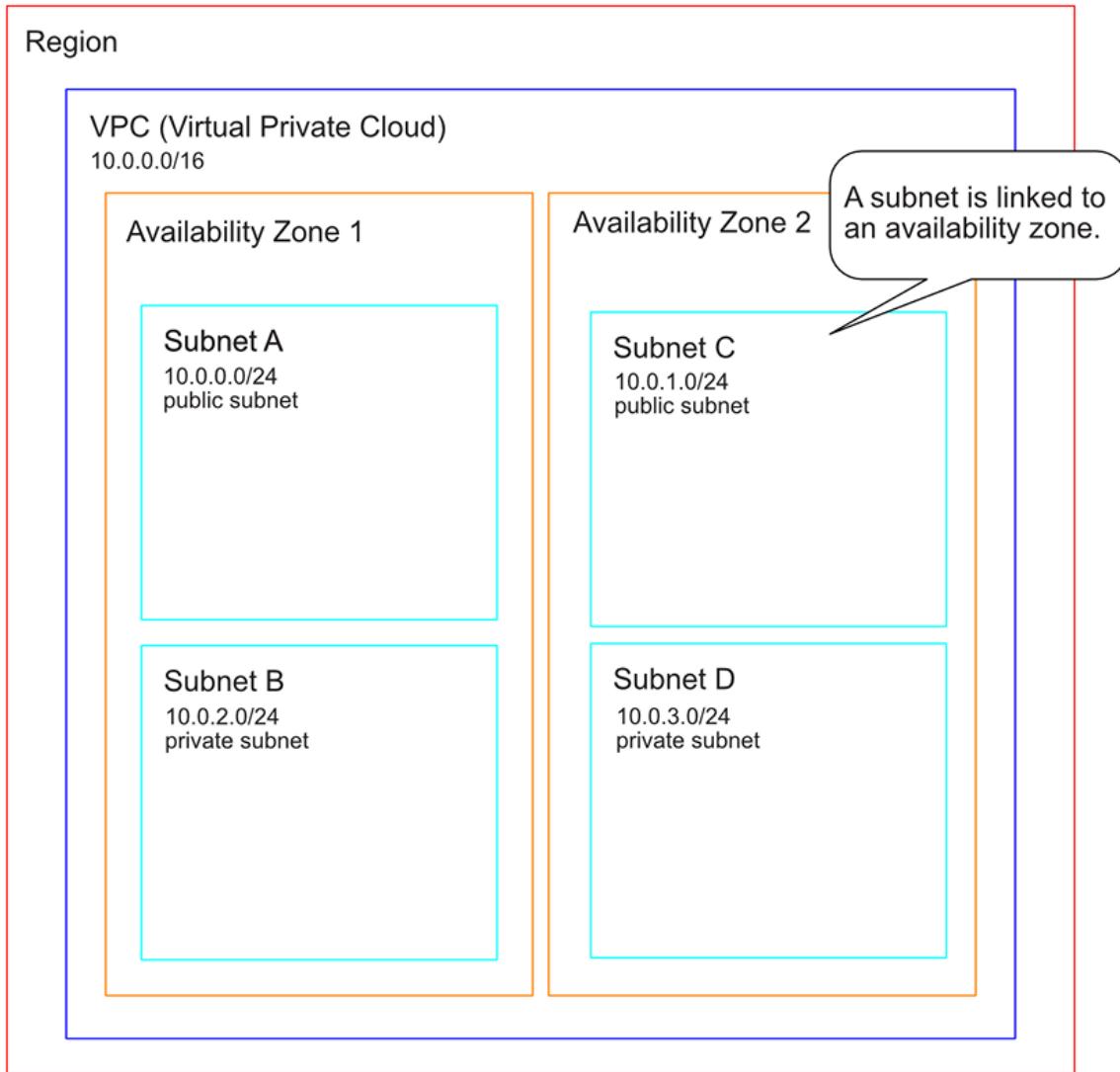


Figure 13.6 A VPC is bound to a region, and a subnet is linked to an availability zone.

Therefore, you can't keep the private IP address when using auto-scaling to recover from a EC2 instance or availability zone outage. If a virtual machine has to be started in another availability zone, it must be started in another subnet. So it's not possible to use the same private IP address for the new virtual machine, as figure [14.7](#) shows.

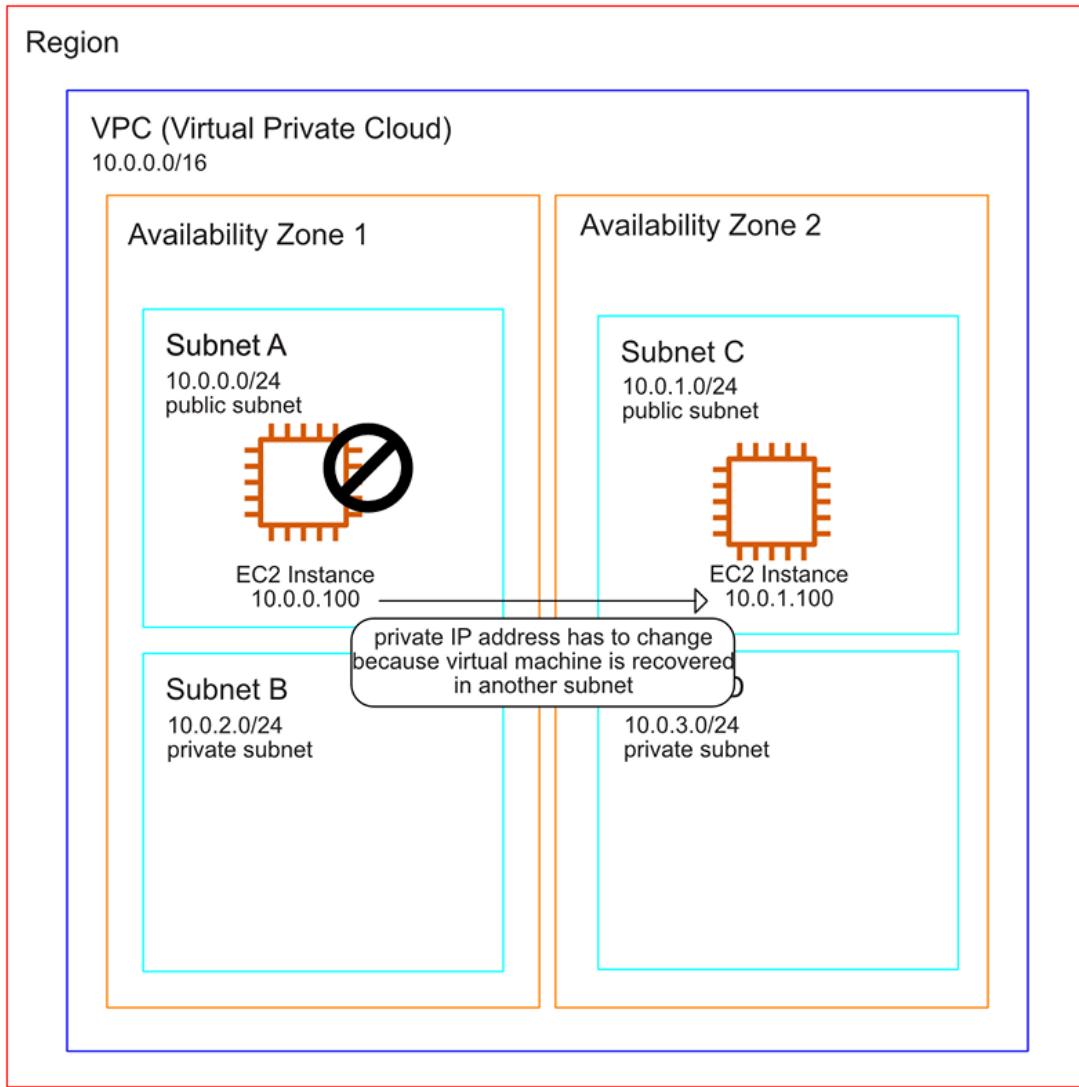


Figure 13.7 The virtual machine starts in another subnet in case of a failover and changes the private IP address.

By default, you also can't use an Elastic IP as a public IP address for a virtual machine launched by auto-scaling. But the requirement for a static endpoint to receive requests is common. For the use case of a Jenkins server, developers want to bookmark an IP address or a hostname to reach the web interface. There are different possibilities for providing a static endpoint when using auto-scaling to build high availability for a single virtual machine:

- Allocate an Elastic IP, and associate this public IP address during the bootstrap of the virtual machine.
- Create or update a DNS entry linking to the current public or private IP address of the virtual machine.
- Use an Elastic Load Balancer (ELB) as a static endpoint that forwards requests to the current virtual machine.

To use the second solution, you need to link a domain with the Route 53 (DNS) service; we've chosen to skip this solution because you need a registered domain to implement it. The ELB solution is covered in chapter 15, so we'll skip it in this chapter as well. We'll focus on the first solution: allocating an Elastic IP and associating this public IP address during the virtual machine's bootstrap.

Execute the following command to create the Jenkins setup based on auto-scaling again, using an Elastic IP address as static endpoint:

```
$ aws cloudformation create-stack --stack-name jenkins-multiaz-efs-eip \
→--template-url https://s3.amazonaws.com/\
→awsinaction-code3/chapter14/multiaz-efs-eip.yaml \
→--parameters "ParameterKey=JenkinsAdminPassword,
→ParameterValue=$Password" \
→--capabilities CAPABILITY_IAM
```

The command creates a stack based on the template shown in listing 13.5. The differences from the original template spinning up a Jenkins server with auto-scaling are as follows:

- Allocating an Elastic IP
- Adding the association of an Elastic IP to the script in the user data
- Creating an IAM role and policy to allow the EC2 instance to associate an Elastic IP

Listing 13.5 Using an EIP as a static endpoint for a virtual machine launched by auto-scaling

```

# [...]
ElasticIP:
  Type: 'AWS::EC2::EIP'    ①
  Properties:
    Domain: vpc
    DependsOn: VPCGatewayAttachment
IamRole:
  Type: 'AWS::IAM::Role'    ②
  Properties:
    AssumeRolePolicyDocument:
      Version: '2012-10-17'
      Statement:
        - Effect: Allow
          Principal:
            Service: 'ec2.amazonaws.com'    ③
          Action: 'sts:AssumeRole'
    Policies:
      - PolicyName: ec2
        PolicyDocument:
          Version: '2012-10-17'
          Statement:
            - Action: 'ec2:AssociateAddress'    ④
              Resource: '*'
              Effect: Allow
      - PolicyName: ssm    ⑤
        PolicyDocument:
          Version: '2012-10-17'
          Statement:
            - Effect: Allow
              Action:
                - 'ssmmessages:/*'
                - 'ssm:UpdateInstanceInformation'
                - 'ec2messages:/*'
              Resource: '*'
IamInstanceProfile:
  Type: 'AWS::IAM::InstanceProfile'    ⑥
  Properties:
    Roles:
      - !Ref IamRole
LaunchTemplate:
  Type: 'AWS::EC2::LaunchTemplate'    ⑦
  Properties:
    LaunchTemplateData:
      IamInstanceProfile:
        Name: !Ref IamInstanceProfile    ⑧
      ImageId: !FindInMap [RegionMap, !Ref 'AWS::Region', AMI]
      Monitoring:
        Enabled: false
      InstanceType: 't2.micro'
      NetworkInterfaces:
        - AssociatePublicIpAddress: true
          DeviceIndex: 0
          Groups:
            - !Ref SecurityGroup
      UserData:
        'Fn::Base64': !Sub |
          #!/bin/bash -ex
          trap '/opt/aws/bin/cfn-signal -e 1 --stack ${AWS::StackName}' ERR
          ➔--resource AutoScalingGroup --region ${AWS::Region}' ERR
          # Attaching EIP
          INSTANCE_ID="$(curl
          ➔-s http://169.254.169.254/latest/meta-data/instance-id)"    ⑨
          aws --region ${AWS::Region} ec2 associate-address
          ➔--instance-id $INSTANCE_ID

```

```

↳--allocation-id ${ElasticIP.AllocationId}
↳--allow-reassociation ⑩
sleep 30
# Installing Jenkins [...]
# Mounting EFS volume [...]
# Configuring Jenkins [...]
# Starting Jenkins
systemctl enable jenkins.service
systemctl start jenkins.service
/opt/aws/bin/cfn-signal -e $? --stack ${AWS::StackName}
--resource AutoScalingGroup --region ${AWS::Region}

```

- ① Creates a static public IP address
- ② Creates an IAM role granting access to AWS services to the EC2 instance
- ③ The IAM role can only be used by EC2 instances
- ④ The IAM policy allows access to the EC2 API action called `AssociateAddress` which is used to associate an Elastic IP with an EC2 instance
- ⑤ The other IAM policy enables access to the Session Manager enabling you to open a terminal connection with the EC2 instance
- ⑥ An IAM instance profile is needed to be able to attach an IAM role to an EC2 instance
- ⑦ The launch template defines the blueprint for launching the EC2 instance
- ⑧ Attach the IAM instance profile defined when starting the virtual machine
- ⑨ Get the ID of the running instance from the metadata service (see <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/instancedata-data-retrieval.html> for details)
- ⑩ The EC2 instance associates the Elastic IP address with itself by using the AWS CLI

If the query returns output as shown in the following listing, containing a URL, a user, and a password, the stack has been created and the Jenkins server is ready to use. Open the URL in your browser, and log in to the Jenkins server with user `admin` and the password you've chosen. If the output is `null`, try again in a few minutes:

```
$ aws cloudformation describe-stacks --stack-name jenkins-multiaz-efs-eip \
↳ --query "Stacks[0].Outputs"
```

You can now test whether the recovery of the virtual machine works as expected. To do so, you'll need to know the instance ID of the running virtual machine. Run the following command to get this information:

```
$ aws ec2 describe-instances --filters "Name>tag:Name, \
↳ Values=jenkins-multiaz-efs-eip" "Name=instance-state-code,Values=16" \
↳ --query "Reservations[0].Instances[0].InstanceId" --output text
```

Execute the following command to terminate the virtual machine and test the recovery process triggered by auto-scaling. Replace `$InstanceId` with the instance from the output of the previous command:

```
$ aws ec2 terminate-instances --instance-ids $InstanceId
```

Wait a few minutes for your virtual machine to recover. Because you're using an Elastic IP

Execute the following command to terminate the virtual machine and test the recovery process triggered by auto-scaling. Replace `$InstanceId` with the instance from the output of the previous command:

```
$ aws ec2 terminate-instances --instance-ids $InstanceId
```

Wait a few minutes for your virtual machine to recover. Because you're using an Elastic IP assigned to the new virtual machine on bootstrap, you can open the same URL in your browser as you did before the termination of the old instance.

SIDE BAR

Cleaning up



It's time to clean up to avoid unwanted costs. Execute the following command to delete all resources corresponding to the Jenkins setup:

```
$ aws cloudformation delete-stack --stack-name jenkins-multiaz-efs-eip
$ aws cloudformation wait stack-delete-complete \
  --stack-name jenkins-multiaz-efs-eip ①
```

① Waits until the stack is deleted

Now the public IP address of your virtual machine running Jenkins won't change, even if the running virtual machine needs to be replaced by another virtual machine in another availability zone.

Last but not least, we want to come back to the concept of an availability zone and dive into some of the details.

13.2.5 Insights into availability zones

A region consists of multiple availability zones. Each availability zone consists of at least one isolated data center. The identifier for an availability zone consists of the identifier for the region (such as us-east-1) and a character (a, b, c, ...). So us-east-1a is the identifier for an availability zone in region us-east-1. To distribute resources across the different availability zones, the AZ identifier is mapped to one or multiple data centers randomly when creating an AWS account. This means us-east-1a points might point to a different availability zone in your AWS account than it does in our AWS account.

We recommend, that you take some time to explore the worldwide infrastructure provided by AWS. You can use the following commands to discover all regions available for your AWS account:

```
$ aws ec2 describe-regions ①
{
  "Regions": [
    {
      "Endpoint": "ec2.eu-north-1.amazonaws.com", ②
      "RegionName": "eu-north-1", ③
      "OptInStatus": "opt-in-not-required" ④
    },
    {
      "Endpoint": "ec2.ap-south-1.amazonaws.com",
      "RegionName": "ap-south-1",
      "OptInStatus": "opt-in-not-required"
    },
    [...]
    {
      "Endpoint": "ec2.us-west-2.amazonaws.com",
      "RegionName": "us-west-2",
      "OptInStatus": "opt-in-not-required"
    }
  ]
}
```

- ① Lists all regions available for your AWS account.
- ② The endpoint URL, used to access the EC2 service in the region.
- ③ The name of the region.
- ④ Newer regions require an opt-in.

Next, to list all availability zones for a region, execute the following command and replace `$Region` with `RegionName` of a region from the previous output:

```
$ aws ec2 describe-availability-zones --region $Region ①
{
  "AvailabilityZones": [
    {
      "State": "available",
      "OptInStatus": "opt-in-not-required",
      "Messages": [],
      "RegionName": "us-east-1", ②
      "ZoneName": "us-east-1a", ③
      "ZoneId": "use1-az1", ④
      "GroupName": "us-east-1",
      "NetworkBorderGroup": "us-east-1",
      "ZoneType": "availability-zone"
    },
    {
      "State": "available",
      "OptInStatus": "opt-in-not-required",
      "Messages": [],
      "RegionName": "us-east-1",
      "ZoneName": "us-east-1b",
      "ZoneId": "use1-az2",
      "GroupName": "us-east-1",
      "NetworkBorderGroup": "us-east-1",
      "ZoneType": "availability-zone"
    },
    [...]
    {
      "State": "available",
      "OptInStatus": "opt-in-not-required",
      "Messages": [],
      "RegionName": "us-east-1",
      "ZoneName": "us-east-1f",
      "ZoneId": "use1-az5",
      "GroupName": "us-east-1",
      "NetworkBorderGroup": "us-east-1",
      "ZoneType": "availability-zone"
    }
  ]
}
```

- ① List the availability zones of a region.
- ② The region name.
- ③ The name of the availability zone might point to different data centers in different AWS accounts.
- ④ The ID of the availability zone points to the same data centers in all AWS accounts.

At the end of the chapter, you will learn about how to analyze resilience requirements and derive an AWS architecture from the results.

13.3 Architecting for high availability

Before you begin implementing highly available or even fault-tolerant architectures on AWS, you should start by analyzing your disaster-recovery requirements. Disaster recovery is easier and cheaper in the cloud than in a traditional data center, but building for high availability increases the complexity and therefore the initial costs as well as the operating costs of your system. The recovery time objective (RTO) and recovery point objective (RPO) are standards for defining the importance of disaster recovery from a business point of view.

Recovery time objective (RTO) is the time it takes for a system to recover from a failure; it's the length of time until the system reaches a working state again, defined as the system service level, after an outage. In the example with a Jenkins server, the RTO would be the time until a new virtual machine is started and Jenkins is installed and running after a virtual machine or an entire availability zone goes down.

Recovery point objective (RPO) is the acceptable data-loss time caused by a failure. The amount of data loss is measured in time. If an outage happens at 10:00 a.m. and the system recovers with a data snapshot from 09:00 a.m., the time span of the data loss is one hour. In the example of a Jenkins server using auto-scaling, the RPO would be zero, because data is stored on EFS and is not lost during an AZ outage. Figure 13.8 illustrates the definitions of RTO and RPO.

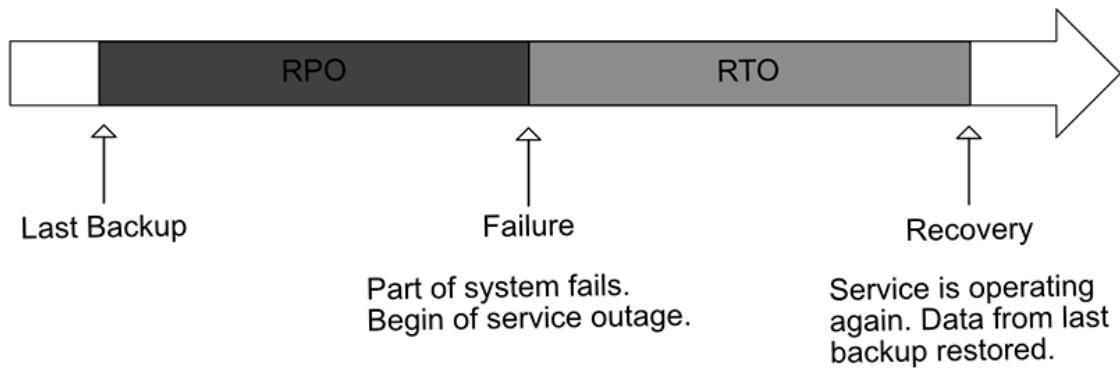


Figure 13.8 Definitions of RTO and RPO

13.3.1 RTO and RPO comparison for a single EC2 instance

You've learned about two possible solutions for making a single EC2 instance highly available. When choosing the solution, you have to know the application's business requirements. Can you tolerate the risk of being unavailable if an availability zone goes down? If so, EC2 instance recovery is the simplest solution where you don't lose any data. If your application needs to survive an unlikely availability zone outage, your safest bet is auto-scaling with data stored on EFS. But this also has performance impacts compared to storing data on EBS volumes. As you can see, there is no one-size-fits-all solution. You have to pick the solution that fits your business problem best. Table 13.2 compares the solutions.

Table 13.2 Comparison of high availability for a single EC2 instance

	RTO	RPO	Availability
EC2 instance, data stored on EBS root volume: recovery triggered by a CloudWatch alarm	About 10 minutes	No data loss	Recover from a failure of a virtual machine but not from an outage of an entire availability zone
EC2 instance, data stored on EBS root volume: recovery triggered by auto-scaling	About 10 minutes	All data is lost.	Recover from a failure of a virtual machine and from an outage of an entire availability zone
EC2 instance, data stored on EBS root volume with regular snapshots: recovery triggered by auto-scaling	About 10 minutes	Realistic time span for snapshots is between 30 minutes and 24 hours.	Recover from a failure of a virtual machine and from an outage of an entire availability zone
EC2 instance, data stored on EFS filesystem: recovery triggered by auto-scaling	About 10 minutes	No data loss.	Recover from a failure of a virtual machine and from an outage of an entire availability zone

If you want to be able to recover from an outage of an availability zone and need to decrease the RPO, you should try to achieve a stateless server. Using storage services like RDS, EFS, S3, and DynamoDB can help you to do so. See part 3 if you need help with using these services.

13.3.2 AWS services come with different high availability guarantees

It is important to note, that some AWS services are highly available or even fault-tolerant by default. Other services provide building blocks to achieve a highly available architecture. You can use multiple availability zones or even multiple regions to build a highly available architecture, as figure 13.9 shows:

- Route 53 (DNS) and CloudFront (CDN) operate globally over multiple regions and are highly available by default.
- S3 (object store), EFS (network file system) and DynamoDB (NoSQL database) use multiple availability zones within a region so they can recover from an availability zone outage
- The Relational Database Service (RDS) offers the ability to deploy a primary-standby setup, called *Multi-AZ deployment*, so you can fail over into another availability zone with a short downtime if necessary.

- A virtual machine runs in a single availability zone. But AWS offers tools to build an architecture based on EC2 instances that can fail over into another availability zone.

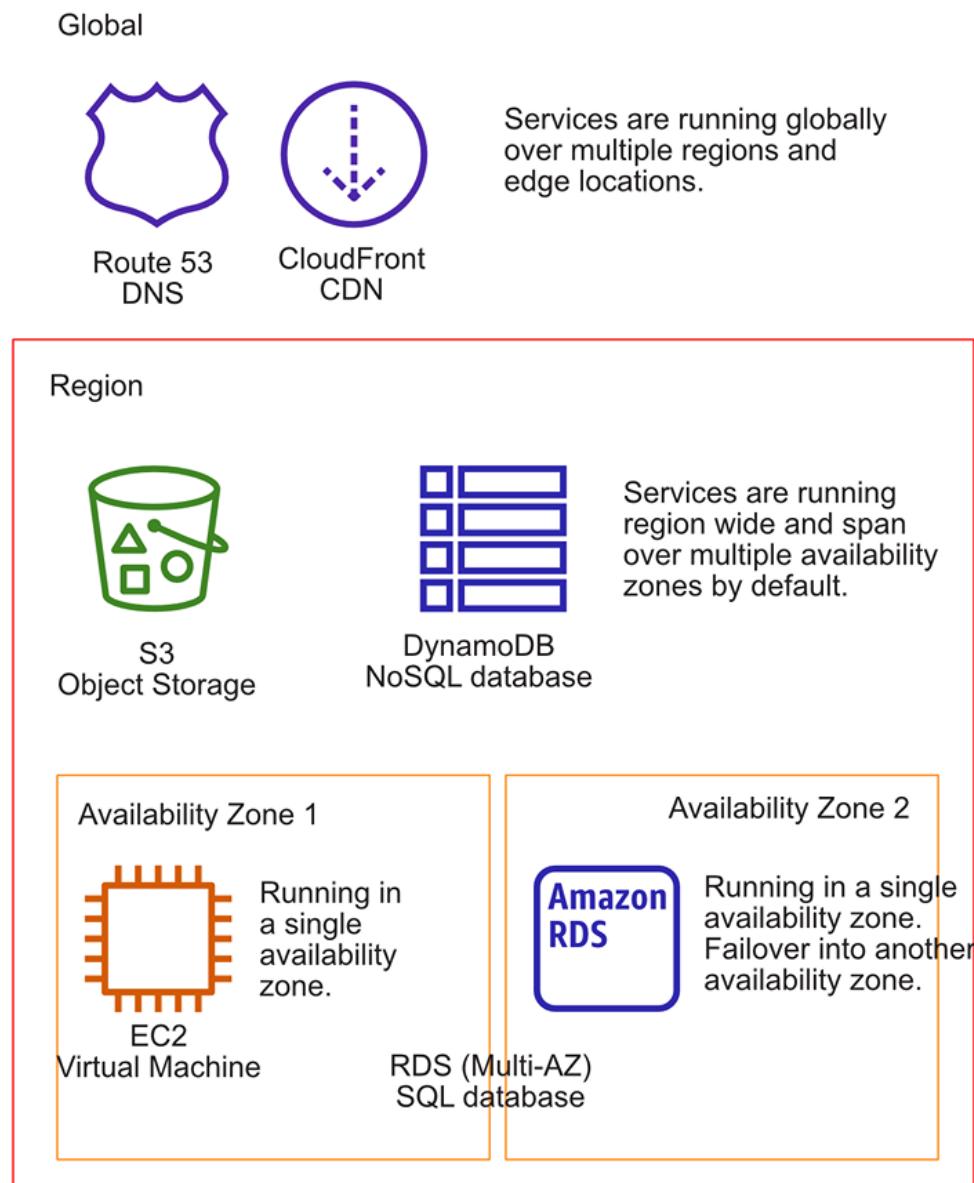


Figure 13.9 AWS services can operate in a single availability zone, over multiple availability zones within a region, or even globally.

When planning for failure it is also important to consider the service level objective (SLO) and service level agreement (SLA) committed by AWS. Most services define an SLA, that helps you as a customer when estimating the availability of an architecture. You can read them here [AWS Service Level Agreements](#).

So when designing a system for AWS, you need to look into the SLA and resilience specifications of each building block. To do so, check the AWS documentation which includes a section on *Resilience* for most services.

13.4 Summary

- A virtual machine fails if the underlying hardware or virtualization layer fails.
- You can recover a failed virtual machine with the help of a CloudWatch alarm: by default, data stored on EBS, as well as the private and public IP addresses, stays the same.
- An AWS region consists of multiple isolated groups of data centers called availability zones.
- Recovering from a data center outage is possible when using multiple availability zones.
- You can use auto-scaling to guarantee that a single virtual machine is always running even if an availability zone fails. The pitfalls are that you can no longer blindly rely on EBS volumes and by default, IP addresses will change.
- Recovering data in another availability zone is tricky when stored on EBS volumes instead of managed storage services like RDS, EFS, S3, and DynamoDB.
- Some AWS services use multiple availability zones by default, but virtual machines run in a single availability zone.

14

Decoupling your infrastructure: Elastic load balancing and simple queue service

This chapter covers

- The reasons for decoupling a system
- Synchronous decoupling with load balancers to distribute requests
- Hiding your backend from users and message producers
- Asynchronous decoupling with message queues to buffer message peaks

Imagine that you want some advice from us about using AWS, and therefore we plan to meet in a cafe. To make this meeting successful, we must

- Be available at the same time
- Be at the same place
- Find each other at the cafe

The problem with making our meeting happen is that it's *tightly coupled* to a location. We live in Germany; you probably don't. We can solve that issue by decoupling our meeting from the location. So we change plans and schedule a Google Hangout session. Now we must:

- Be available at the same time
- Find each other in Google Hangouts

Google Hangouts (and other video/voice chat services) does *synchronous decoupling*. It removes the need to be at the same place, while still requiring us to meet at the same time.

We can even decouple from time by using email. Now we must:

- Find each other via email

Email does *asynchronous decoupling*. You can send an email when the recipient is asleep, and

they can respond later when they're awake.

SIDE BAR**Examples are 100% covered by the Free Tier**

The examples in this chapter are totally covered by the Free Tier. As long as you don't run the examples longer than a few days, you won't pay anything for it. Keep in mind that this applies only if you created a fresh AWS account for this book and there is nothing else going on in your AWS account. Try to complete the chapter within a few days, because you'll clean up your account at the end of the chapter.

NOTE

To fully understand this chapter, you'll need to have read and understood the concept of auto-scaling covered in chapter 14.

In summary, to meet up, we have have to be at the same place (the cafe), at the same time (3 p.m.), and find each other (I have black hair and I'm wearing a white shirt). Our meeting is tightly coupled to a location and a place. There are two ways to decouple a meeting:

- *Synchronous decoupling*—We can now be at different places. But still, we have to find a common time (3 p.m.), and find each other (exchange Skype IDs, for instance).
- *Asynchronous decoupling*—We can be at different places and now also don't have to find a common time. We only have to find each other (exchange email addresses).

A meeting isn't the only thing that can be decoupled. In software systems, you can find a lot of tightly coupled components:

- A public IP address is like the location of our meeting: To make a request to a web server, you must know its public IP address, and the virtual machine must be connected to that address. If you want to change the public IP address, both parties are involved in making the appropriate changes. The public IP address is tightly coupled with the web server.
- If you want to make a request to a web server, the web server must be online at the same time. Otherwise your request will be rejected. There are many reasons why a web server can be offline: someone might be installing updates, a hardware failure, and so on. The client is tightly coupled with the web server.

AWS offers a solution for *synchronous and asynchronous decoupling*. Typically, *synchronous decoupling* is used when the client expects an immediate response. For example, a user expects an response to the request to load the HTML of a website with very little latency. The *Elastic Load Balancing (ELB)* service provides different types of load balancers that sit between your web servers and the client to decouple your requests synchronously. The client sends a request to the ELB, the ELB forwards the request to a virtual machine or similar target. Therefore, the client does not need to know about the target, it only knows about the load balancer.

Asynchronous decoupling is different and commonly used in scenarios, where the client does not

expect a immediate response. For example, a web application could scale and optimize an image uploaded by the user in the background and use the raw image until that process finished in the background. For asynchronous decoupling, AWS offers the *Simple Queue Service (SQS)* that provides a message queue. The producer sends a message to the queue, and a receiver fetches the message from the queue and processes the request.

You'll learn about both services ELB and SQS in this chapter. Let's start with ELB.

14.1 Synchronous decoupling with load balancers

Exposing a single EC2 instance running a web server to the outside world introduces a dependency: your users now depend on the public IP address of the EC2 instance. As soon as you distribute the public IP address to your users, you can't change it anymore. You're faced with the following issues:

- Changing the public IP address is no longer possible because many clients rely on it.
- If you add an additional EC2 instance (and IP address) to handle the increasing load, it's ignored by all current clients: they're still sending all requests to the public IP address of the first server.

You can solve these issues with a DNS name that points to your server. But DNS isn't fully under your control. DNS resolvers cache responses. DNS servers cache entries, and sometimes they don't respect your time to live (TTL) settings. For example, you might ask DNS servers to only cache the name-to-IP address mapping for one minute, but some DNS servers might use a minimum cache of one day. A better solution is to use a load balancer.

A load balancer can help decouple a system where the requester awaits an immediate response. Instead of exposing your EC2 instances (running web servers) to the outside world, you only expose the load balancer to the outside world. The load balancer then forwards requests to the EC2 instances behind it. Figure [14.1](#) shows how this works.

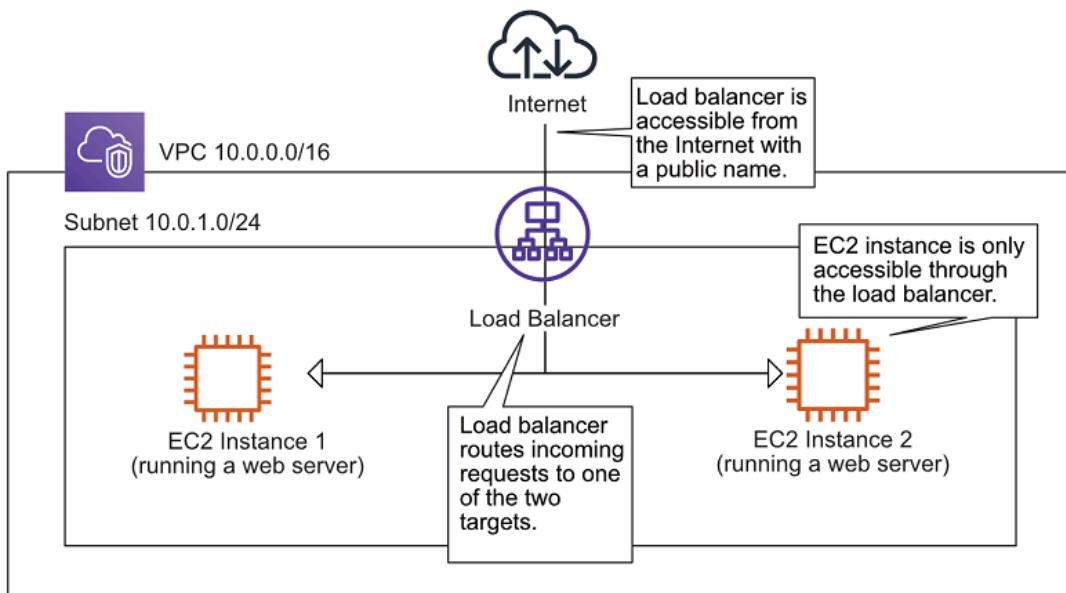


Figure 14.1 A load balancer synchronously decouples your EC2 instances.

The requester (such as a web browser) send an HTTP request to the load balancer. The load balancer then selects one of the EC2 instances and copies the original HTTP request to send to the EC2 instance that it selected. The EC2 instance then processes the request and sends a response. The load balancer receives the response, and sends the same response to the original requester.

AWS offers different types of load balancers through the Elastic Load Balancing (ELB) service. All load balancer types are fault-tolerant and scalable. They differ mainly in the protocols they support:

- *Application Load Balancer (ALB)*—HTTP, HTTPS
- *Network Load Balancer (NLB)*—TCP, TCP+TLS
- *Classic Load Balancer (CLB)*—HTTP, HTTPS, TCP, TCP+TLS

The Classic Load Balancer is the oldest of the load balancers. If you start a new project, we recommend going with the ALB or NLB, because they are in most cases more cost efficient and more feature-rich.

NOTE

The ELB service doesn't have an independent Management Console. It's integrated into the EC2 Management Console.

Load balancers can be used with more than web servers—you can use load balancers in front of any systems that deal with request/response-style communication as long as the protocol is based on TCP.

14.1.1 Setting up a load balancer with virtual machines

AWS shines when it comes to integrating services. In chapter 14, you learned about auto-scaling groups. You'll now put an ALB in front of an auto-scaling group to decouple traffic to web servers, to remove a dependency between your users and the EC2 instance's public IP address. The auto-scaling group will make sure you always have two web servers running. As you learned in chapter 14, that's the way to protect against downtime caused by hardware failure. Servers that are started in the auto-scaling group will automatically register with the ALB.

Figure 14.2 shows what the setup will look like. The interesting part is that the EC2 instances are no longer accessible directly from the public internet, so your users don't know about them. They don't know if there are 2 or 20 EC2 instances running behind the load balancer. Only the load balancer is accessible and forwards requests to the backend servers behind it. The network traffic to load balancers and backend EC2 instances is controlled by security groups, which you learned about in chapter 6.

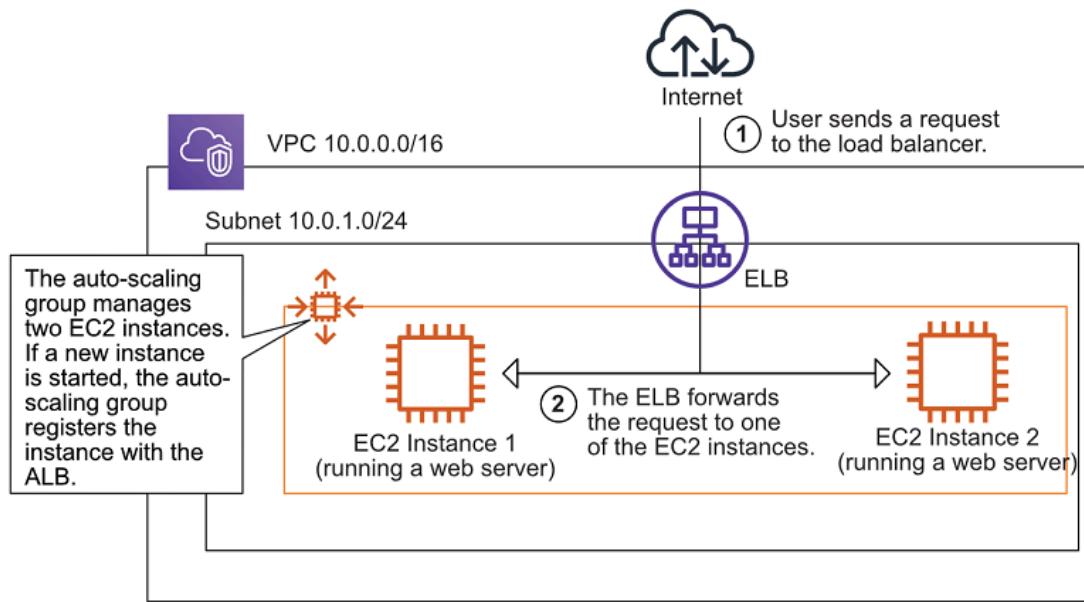


Figure 14.2 The load balancer evaluates rules to forward incoming rules to a specific taget group.

If the auto-scaling group adds or removes EC2 instances, it will also register new EC2 instances with the load balancer and deregister EC2 instances that have been removed.

An ALB consists of three required parts and one optional part:

- *Load balancer*—Defines some core configurations, like the subnets the load balancer runs in, whether the load balancer gets public IP addresses, whether it uses IPv4 or both IPv4 and IPv6, and additional attributes.
- *Listener*—The listener defines the port and protocol that you can use to make requests to

the load balancer. If you like, the listener can also terminate TLS for you. A listener links to a target group that is used as the default if no other listener rules match the request.

- *Target group*—A target group defines your group of backends. The target group is responsible for checking the backends by sending periodic health checks. Usually backends are EC2 instances, but could also be a container running on Elastic Container Service (ECS) as well as Elastic Kubernetes Service (EKS), a Lambda function, or a machine in your data center connected with your VPC.
- *Listener rule*—Optional. You can define a listener rule. The rule can choose a different target group based on the HTTP path or host. Otherwise requests are forwarded to the default target group defined in the listener.

Figure 14.3 shows the ALB parts.

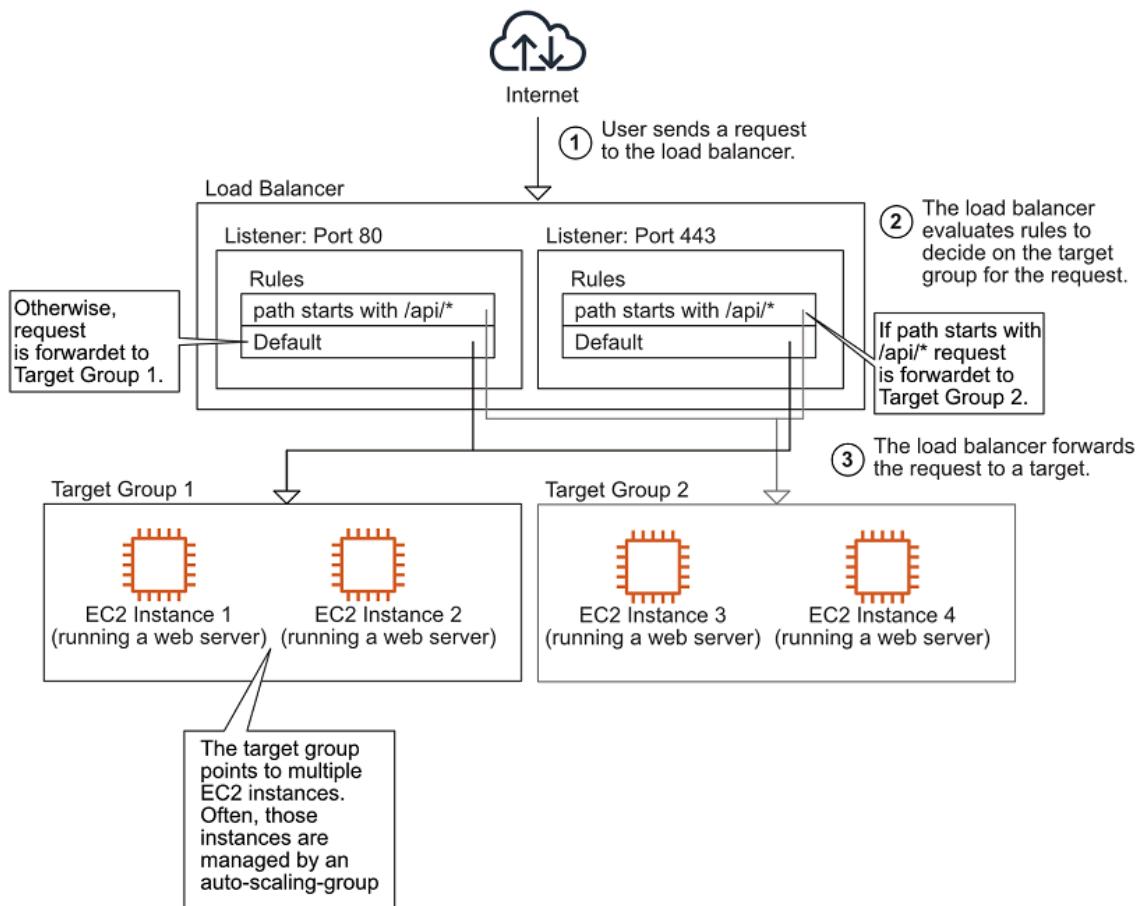


Figure 14.3 Creating an ALB, listener, and target group. Also the auto-scaling group registers instances at the target group automatically.

The following three listing implement the example shown in figure 14.3. The first part -listing 14.1- shows a CloudFormation template snippet to create an ALB and its firewall rules, the security group.

Listing 14.1 Creating a load balancer and connecting it to an auto-scaling group

```
# [...]
LoadBalancerSecurityGroup:
  Type: 'AWS::EC2::SecurityGroup'
  Properties:
    GroupDescription: 'alb-sg'
    VpcId: !Ref VPC
    SecurityGroupIngress:
      - CidrIp: '0.0.0.0/0'
        FromPort: 80 ①
        IpProtocol: tcp
        ToPort: 80
LoadBalancer:
  Type: 'AWS::ElasticLoadBalancingV2::LoadBalancer'
  Properties:
    Scheme: 'internet-facing' ②
    SecurityGroups:
      - !Ref LoadBalancerSecurityGroup ③
    Subnets:
      - !Ref SubnetA
      - !Ref SubnetB
    Type: application
  DependsOn: 'VPCGatewayAttachment'
```

- ① Only traffic on port 80 from the internet will reach the load balancer.
- ② The ALB is publicly accessible (use internal instead of internet-facing to define a load balancer reachable from private network only).
- ③ Assigns the security group to the load balancer
- ④ Attaches the ALB to the subnets

The second part configures the load balancer to listen on port 80 for incoming HTTP requests. Listing [14.2](#) also creates a target group. The default action of the listener forwards all incoming requests to the target group.

Listing 14.2 Creating a load balancer and connecting it to an auto-scaling group

```

Listener:
  Type: 'AWS::ElasticLoadBalancingV2::Listener'
  Properties:
    DefaultActions: ①
      - TargetGroupArn: !Ref TargetGroup
        Type: forward
      LoadBalancerArn: !Ref LoadBalancer
      Port: 80 ②
      Protocol: HTTP
    TargetGroup:
      Type: 'AWS::ElasticLoadBalancingV2::TargetGroup'
      Properties:
        HealthCheckIntervalSeconds: 10 ③
        HealthCheckPath: '/index.html' ④
        HealthCheckProtocol: HTTP
        HealthCheckTimeoutSeconds: 5
        HealthyThresholdCount: 3
        UnhealthyThresholdCount: 2
        Matcher:
          HttpCode: '200-299' ⑤
        Port: 80 ⑥
        Protocol: HTTP
        VpcId: !Ref VPC

```

- ① The load balancer forwards all requests to the default target group
- ② The load balancer listens on port 80 for HTTP requests.
- ③ Every 10 seconds...
- ④ ...HTTP requests are made to /index.html.
- ⑤ If HTTP status code is 2XX, the backend is considered healthy.
- ⑥ The web server on the EC2 instances listens on port 80.

The missing part is shown in listing 14.3: the targets. In our example, we are using an auto-scaling group to launch EC2 instances. The auto-scaling group registers the virtual machine at the target group.

Listing 14.3 Creating a load balancer and connecting it to an auto-scaling group

```

LaunchTemplate:
  Type: 'AWS::EC2::LaunchTemplate'
  # [...]
  Properties:
    LaunchTemplateData:
      IamInstanceProfile:
        Name: !Ref InstanceProfile
      ImageId: !FindInMap [RegionMap, !Ref 'AWS::Region', AMI]
      Monitoring:
        Enabled: false
      InstanceType: 't2.micro'
      NetworkInterfaces:
        - AssociatePublicIpAddress: true
          DeviceIndex: 0
          Groups:
            - !Ref WebServerSecurityGroup
      UserData: # [...]
  AutoScalingGroup:
    Type: 'AWS::AutoScaling::AutoScalingGroup'
    Properties:
      LaunchTemplate:
        LaunchTemplateId: !Ref LaunchTemplate
        Version: !GetAtt 'LaunchTemplate.LatestVersionNumber'
      MinSize: !Ref NumberOfVirtualMachines ①
      MaxSize: !Ref NumberOfVirtualMachines
      DesiredCapacity: !Ref NumberOfVirtualMachines
      TargetGroupARNs: ②
        - !Ref TargetGroup
      VPCZoneIdentifier:
        - !Ref SubnetA
        - !Ref SubnetB
      CreationPolicy:
        ResourceSignal:
          Timeout: 'PT10M'
      DependsOn: 'VPCGatewayAttachment'

```

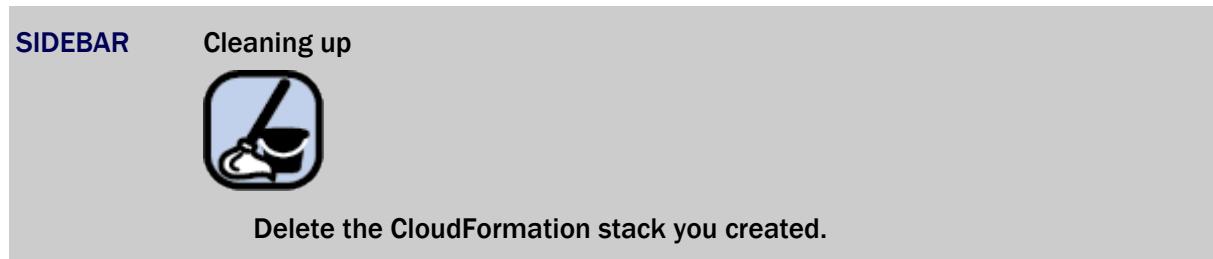
- ① Keeps two EC2 instances running (MinSize DesiredCapacity MaxSize)
- ② The auto-scaling group registers new EC2 instances with the default target group.

The connection between the ALB and the auto-scaling group is made in the auto-scaling group description by specifying `TargetGroupARNs`.

The full CloudFormation template is located at <http://s3.amazonaws.com/awsinaction-code3/chapter14/loadbalancer.yaml>. Create a stack based on that template by clicking on the Quick-Create link at <http://mng.bz/gwER>, and then visit the output of your stack with your browser. Every time you reload the page, you should see one of the private IP addresses of a backend web server.

To get some detail about the load balancer in the graphical user interface, navigate to the EC2 Management Console. The sub-navigation menu on the left has a Load Balancing section where you can find a link to your load balancers. Select the one and only load balancer. You will see

details at the bottom of the page. The details contain a Monitoring tab, where you can find charts about latency, number of requests, and much more. Keep in mind that those charts are one minute behind, so you may have to wait until you see the requests you made to the load balancer.



14.2 Asynchronous decoupling with message queues

Synchronous decoupling with ELB is easy; you don't need to change your code to do it. But for asynchronous decoupling, you have to adapt your code to work with a message queue.

A message queue has a head and a tail. You can add new messages to the tail while reading messages from the head. This allows you to decouple the production and consumption of messages. Now, why would you want to decouple the producers/requesters from consumers/receivers? There are two key benefits:

- *The queue acts as a buffer*—Producers and consumers don't have to run at the same speed. For example, you can add a batch of 1,000 messages in one minute while your consumers always process 10 messages per second. Sooner or later, the consumers will catch up and the queue will be empty again.
- *The queue hides your backend*—Similar to the load balancer, messages producers have no knowledge of the consumers. You can even stop all consumers and still produce messages. This is handy while doing maintenance on your consumers.

When decoupled, the producers and consumers don't know each other; they both only know about the message queue. Figure 14.4 illustrates this principle.

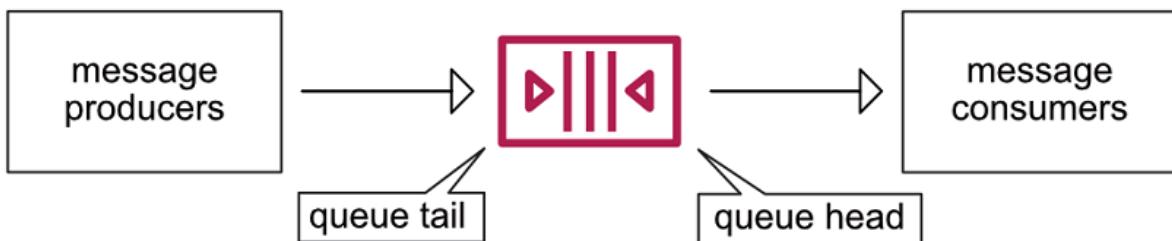


Figure 14.4 Producers send messages to a message queue while consumers read messages

As you decoupled the sender from the receiver, the sender could even put new messages into the queue while no one is consuming messages, and the message queue acts as a buffer. To prevent

message queues from growing infinitely large, messages are only saved for a certain amount of time. If you consume a message from a message queue, you must acknowledge the successful processing of the message to permanently delete it from the queue.

So how to implement asynchronous decoupling on AWS? That's where the *Simple Queue Service (SQS)* comes into play. SQS offers simple but highly scalable -throughput and storage-message queues that guarantee the delivery of messages at least once:

- Under rare circumstances, a single message will be available for consumption twice. This may sound strange if you compare it to other message queues, but you'll see how to deal with this problem later in the chapter.
- SQS doesn't guarantee the order of messages, so you may read messages in a different order than they were produced.

This limitation of SQS is also beneficial:

- You can put as many messages into SQS as you like.
- The message queue scales with the number of messages you produce and consume.
- SQS is highly available by default.
- You pay per message.

The pricing model is simple: \$0.24 to \$0.40 USD per million requests. Also, the first million of requests per month is free of charge. It is important to know, that producing a message counts as a request, and consuming is another request - if your payload is larger than 64 KB, every 64 KB chunk counts as one request.

We observe, that many applications default to a synchronous process. That's probably, because we are used to the request-response model and sometimes forget to think outside the box. However, replacing a synchronous with an asynchronous process enables many advantages in the cloud. Most importantly, scaling becomes much simpler, when there is a queue that can buffer requests for a while. Therefore, you will learn how to transition to an asynchronous process with the help of SQS next.

14.2.1 Turning a synchronous process into an asynchronous one

A typical synchronous process looks like this: a user makes a request to your web server, something happens on the web server, and a result is returned to the user. To make things more concrete, we'll talk about the process of creating a preview image of an URL in the following example illustrated in figure [15.5](#):

1. The user submits a URL.
2. The web server downloads the content at the URL, takes a screenshot, and renders it as a PNG image.
3. The web server returns the PNG to the user.

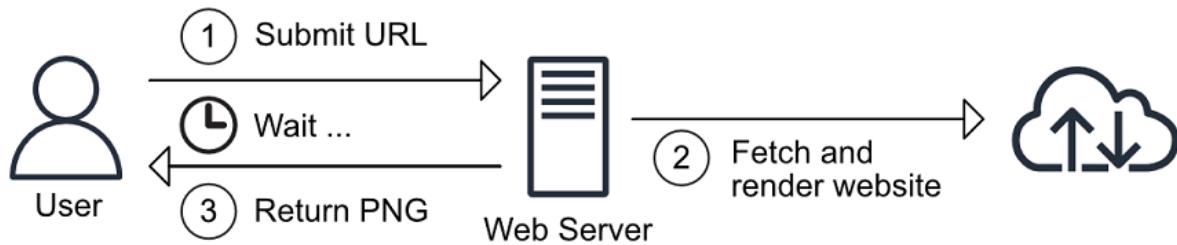


Figure 14.5 A synchronous process to create a screenshot of a website.

With one small trick, this process can be made asynchronous, and benefit from the elasticity of a message queue, for example during peak traffic as shown in figure 15.6:

1. The user submits a URL.
2. The web server puts a message into a queue that contains a random ID and the URL.
3. The web server returns a link to the user where the PNG image will be found in the future. The link contains the random ID (such as [http://\\$Bucket.s3.amazonaws.com/\\$RandomId.png](http://$Bucket.s3.amazonaws.com/$RandomId.png)).
4. In the background, a worker consumes the message from the queue.
5. The worker downloads the content, converts the content into a PNG.
6. Next, the worker uploads the image to S3.
7. At some point, the user tries to download the PNG at the known location. If the file is not found, the user should reload the page in a few seconds.

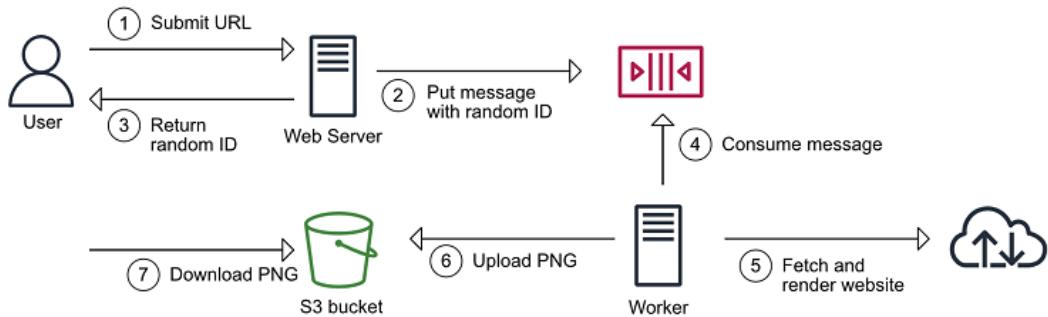


Figure 14.6 The same process, but asynchronous.

If you want to make a process asynchronous, you must manage the way the process initiator tracks the process status. One way of doing that is to return an ID to the initiator that can be used to look up the process. During the process, this ID is passed from step to step.

14.2.2 Architecture of the URL2PNG application

You'll now create a simple but decoupled piece of software named URL2PNG that renders a PNG from a given web URL. You'll use Node.js to do the programming part, and you'll use SQS as the message queue implementation. Figure 14.7 shows how the URL2PNG application works.

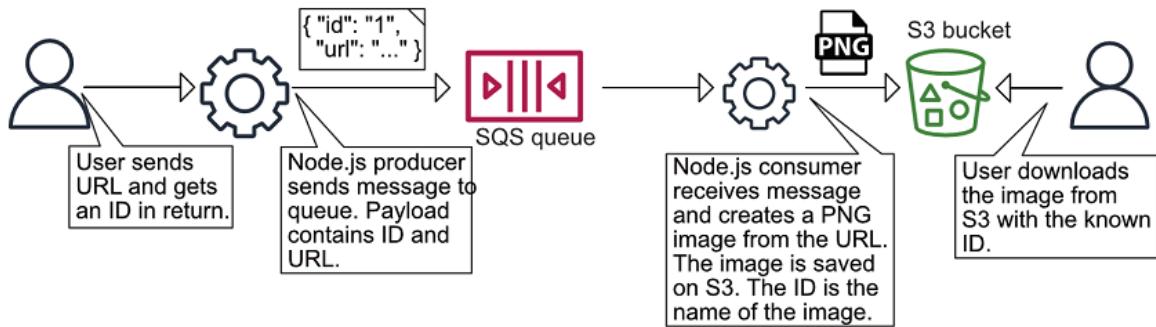


Figure 14.7 Node.js producer sends a message to the queue. The payload contains an ID and URL

On the message producer side, a small Node.js script generates a unique ID, sends a message to the queue with the URL and ID as the payload, and returns the ID to the user. The user now starts checking if a file is available on the S3 bucket using the returned ID as the filename.

Simultaneously, on the message consumer side, a small Node.js script reads a message from the queue, generates the screenshot of the URL from the payload, and uploads the resulting image to an S3 bucket using the unique ID from the payload as the filename.

To complete the example, you need to create an S3 bucket with web hosting enabled. Execute the following commands, replacing `$yourusername` with your name or nickname to prevent name clashes with other readers (remember that S3 bucket names have to be globally unique across all AWS accounts):

```
$ aws s3 mb s3://url2png-$yourusername
```

Now it's time to create the message queue.

14.2.3 Setting up a message queue

Creating an SQS queue is simple: you only need to specify the name of the queue:

```
$ aws sqs create-queue --queue-name url2png
{
  "QueueUrl": "https://queue.amazonaws.com/878533158213/url2png"
}
```

The returned `QueueUrl` is needed later in the example, so take a note.

14.2.4 Producing messages programmatically

You now have an SQS queue to send messages to. To produce a message, you need to specify the queue and a payload. You'll use Node.js in combination with the AWS SDK to make requests to AWS.

SIDE BAR**Installing and getting started with Node.js**

Node.js is a platform for executing JavaScript in an event-driven environment so you can easily build network applications. To install Node.js, visit <https://nodejs.org> and download the package that fits your OS. All examples in this book are tested with Node.js 14.

After Node.js is installed, you can verify if everything works by typing `node --version` into your terminal. Your terminal should respond with something similar to `v14.*`. Now you're ready to run JavaScript examples like URL2PNG.

Do you want to get started with Node.js? We recommend Node.js in Action (2nd edition) from Alex Young, et al., (Manning, 2017) or the video course Node.js in Motion from P.J. Evans, (Manning, 2018).

Here's how the message is produced with the help of the AWS SDK for Node.js; it will later be consumed by the URL2PNG worker. The Node.js script can then be used like this (don't try to run this command now—you need to install and configure URL2PNG first):

```
$ node index.js "http://aws.amazon.com"
PNG will be available soon at
http://url2png-$yourname.s3.amazonaws.com/XYZ.png
```

As usual, you'll find the code in the book's code repository on GitHub <https://github.com/AWSinAction/code3>. The URL2PNG example is located at / chapter15/url2png/. Listing 14.4 shows implementation of index.js.

Listing 14.4 index.js: sending a message to the queue

```
const AWS = require('aws-sdk');
var { v4: uuidv4 } = require('uuid');
const config = require('./config.json');
const sqs = new AWS.SQS({}); ①

if (process.argv.length !== 3) { ②
  console.log('URL missing');
  process.exit(1);
}

const id = uuidv4(); ③
const body = { ④
  id,
  url: process.argv[2]
};

sqs.sendMessage({ ⑤
  MessageBody: JSON.stringify(body), ⑥
  QueueUrl: config.QueueUrl ⑦
}, (err) => {
  if (err) {
    console.log('error', err);
  } else {
    console.log('PNG will be soon available at http://' + config.Bucket
      + '.s3.amazonaws.com/' + id + '.png');
  }
});
```

- ① Creates an SQS client
- ② Checks whether a URL was provided
- ③ Creates a random ID
- ④ The payload contains the random ID and the URL.
- ⑤ Invokes the sendMessage operation on SQS
- ⑥ Converts the payload into a JSON string
- ⑦ Queue to which the message is sent (was returned when creating the queue).

Before you can run the script, you need to install the Node.js modules. Run `npm install` in your terminal to install the dependencies. You'll find a `config.json` file that needs to be modified. Make sure to change `QueueUrl` to the queue you created at the beginning of this example, and change `Bucket` to `url2png-$yourname`.

Now you can run the script with `node index.js "http://aws.amazon.com"`. The program should respond with something like “PNG will be available soon at [http://url2png-\\$yourname.s3.amazonaws.com/XYZ.png](http://url2png-$yourname.s3.amazonaws.com/XYZ.png)”. To verify that the message is ready for consumption, you can ask the queue how many messages are inside. Replace `$QueueUrl` with your queue's URL.

```
$ aws sqs get-queue-attributes \
  --queue-url "$QueueUrl" \
  --attribute-names ApproximateNumberOfMessages
{
  "Attributes": {
    "ApproximateNumberOfMessages": "1"
  }
}
```

SQS only returns an approximation of the number of messages. This is due to the distributed nature of SQS. If you don't see your message in the approximation, run the command again and eventually you will see your message.

Next, it's time to create the worker that consumes the message and does all the work of generating a PNG.

14.2.5 Consuming messages programmatically

Processing a message with SQS takes three steps:

1. Receive a message.
2. Process the message.
3. Acknowledge that the message was successfully processed.

You'll now implement each of these steps to change a URL into a PNG.

To receive a message from an SQS queue, you must specify the following:

- QueueUrl—The unique queue identifier.
- MaxNumberOfMessages—The maximum number of messages you want to receive (from 1 to 10). To get higher throughput, you can get messages in a batch. We usually set this to 10 for best performance and lowest overhead.
- VisibilityTimeout—The number of seconds you want to remove this message from the queue to process it. Within that time, you must delete the message, or it will be delivered back to the queue. We usually set this to the average processing time multiplied by four.
- WaitTimeSeconds—The maximum number of seconds you want to wait to receive messages if they're not immediately available. Receiving messages from SQS is done by polling the queue. But AWS allows long polling, for a maximum of 20 seconds. When using long polling, you will not get an immediate response from the AWS API if no messages are available. If a new message arrives within 10 seconds, the HTTP response will be sent to you. After 20 seconds, you also get an empty response.

Listing 14.5 shows how this is done with the SDK.

Listing 14.5 worker.js: receiving a message from the queue

```
const fs = require('fs');
const AWS = require('aws-sdk');
const puppeteer = require('puppeteer');
const config = require('./config.json');
const sqs = new AWS.SQS();
const s3 = new AWS.S3();

async function receive() {
  const result = await sqs.receiveMessage({ ①
    QueueUrl: config.QueueUrl,
    MaxNumberOfMessages: 1, ②
    VisibilityTimeout: 120, ③
    WaitTimeSeconds: 10 ④
  }).promise();

  if (result.Messages) { ⑤
    return result.Messages[0] ⑥
  } else {
    return null;
  }
};
```

- ① Invokes the receiveMessage operation on SQS
- ② Consumes no more than one message at once
- ③ Takes the message from the queue for 120 seconds
- ④ Long poll for 10 seconds to wait for new messages
- ⑤ Checks whether a message is available
- ⑥ Gets the one and only message

The receive step has now been implemented. The next step is to process the message. Thanks to the Node.js module `puppeteer`, it's easy to create a screenshot of a website as demonstrated by

listing 14.6.

Listing 14.6 worker.js: processing a message (take screenshot and upload to S3)

```
async function process(message) {
  const body = JSON.parse(message.Body); ①
  const browser = await puppeteer.launch(); ②
  const page = await browser.newPage();

  await page.goto(body.url);
  page.setViewport({ width: 1024, height: 768}); ③
  const screenshot = await page.screenshot(); ④

  await s3.upload({ <4> Uploads screenshot to S3
    Bucket: config.Bucket, ⑤
    Key: `${body.id}.png`, ⑥
    Body: screenshot,
    ContentType: 'image/png', ⑦
    ACL: 'public-read',
  }).promise();

  await browser.close();
};
```

- ① The message body is a JSON string. You convert it back into a JavaScript object.
- ② Launches a headless browser
- ③ Takes a screenshot
- ④ The S3 bucket to upload image to
- ⑤ The key consisting of the random ID generated by the client and included in the SQS message
- ⑥ Set the content type to make sure browsers are showing the image correctly
- ⑦ Allows anyone to read the image from S3 (public access)

The only step that's missing is to acknowledge that the message was successfully consumed. This is done by deleting the message from the queue after successfully completing the task. If you receive a message from SQS, you get a `ReceiptHandle`, which is a unique ID that you need to specify when you delete a message from a queue.

Listing 147 worker.js: acknowledging a message (deletes the message from the queue)

```
async function acknowledge(message) {
  await sqs.deleteMessage({ ①
    QueueUrl: config.QueueUrl,
    ReceiptHandle: message.ReceiptHandle ②
  }).promise();
};
```

- ① Invokes the `deleteMessage` operation on SQS
- ② `ReceiptHandle` is unique for each receipt of a message

You have all the parts; now it's time to connect them.

Listing 14.8 worker.js: connecting the parts

```

async function run() {
  while(true) {    ①
    const message = await receive();  ②
    if (message) {
      console.log('Processing message', message);
      await process(message);  ③
      await acknowledge(message);  ④
    }
    await new Promise(r => setTimeout(r, 1000));  ⑤
  }
}

run();  ⑥

```

- ① An endless loop polling and processing messages
- ② Receive a message
- ③ Process the message
- ④ Acknowledge the message by deleting it from the queue
- ⑤ Sleep for 1 second to decrease number of requests to SQS
- ⑥ Start the loop

Now you can start the worker to process the message that is already in the queue. Run the script with `node worker.js`. You should see some output that says the worker is in the process step and then switches to Done. After a few seconds, the screenshot should be uploaded to S3. Your first asynchronous application is complete.

Remember the output you got when you invoked `node index.js "http://aws.amazon.com"` to send a message to the queue? It looked similar to this: `http://url2png-$yourname.s3.amazonaws.com/XYZ.png`. Now put that URL in your web browser and you will find a screenshot of the AWS website (or whatever you used as an example).

You've created an application that is asynchronously decoupled. If the URL2PNG service becomes popular and millions of users start using it, the queue will become longer and longer because your worker can't produce that many PNGs from URLs. The cool thing is that you can add as many workers as you like to consume those messages. Instead of only one worker, you can start 10 or 100. The other advantage is that if a worker dies for some reason, the message that was in flight will become available for consumption after two minutes and will be picked up by another worker. That's fault-tolerant! If you design your system to be asynchronously decoupled, it's easy to scale and a good foundation to be fault-tolerant. The next chapter will concentrate on this topic.

SIDE BAR**Cleaning up**

Delete the message queue as follows:

```
$ aws sqs delete-queue --queue-url "$QueueUrl"
```

And don't forget to clean up and delete the S3 bucket used in the example. Issue the following command, replacing \$yourname with your name:

```
$ aws s3 rb --force s3://url2png-$yourname
```

14.2.6 Limitations of messaging with SQS

Earlier in the chapter, we mentioned a few limitations of SQS. This section covers them in more detail. But before we start with the limitations, here are the benefits:

- You can put as many messages into SQS as you like. SQS scales the underlying infrastructure for you.
- SQS is highly available by default.
- You pay per message.

Those benefits come with some trade-offs. Let's have a look at those limitations in more detail now.

SQS DOESN'T GUARANTEE THAT A MESSAGE IS DELIVERED ONLY ONCE

There are two reasons why a message might be delivered more than once:

1. Common reason: If a received message isn't deleted within `VisibilityTimeout`, the message will be received again.
2. Rare reason: If a `DeleteMessage` operation doesn't delete all copies of a message because one of the servers in the SQS system isn't available at the time of deletion.

The problem of repeated delivery of a message can be solved by making the message processing idempotent. *Idempotent* means that no matter how often the message is processed, the result stays the same. In the URL2PNG example, this is true by design: If you process the message multiple times, the same image will be uploaded to S3 multiple times. If the image is already available on S3, it's replaced. Idempotence solves many problems in distributed systems that guarantee messages will be delivered at least once.

Not everything can be made idempotent. Sending an email is a good example: if you process a message multiple times and it sends an email each time, you'll annoy the addressee.

In many cases, processing at least once is a good trade-off. Check your requirements before using SQS if this trade-off fits your needs.

SQS DOESN'T GUARANTEE THE MESSAGE ORDER

Messages may be consumed in a different order than the order in which you produced them. If you need a strict order, you should search for something else. SQS is a fault-tolerant and scalable message queue. If you need a stable message order, you'll have difficulty finding a solution that scales like SQS. Our advice is to change the design of your system so you no longer need the stable order, or put the messages in order on the client side.

SIDE BAR **SQS FIFO (first-in-first-out) queues**

FIFO queues guarantee order of messages and have a mechanism to detect duplicate messages. If you need a strict message order, they are worth a look. The disadvantages are higher pricing and a limitation on 3000 operations per second. Check out the documentation at <http://mng.bz/e7WJ> for more information

SQS DOESN'T REPLACE A MESSAGE BROKER

SQS isn't a message broker like ActiveMQ—SQS is only a message queue. Don't expect features like message routing or message priorities. Comparing SQS to ActiveMQ is like comparing DynamoDB to MySQL.

SIDE BAR **Amazon MQ**

AWS announced an alternative to Amazon SQS in November 2017: Amazon MQ provides Apache ActiveMQ as a service. Therefore, you can use Amazon MQ as a message broker that speaks the JMS, NMS, AMQP, STOMP, MQTT, and WebSocket protocols.

Go to the Amazon MQ Developer Guide at <https://docs.aws.amazon.com/amazon-mq/latest/developer-guide/> to learn more.

14.3 Summary

- Decoupling makes things easier because it reduces dependencies.
- Synchronous decoupling requires two sides to be available at the same time, but the sides don't have to know each other.
- With asynchronous decoupling, you can communicate without both sides being available.
- Most applications can be synchronously decoupled without touching the code, by using a load balancer offered by the ELB service.
- A load balancer can make periodic health checks to your application to determine whether the backend is ready to serve traffic.
- Asynchronous decoupling is only possible with asynchronous processes. But you can modify a synchronous process to be an asynchronous one most of the time.
- Asynchronous decoupling with SQS requires programming against SQS with one of the SDKs.

16

Designing for fault tolerance

This chapter covers

- What is fault-tolerance and why do you need it?
- Using redundancy to remove single points of failure
- Improve fault-tolerance by retrying on failure
- Using idempotent operations to achieve retry on failure
- AWS service guarantees

Failure is inevitable: hard disks, networks, power, and so on all fail from time to time. But failures do not have to impact the users of your system.

A fault-tolerant system provides the highest quality to your users. No matter what happens in your system, the user is never affected and can continue to go about her work, consume entertaining content, buy goods and services, or have conversations with friends. A few years ago, achieving fault tolerance was expensive and complicated, but with AWS, providing fault-tolerant systems is becoming an affordable standard. Nevertheless, building fault-tolerant systems is the top tier of cloud computing, and might be challenging at the beginning.

Designing for fault tolerance means to plan for failure and built systems capable to resolve failure conditions automatically. An important aspect when is to avoid single points of failures. You can achieve fault-tolerance by introducing redundancy into your system. Instead of running your application on a single EC2 instance, you distribute the application among multiple machines. Also decoupling the parts of your architecture such that one side does not rely on the uptime of the other is important. For example, the web server could deliver cached content, in case the database is not reachable.

The services provided by AWS offer different types of *failure resilience*. Resilience is the ability

to deal with a failure with no or little impact on the user. You will learn about the resilience guarantees of major services in the following. But in general, if you are unsure about the resilience capabilities of an AWS service, refer to the *Resilience* section of the official documentation for that service. A fault-tolerant system is very resilient to failure. We group AWS services into three different categories.

- *No guarantees (single point of failure)*—No requests are served in case of failure.
- *High availability*—In case of failure, it takes some time until requests are served as before.
- *Fault-tolerance*—In case of failure, requests are served as before without any availability issues.

The most convenient way to make your system fault-tolerant is to build the architecture using fault-tolerant services only. You will learn about fault-tolerant services in the following. If all building blocks are fault-tolerant, the whole system will be fault-tolerant as well. Luckily, many AWS services are fault-tolerant by default. If possible, use them. Otherwise you'll need to deal with the consequences and handle failures yourself.

Unfortunately, one important service isn't fault-tolerant by default: EC2 instances. Virtual machines aren't fault-tolerant. This means an architecture that uses EC2 isn't fault-tolerant by default. But AWS provides the building blocks to help you improve the fault-tolerance of virtual machines. In this chapter, we will show you how to use auto-scaling groups, Elastic Load Balancing (ELB), and Simple Queue Service (SQS) to turn EC2 instances into fault-tolerant system.

First, however, let's look at the level of failure resistance of key services. Knowing which services are fault tolerant, which are highly available, and which are neither will help you create the kind of fault tolerance your system needs.

The following services provided by AWS are neither highly available nor fault-tolerant. When using one of these services in your architecture, you are adding a *single point of failure (SPOF)* to your infrastructure. In this case, to achieve fault-tolerance, you need to plan and build for failure as discussed during the rest of the chapter.

- *Amazon Elastic Compute Cloud (EC2) instance*—A single EC2 instance can fail for many reasons: hardware failure, network problems, availability-zone outage, and so on. To achieve high availability or fault-tolerance, use auto-scaling groups to set up a fleet of EC2 instances that serve requests in a redundant way.
- *Amazon Relational Database Service (RDS) single instance*—A single RDS instance could fail for the same reasons than an EC2 instance might fail. Use Multi-AZ mode to achieve high availability.

All the following services are *highly available (HA)* by default. When a failure occurs, the services will suffer from a short downtime but will recover automatically:

- *Elastic Network Interface (ENI)*—A network interface is bound to an AZ, so if this AZ goes down, your network interface will be unavailable as well.
- *Amazon Virtual Private Cloud (VPC) subnet*—A VPC subnet is bound to an AZ, so if this AZ suffers from an outage, your subnet will not be reachable as well. Use multiple subnets in different AZs to remove the dependency on a single AZ.
- *Amazon Elastic Block Store (EBS) volume*—An EBS volume distributes data among multiple machines within an AZ. But if the whole AZ fails, your volume will be unavailable (you won't lose your data though). You can create EBS snapshots from time to time so you can re-create an EBS volume in another AZ.
- *Amazon Relational Database Service (RDS) Multi-AZ instance*—When running in Multi-AZ mode, a short downtime (1 minute) is expected if an issue occurs with the master instance while changing DNS records to switch to the standby instance.

The following services are *fault-tolerant* by default. As a consumer of the service, you won't notice any failures.

- Elastic Load Balancing (ELB), deployed to at least two AZs
- Amazon EC2 security groups
- Amazon Virtual Private Cloud (VPC) with an ACL and a route table
- Elastic IP addresses (EIP)
- Amazon Simple Storage Service (S3)
- Amazon Elastic Block Store (EBS) snapshots
- Amazon DynamoDB
- Amazon CloudWatch
- Auto-scaling groups
- Amazon Simple Queue Service (SQS)
- AWS CloudFormation
- AWS Identity and Access Management (IAM, not bound to a single region; if you create an IAM user, that user is available in all regions)

SIDE BAR

Chapter requirements

To fully understand this chapter, you need to have read and understood the following concepts:

- **EC2 (chapter 3)**
- **Auto-scaling (chapter 14)**
- **Elastic Load Balancing (chapter 15)**
- **Simple Queue Service (chapter 15)**

On top of that, the example included in this chapter makes intensive use of the following:

- **DynamoDB (chapter 13)**
- **Express, a Node.js web application framework**

In this chapter, you'll learn everything you need to design a fault-tolerant web application based on EC2 instances (which aren't fault-tolerant by default).

During this chapter, you will build a fault-tolerant web application which allows a user to upload an image, apply a sepia filter on the image, and download the image. First, you will learn how to distribute a workload among multiple EC2 instances. Instead of running a single virtual machine, you will spin up multiple machines in different data centers - also known as availability zones. Next, you will learn how to increase the resilience of your code. Afterwards, you will create an infrastructure consisting of a queue (SQS), a load balancer (ALB), EC2 instances managed by auto-scaling groups, and a database (DynamoDB).

16.1 Using redundant EC2 instances to increase availability

Here are just a few reasons why your virtual machine might fail:

- If the host hardware fails, it can no longer host the virtual machine on top of it.
- If the network connection to/from the host is interrupted, the virtual machine will lose the ability to communicate over network.
- If the host system is disconnected from the power supply, the virtual machine will fail as well.

Additionally the software running inside your virtual machine may also cause a crash:

- If your application contains a memory leak, you'll run out of memory and fail. It may take a day, a month, a year, or more, but eventually it will happen.
- If your application writes to disk and never deletes its data, you'll run out of disk space sooner or later, causing your application to fail.
- Your application may not handle edge cases properly and may instead crash unexpectedly.

Regardless of whether the host system or your application is the cause of a failure, a single EC2 instance is a single point of failure. If you rely on a single EC2 instance, your system will blow up eventually. It's merely a matter of time.

16.1.1 Redundancy can remove a single point of failure

Imagine a production line that makes fluffy cloud pies. Producing a fluffy cloud pie requires several production steps (simplified!):

1. Produce a pie crust.
2. Cool the pie crust.
3. Put the fluffy cloud mass on top of the pie crust.
4. Cool the fluffy cloud pie.
5. Package the fluffy cloud pie.

The current setup is a single production line. The big problem with this setup is that whenever one of the steps crashes, the entire production line must be stopped. Figure 16.1 illustrates the problem when the second step (cooling the pie crust) crashes. The steps that follow no longer work either, because they no longer receive cool pie crusts.

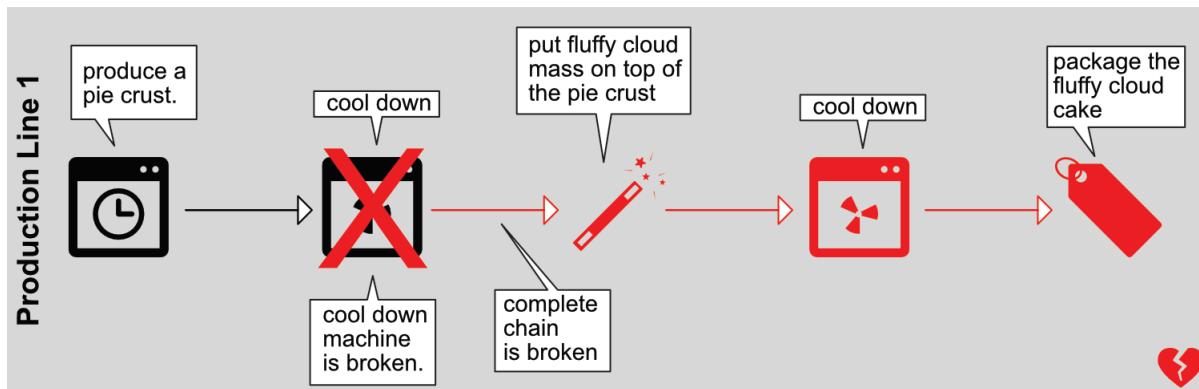


Figure 16.1 A single point of failure affects not only itself, but the entire system.

Why not have multiple production lines, each producing pies from pie crust through packaging? Instead of one line, suppose we have three. If one of the lines fails, the other two can still produce fluffy cloud pies for all the hungry customers in the world. Figure 16.2 shows the improvements; the only downside is that we need three times as many machines.

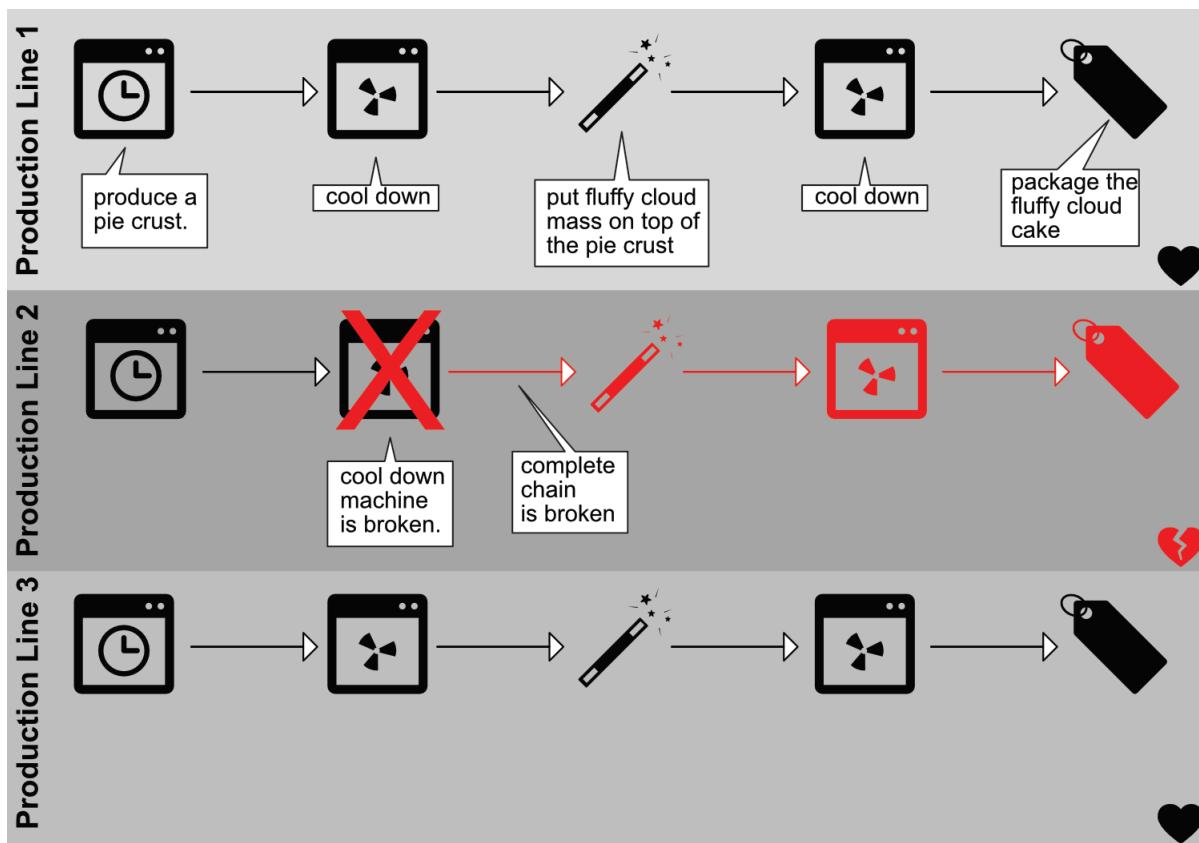


Figure 16.2 Redundancy eliminates single points of failure and makes the system more stable.

The example can be transferred to EC2 instances as well. Instead of having only one EC2 instance running your application, you can have three. If one of those instances fails, the other two will still be able to serve incoming requests. You can also minimize the cost impact of one versus three instances: instead of one large EC2 instance, you can choose three small ones. The problem that arises when using multiple virtual machines: how can the client communicate with the instances? The answer is *decoupling*: put a load balancer or message queue between your EC2 instances and the client. Read on to learn how this works.

16.1.2 Redundancy requires decoupling

In chapter 15, you learned about how to use Elastic Load Balancing (ELB) and the Simple Queue Service (SQS) to decouple different parts of a system. You will apply both approaches to built a fault tolerant system next.

First, figure 16.3 shows how EC2 instances can be made fault-tolerant by using redundancy and synchronous decoupling. If one of the EC2 instances crashes, the load balancer stops routing requests to the crashed instances. The auto-scaling group replaces the crashed EC2 instance within minutes, and the load balancer begins to route requests to the new instance.

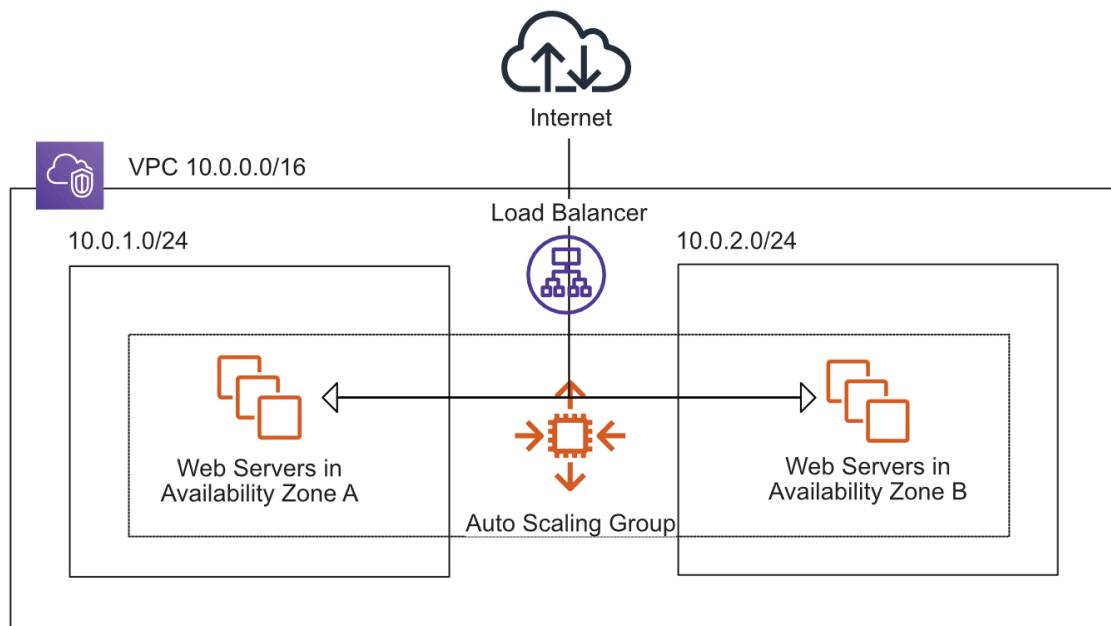


Figure 16.3 Fault-tolerant EC2 instances with an auto-scaling group and an Elastic Load Balancer

Take a second look at figure 16.3 and see what parts are redundant:

- *Availability zones (AZs)*—Two are used. If one AZ suffers from an outage, we still have instances running in the other AZ.
- *Subnets*—A subnet is tightly coupled to an AZ. Therefore we need one subnet in each AZ.
- *EC2 instances*—Two subnets with one ore more EC2 instanecs leads to redudancy

among availability zones.

- *Load Balancer*—The load balancer spans multiple subnets and therefore multiple availability zones.

Next, figure 16.4 shows a fault-tolerant system built with EC2 that uses the power of redundancy and asynchronous decoupling to process messages from an SQS queue.

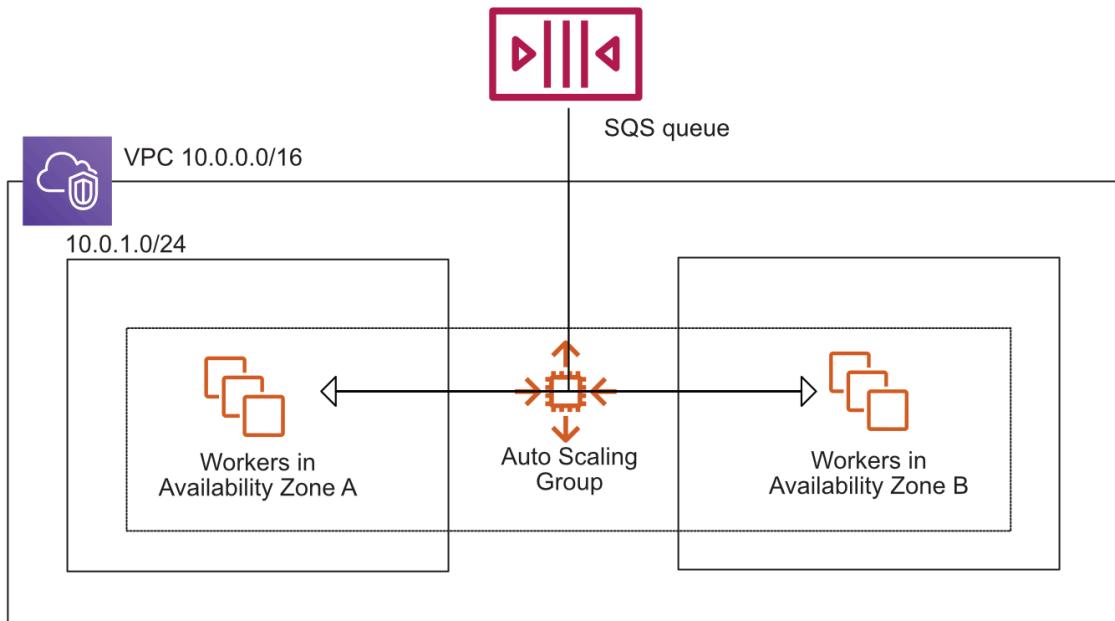


Figure 16.4 Fault-tolerant EC2 instances with an auto-scaling group and SQS

Second, figure 16.3 and 16.4 the load balancer and the SQS queue appear only once. This doesn't mean that ELB or SQS are single points of failure; on the contrary, ELB and SQS are both fault-tolerant by default.

You will learn how to use both models, synchronous decoupling with a load balancer and asynchronous decoupling with a queue, to build a fault-tolerant system in the following. But before we do so, let's have a look into important considerations for making your code more resilient.

16.2 Considerations for making your code fault-tolerant

If you want to achieve fault tolerance, you have to build your application accordingly. You can design fault tolerance into your application by following two suggestions:

1. In case of failure, let it crash, but also retry.
2. Try to write idempotent code wherever possible.

16.2.1 Let it crash, but also retry

The Erlang programming language is famous for the concept of “let it crash.” That means whenever the program doesn’t know what to do, it crashes, and someone needs to deal with the crash. Most often people overlook the fact that Erlang is also famous for retrying. Letting it crash without retrying isn’t useful—if you can’t recover from a crash, your system will be down, which is the opposite of what you want.

You can apply the “let it crash” concept (some people call it “fail fast”) to synchronous and asynchronous decoupled scenarios. In a synchronous decoupled scenario, the sender of a request must implement the retry logic. If no response is returned within a certain amount of time, or an error is returned, the sender retries by sending the same request again. In an asynchronous decoupled scenario, things are easier. If a message is consumed but not acknowledged within a certain amount of time, it goes back to the queue. The next consumer then grabs the message and processes it again. Retrying is built into asynchronous systems by default.

“Let it crash” isn’t useful in all situations. If the program wants to respond to the sender that the request contained invalid content, this isn’t a reason for letting the server crash: the result will stay the same no matter how often you retry. But if the server can’t reach the database, it makes a lot of sense to retry. Within a few seconds, the database may be available again and able to successfully process the retried request.

Retrying isn’t that easy. Imagine that you want to retry the creation of a blog post. With every retry, a new entry in the database is created, containing the same data as before. You end up with many duplicates in the database. Preventing this involves a powerful concept that’s introduced next: idempotent retry.

16.2.2 Idempotent retry makes fault tolerance possible

How can you prevent a blog post from being added to the database multiple times because of a retry? A naïve approach would be to use the title as primary key. If the primary key is already used, you can assume that the post is already in the database and skip the step of inserting it into the database. Now the insertion of blog posts is *idempotent*, which means no matter how often a certain action is applied, the outcome must be the same. In the current example, the outcome is a database entry.

It continues with a more complicated example. Inserting a blog post is more complicated in reality, as the process might look something like this:

1. Create a blog post entry in the database.
2. Invalidate the cache because data has changed.
3. Post the link to the blog’s Twitter feed.

Let's take a close look at each step.

1. CREATING A BLOG POST ENTRY IN THE DATABASE

We covered this step earlier by using the title as a primary key. But this time, we use a universally unique identifier (UUID) instead of the title as the primary key. A UUID like 550e8400-e29b-11d4-a716-446655440000 is a random ID that's generated by the client. Because of the nature of a UUID, it's unlikely that two identical UUIDs will be generated. If the client wants to create a blog post, it must send a request to the load balancer containing the UUID, title, and text. The load balancer routes the request to one of the backend servers. The backend server checks whether the primary key already exists. If not, a new record is added to the database. If it exists, the insertion continues. Figure 16.5 shows the flow.

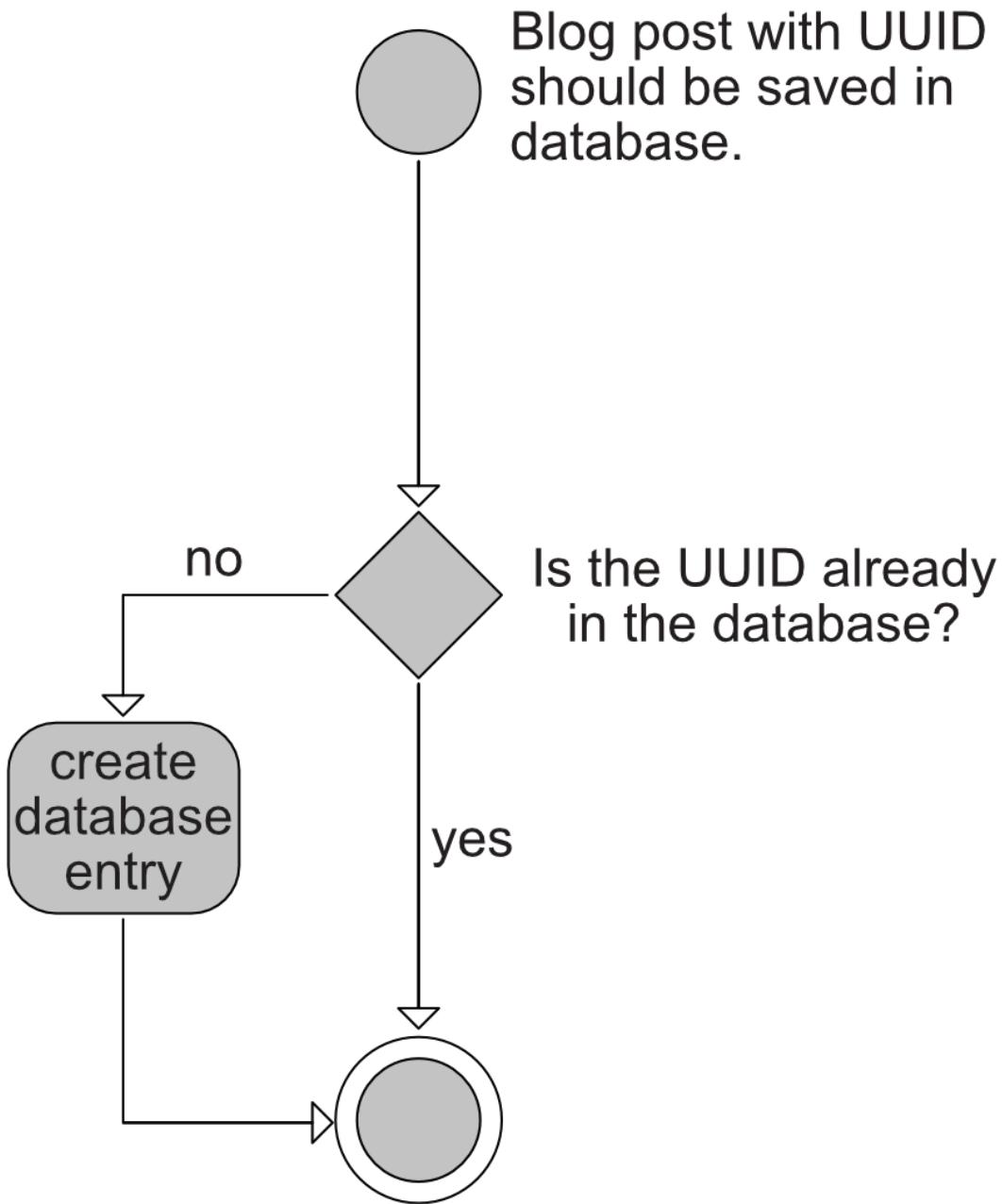


Figure 16.5 Idempotent database insert: creating a blog post entry in the database only if it doesn't already exist

Creating a blog post is a good example of an idempotent operation that is guaranteed by code. You can also use your database to handle this problem. Just send an insert to your database. Three things could happen:

1. Your database inserts the data. The operation is successfully completed.
2. Your database responds with an error because the primary key is already in use. The operation is successfully completed.

3. Your database responds with a different error. The operation crashes.

Think twice about the best way to implement idempotence!

2. INVALIDATING THE CACHE

This step sends an invalidation message to a caching layer. You don't need to worry about idempotence too much here: it doesn't hurt if the cache is invalidated more often than needed. If the cache is invalidated, then the next time a request hits the cache, the cache won't contain data, and the original source (in this case, the database) will be queried for the result. The result is then put in the cache for subsequent requests. If you invalidate the cache multiple times because of a retry, the worst thing that can happen is that you may need to make a few more calls to your database. That's easy.

3. POSTING TO THE BLOG'S TWITTER FEED

To make this step idempotent, you need to use some tricks, because you interact with a third party that doesn't support idempotent operations. Unfortunately, no solution will guarantee that you post exactly one status update to Twitter. You can guarantee the creation of at least one (one or more than one) status update, or at most one (one or none) status update. An easy approach could be to ask the Twitter API for the latest status updates; if one of them matches the status update that you want to post, you skip the step because it's already done.

But Twitter is an eventually consistent system: there is no guarantee that you'll see a status update immediately after you post it. Therefore, you can end up having your status update posted multiple times. Another approach would be to save in a database whether you already posted the status update. But imagine saving to the database that you posted to Twitter and then making the request to the Twitter API—but at that moment, the system crashes. Your database will state that the Twitter status update was posted, but in reality it wasn't. You need to make a choice: tolerate a missing status update, or tolerate multiple status updates. Hint: it's a business decision. Figure [16.6](#) shows the flow of both solutions.

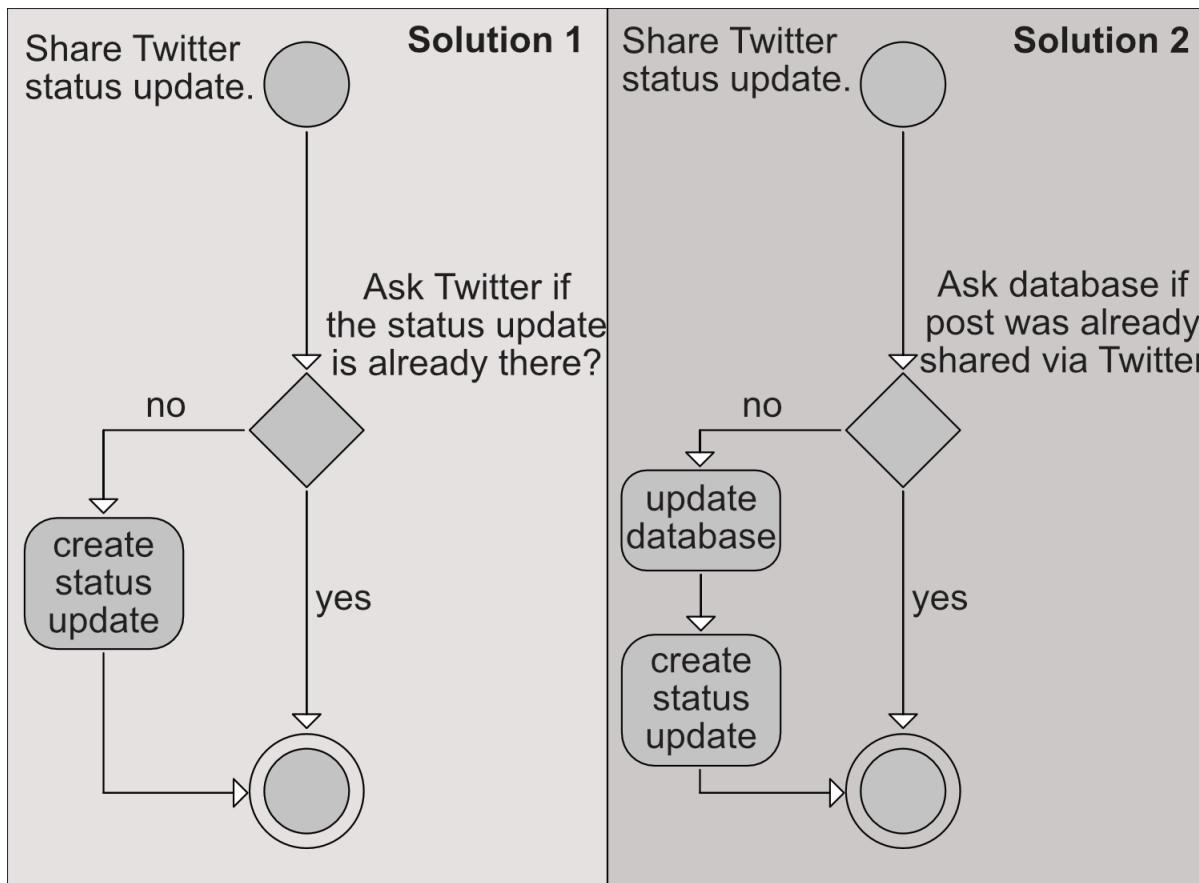


Figure 16.6 Idempotent Twitter status update: only share a status update if it hasn't already been done

Now it's time for a practical example! You'll design, implement, and deploy a distributed, fault-tolerant web application on AWS. This example will demonstrate how distributed systems work and will combine most of the knowledge in this book.

16.3 Building a fault-tolerant web application: Imagery

Before you begin the architecture and design of the fault-tolerant Imagery application, we'll talk briefly about what the application should do. A user should be able to upload an image. This image is then transformed with a sepia filter so that it looks fancy. The user can then view the sepia image. Figure 16.7 shows the process.

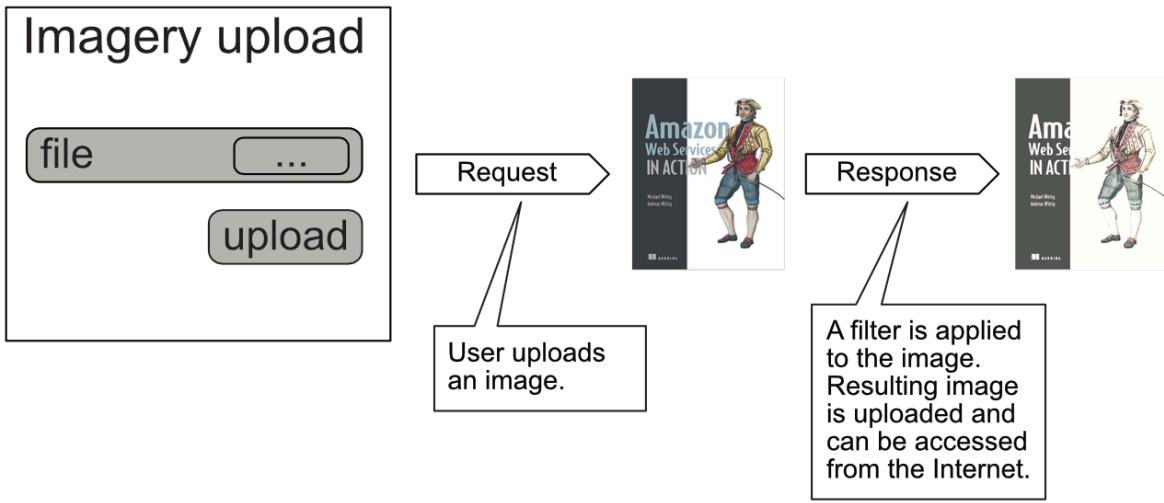


Figure 16.7 The user uploads an image to Imagery, where a filter is applied

The problem with the process shown in figure 16.7 is that it's synchronous. If the web server crashes during request and response, the user's image won't be processed. Another problem arises when many users want to use the Imagery app: the system becomes busy and may slow down or stop working. Therefore the process should be turned into an asynchronous one. Chapter 15 introduced the idea of asynchronous decoupling by using an SQS message queue, as shown in figure 16.8.

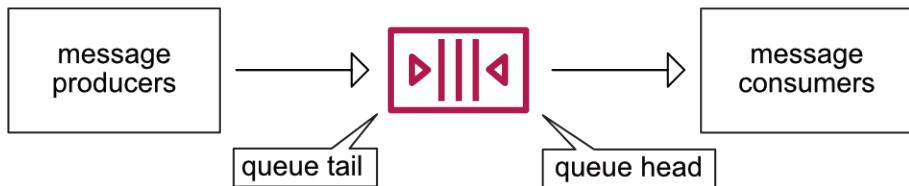


Figure 16.8 Producers send messages to a message queue while consumers read messages.

When designing an asynchronous process, it's important to keep track of the process. You need some kind of identifier for it. When a user wants to upload an image, the user creates a process first. This returns a unique ID. With that ID, the user can upload an image. If the image upload is finished, the worker begins to process the image in the background. The user can look up the process at any time with the process ID. While the image is being processed, the user can't see the sepia image. But as soon as the image is processed, the lookup process returns the sepia image. Figure 16.9 shows the asynchronous process.

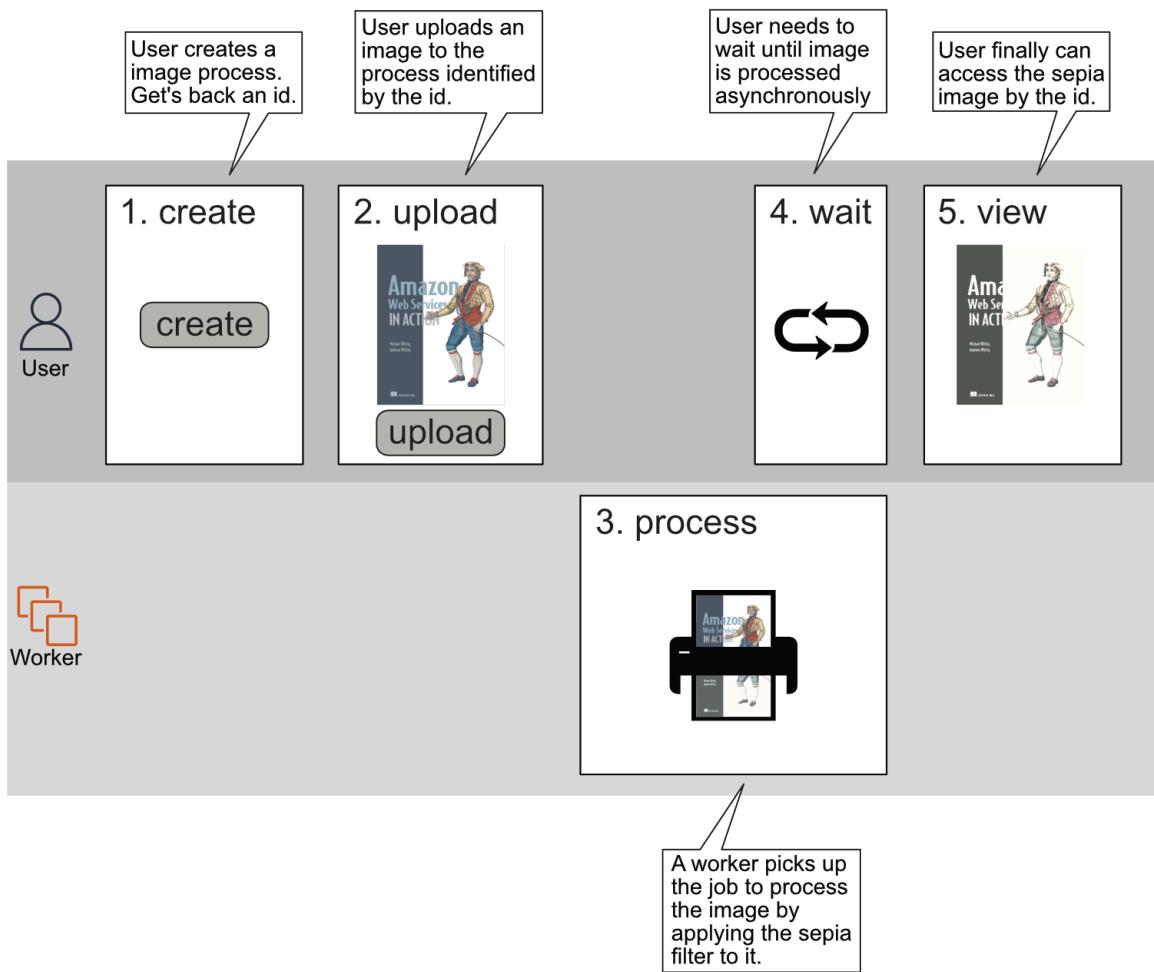


Figure 16.9 The user asynchronously uploads an image to Imagery, where a filter is applied

Now that you have an asynchronous process, it's time to map that process to AWS services. Keep in mind that most services on AWS are fault-tolerant by default, so it makes sense to pick them whenever possible. Figure [16.10](#) shows one way of doing it.

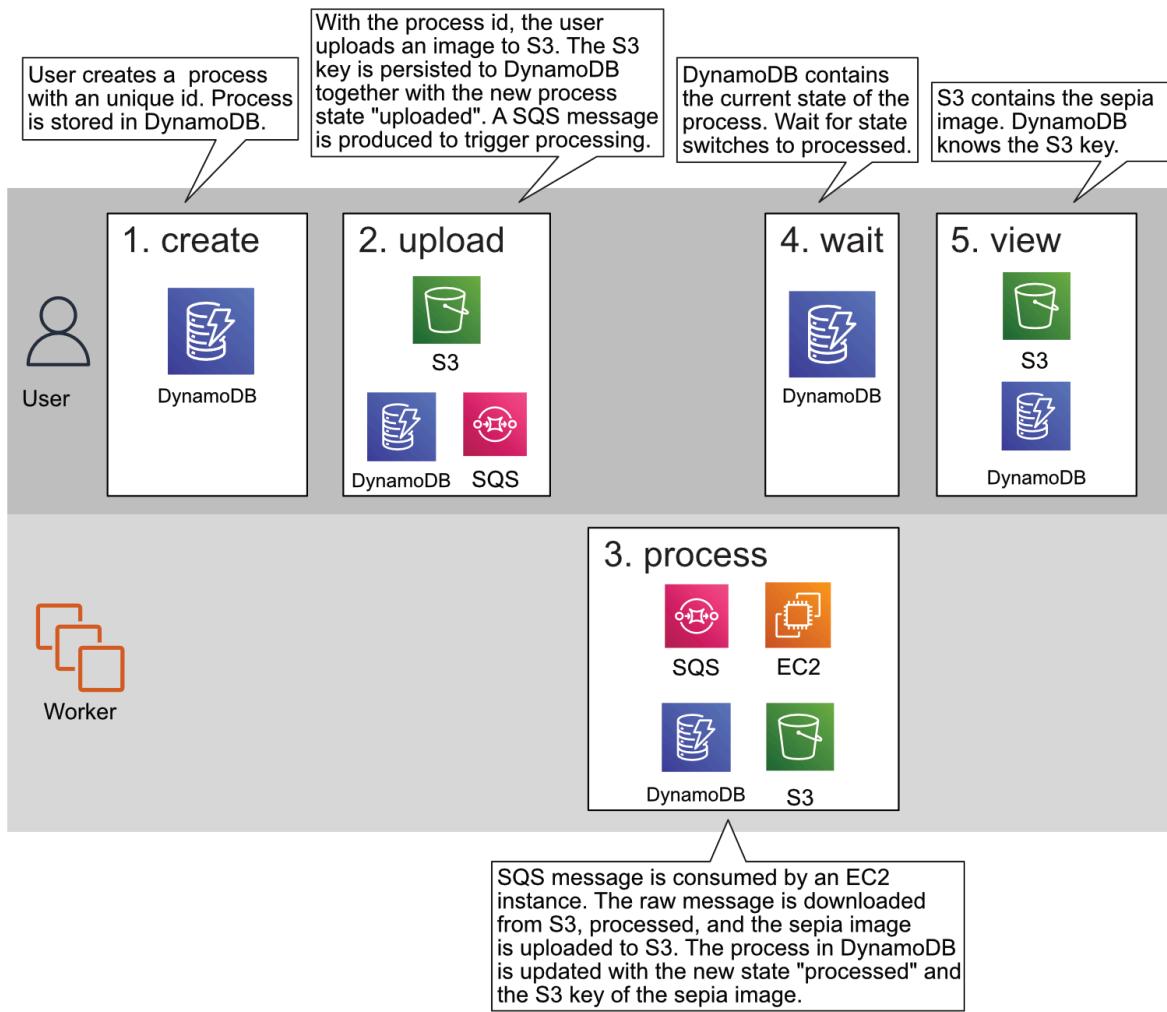


Figure 16.10 Combining AWS services to implement the asynchronous Imagery process

To make things as easy as possible, all the actions will be accessible via a REST API, which will be provided by EC2 instances. In the end, EC2 instances will provide the process and make calls to all the AWS services shown in figure 16.10.

You'll use many AWS services to implement the Imagery application. Most of them are fault-tolerant by default, but EC2 isn't. You'll deal with that problem using an idempotent state machine, as introduced in the next section.

SIDE BAR**Example is 100% covered by the Free Tier**

The example in this chapter is totally covered by the Free Tier. As long as you don't run the example longer than a few days, you won't pay anything for it. Keep in mind that this applies only if you created a fresh AWS account for this book and there is nothing else going on in your AWS account. Try to complete the example within a few days, because you'll clean up your account at the end of the section

16.3.1 The idempotent state machine

An idempotent state machine sounds complicated. We'll take some time to explain it because it's the heart of the Imagery application. Let's look at what a *state machine* is and what *idempotent* means in this context.

THE FINITE STATE MACHINE

A finite state machine has at least one start state and one end state. Between the start and the end state, the state machine can have many other states. The machine also defines transitions between states. For example, a state machine with three states could look like this:

```
(A) -> (B) -> (C).
```

This means:

- State A is the start state.
- There is a transition possible from state A to B.
- There is a transition possible from state B to C.
- State C is the end state.

But there is no transition possible between (A) → (C) or (B) → (A). With this in mind, we apply the theory to our Imagery example. The Imagery state machine could look like this:

```
(Created) -> (Uploaded) -> (Processed)
```

Once a new process (state machine) is created, the only transition possible is to Uploaded. To make this transition happen, you need the S3 key of the uploaded raw image. So the transition between `Created` → `Uploaded` can be defined by the function `uploaded(s3Key)`. Basically, the same is true for the transition `Uploaded` → `Processed`. This transition can be done with the S3 key of the sepia image: `processed(s3Key)`.

Don't be confused by the fact that the upload and the image filter processing don't appear in the state machine. These are the basic actions that happen, but we're only interested in the results; we don't track the progress of the actions. The process isn't aware that 10% of the data has been uploaded or 30% of the image processing is done. It only cares whether the actions are 100%

done. You can probably imagine a bunch of other states that could be implemented, but we're skipping that for the purpose of simplicity in this example: resized and shared are just two examples.

IDEMPOTENT STATE TRANSITIONS

An idempotent state transition must have the same result no matter how often the transition takes place. If you can make sure that your state transitions are idempotent, you can do a simple trick: in case of a failure during transitioning, you retry the entire state transition.

Let's look at the two state transitions you need to implement. The first transition `Created` → `Uploaded` can be implemented like this (pseudo code):

```
uploaded(s3Key) {
    process = DynamoDB.getItem(processId)
    if (process.state !== 'Created') {
        throw new Error('transition not allowed')
    }
    DynamoDB.updateItem(processId, {'state': 'Uploaded', 'rawS3Key': s3Key})
    SQS.sendMessage({'processId': processId, 'action': 'process'})
}
```

The problem with this implementation is that it's not idempotent. Imagine that `SQS.sendMessage` fails. The state transition will fail, so you retry. But the second call to `uploaded(s3Key)` will throw a “transition not allowed” error because `DynamoDB.updateItem` was successful during the first call.

To fix that, you need to change the `if` statement to make the function idempotent (pseudo code):

```
uploaded(s3Key) {
    process = DynamoDB.getItem(processId)
    if (process.state !== 'Created' && process.state !== 'Uploaded') {
        throw new Error('transition not allowed')
    }
    DynamoDB.updateItem(processId, {'state': 'Uploaded', 'rawS3Key': s3Key})
    SQS.sendMessage({'processId': processId, 'action': 'process'})
}
```

If you retry now, you'll make multiple updates to Dynamo, which doesn't hurt. And you may send multiple SQS messages, which also doesn't hurt, because the SQS message consumer must be idempotent as well. The same applies to the transition `Uploaded` → `Processed`.

One little thing is still missing. So far, the code will fetch an item from DynamoDB and will update the item a few lines after that. In between, another process might have set the state to `Uploaded` already. Luckily, the database supports conditional updates which allows us to reduce all the logic into a single DynamoDB request. DynamoDB will evaluate the condition before updating the item (pseudo code):

```

uploaded(s3Key) {
    process = DynamoDB.getItem(processId)
    DynamoDB.updateItem(processId, {
        'state': 'Uploaded',
        'rawS3Key': s3Key,
        condition: 'NOT state IN(Created, Uploaded)'
    })
    SQS.sendMessage({'processId': processId, 'action': 'process'});
}

```

Next, you'll begin to implement the Imagery server.

16.3.2 Implementing a fault-tolerant web service

We'll split the Imagery application into two parts: the web servers and the workers. As illustrated in figure [16.11](#), the web servers provide the REST API to the user, and the workers process images.

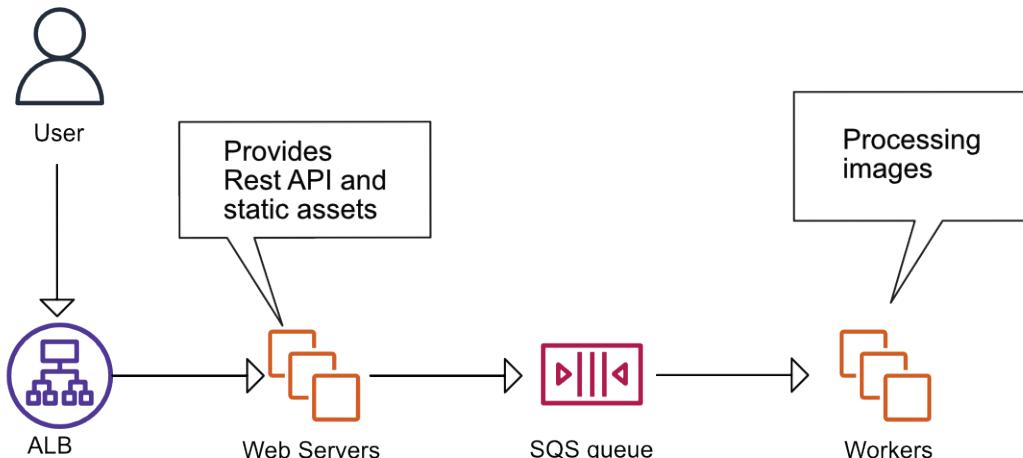


Figure 16.11 The Imagery application consists of two parts: the web servers and the workers

SIDEBAR

Where is the code located?

As usual, you'll find the code in the book's code repository on GitHub: <https://github.com/AWSinAction/code3>. Imagery is located in /chapter16/.

The REST API will support the following routes:

- POST /image—A new image process is created when executing this route.
- GET /image/:id—This route returns the state of the process specified with the path parameter :id.
- POST /image/:id/upload—This route offers a file upload for the process specified

with the `path` parameter `:id`.

To implement the web server, you'll again use Node.js and the Express web application framework. You'll use the Express framework, but don't feel intimidated, as you won't need to understand it in depth to follow along.

SETTING UP THE WEB SERVER PROJECT

As always, you need some boilerplate code to load dependencies, initial AWS endpoints, and things like that. Listing 16.1 explains the code to do so.

Listing 16.1 Initializing the Imagery server (server/server.js)

```
const express = require('express');
const bodyParser = require('body-parser');
const AWS = require('aws-sdk');
const { v4: uuidv4 } = require('uuid');
const multiparty = require('multiparty');

const db = new AWS.DynamoDB({});      ①
const sqs = new AWS.SQS({});        ②
const s3 = new AWS.S3({});          ③

const app = express();            ④
app.use(bodyParser.json());       ⑤

// [...]

app.listen(process.env.PORT || 8080, function() {           ⑥
  console.log('Server started. Open http://localhost:' +
  ↵ + (process.env.PORT || 8080) + ' with browser.');
});
```

- ① Load Node.js modules (dependencies)
- ② Creates a DynamoDB endpoint
- ③ Creates an SQS endpoint
- ④ Creates an S3 endpoint
- ⑤ Creates an Express application
- ⑥ Tells Express to parse the request bodies
- ⑦ Starts Express on the port defined by the environment variable PORT, or defaults to 8080

Don't worry too much about the boilerplate code; the interesting parts will follow.

CREATING A NEW IMAGERY PROCESS

To provide a REST API to create image processes, a fleet of EC2 instances will run Node.js code behind a load balancer. The image processes will be stored in DynamoDB. Figure 16.12 shows the flow of a request to create a new image process.

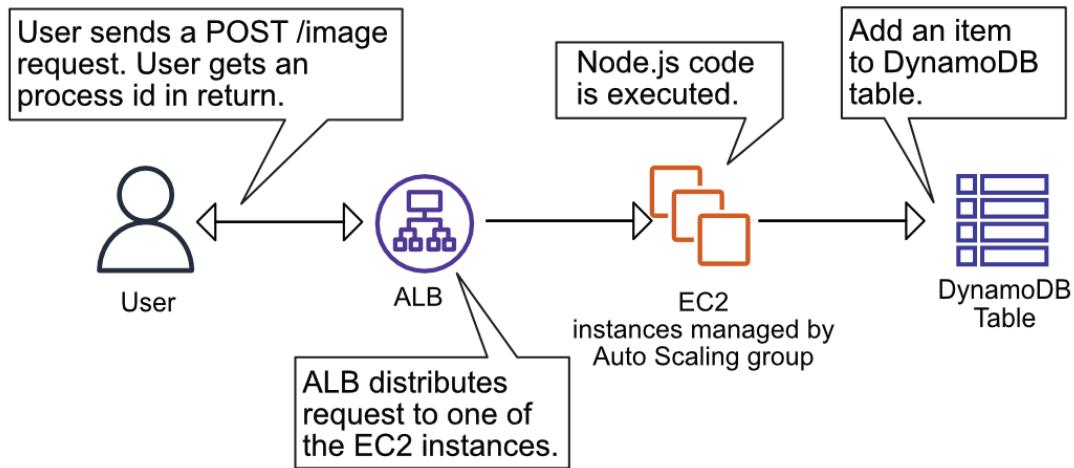


Figure 16.12 Creating a new image process in Imagery

You'll now add a route to the Express application to handle `POST /image` requests, as shown in listing 16.2.

Listing 16.2 Imagery server: `POST /image` creates an image process (server/server.js)

```
app.post('/image', function(request, response) {          ①
  const id = uuidv4();                                ②
  db.putItem({                                         ③
    'Item': {
      'id': {
        'S': id                                     ④
      },
      'version': {
        'N': '0'                                    ⑤
      },
      'created': {
        'N': Date.now().toString()                 ⑥
      },
      'state': {
        'S': 'created'                            ⑦
      }
    },
    'TableName': 'imagery-image',                     ⑧
    'ConditionExpression': 'attribute_not_exists(id)' ⑨
  }, function(err, data) {
    if (err) {
      throw err;
    } else {
      response.json({'id': id, 'state': 'created'}); ⑩
    }
  });
});
```

- ① Registers the route with Express
- ② Creates a unique ID for the process
- ③ Invokes the `putItem` operation on DynamoDB

- ④ The id attribute will be the primary key in DynamoDB.
- ⑤ Use the version for optimistic locking (explained in the following sidebar).
- ⑥ Stores the date and time when the process was created.
- ⑦ The process is now in the created state: this attribute will change when state transitions happen.
- ⑧ The DynamoDB table will be created later in the chapter.
- ⑨ Prevents the item from being replaced if it already exists.
- ⑩ Responds with the process ID

A new process can now be created.

SIDE BAR

Optimistic locking

To prevent multiple updates to an DynamoDB item, you can use a trick called optimistic locking. When you want to update an item, you must specify which version you want to update. If that version doesn't match the current version of the item in the database, your update will be rejected. Keep in mind that optimistic locking is your responsibility, not a default available in DynamoDB. DynamoDB only provides the features to implement optimistic locking.

Imagine the following scenario. An item is created in version 0. Process A looks up that item (version 0). Process B also looks up that item (version 0). Now process A wants to make a change by invoking the `updateItem` operation on DynamoDB. Therefore process A specifies that the expected version is 0. DynamoDB will allow that modification, because the version matches; but DynamoDB will also change the item's version to 1 because an update was performed. Now process B wants to make a modification and sends a request to DynamoDB with the expected item version 0. DynamoDB will reject that modification because the expected version doesn't match the version DynamoDB knows of, which is 1.

To solve the problem for process B, you can use the same trick introduced earlier: `retry`. Process B will again look up the item, now in version 1, and can (you hope) make the change. There is one problem with optimistic locking: If many modifications happen in parallel, a lot of overhead is created because of many retries. But this is only a problem if you expect a lot of concurrent writes to a single item, which can be solved by changing the data model. That's not the case in the Imagery application. Only a few writes are expected to happen for a single item: optimistic locking is a perfect fit to make sure you don't have two writes where one overrides changes made by another.

The opposite of optimistic locking is pessimistic locking. A pessimistic lock strategy can be implemented by using a semaphore. Before you change data, you need to lock the semaphore. If the semaphore is already locked, you wait until the semaphore becomes free again.

The next route you need to implement is to look up the current state of a process.

LOOKING UP AN IMAGERY PROCESS

You'll now add a route to the Express application to handle GET /image/:id requests. Figure 16.13 shows the request flow.

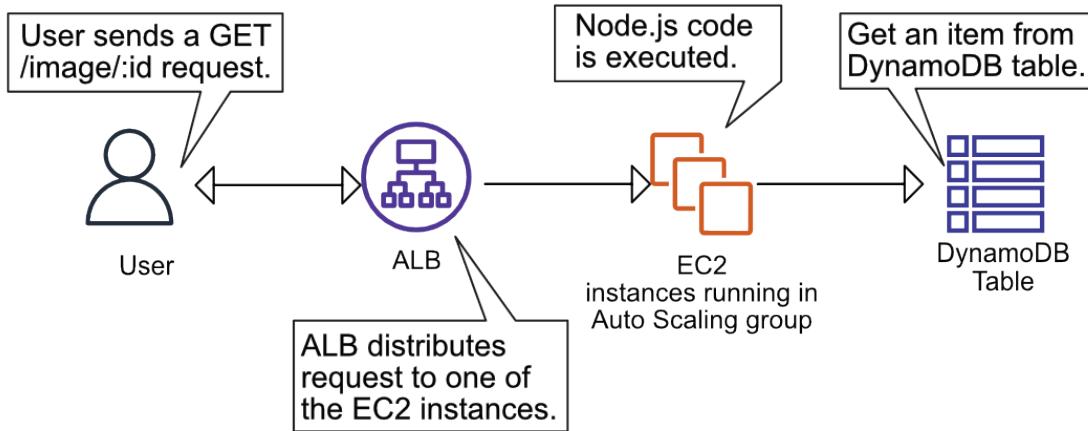


Figure 16.13 Looking up an image process in Imagery to return its state

Express will take care of the path parameter :id; Express will provide it within `request.params.id`. The implementation shown in listing 16.3 needs to get an item from DynamoDB based on the path parameter ID.

Listing 16.3 GET /image/:id looks up an image process (server/server.js)

```

function mapImage = function(item) {          ①
  return {
    'id': item.id.S,
    'version': parseInt(item.version.N, 10),
    'state': item.state.S,
    'rawS3Key': // [...]
    'processedS3Key': // [...]
    'processedImage': // [...]
  };
};

function getImage(id, cb) {                  ②
  db.getItem({
    'Key': {
      'id': {                      ③
        'S': id
      }
    },
    'TableName': 'imagery-image'
  }, function(err, data) {
    if (err) {
      cb(err);
    } else {
      if (data.Item) {
        cb(null, mapImage(data.Item));
      } else {
        cb(new Error('image not found'));
      }
    }
  });
}

app.get('/image/:id', function(request, response) { ④
  getImage(request.params.id, function(err, image) {
    if (err) {
      throw err;
    } else {
      response.json(image);           ⑤
    }
  });
});

```

- ① Helper function to map a DynamoDB result to a JavaSscript object
- ② Invokes the getItem operation on DynamoDB
- ③ id is the partition key.
- ④ Registers the route with Express
- ⑤ Responds with the image process

The only thing missing is the upload part, which comes next.

UPLOADING AN IMAGE

Uploading an image via POST request requires several steps:

1. Upload the raw image to S3.

2. Modify the item in DynamoDB.
3. Send an SQS message to trigger processing.

Figure 16.14 shows this flow.

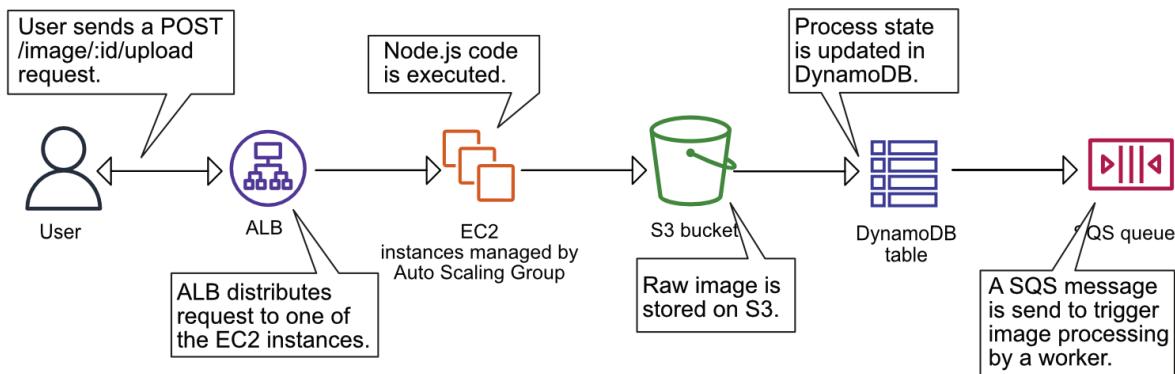


Figure 16.14 Uploading a raw image to Imagery and trigger image processing

Listing 16.4 shows the implementation of these steps.

Listing 16.4 POST /image/:id/upload uploads an image (server/server.js)

```

function uploadImage(image, part, response) {
  const rawS3Key = 'upload/' + image.id + '-' + Date.now();      ①
  s3.putObject({      ②
    'Bucket': process.env.ImageBucket,      ③
    'Key': rawS3Key,
    'Body': part,      ④
    'ContentLength': part.byteCount
  }, function(err, data) {
    if (err) { /* [...] */ } else {
      db.updateItem({      ⑤
        'Key': {'id': {'S': image.id}},
        'UpdateExpression': 'SET #s=:newState,          ⑥
➥ version=:newVersion, rawS3Key=:rawS3Key',
        'ConditionExpression': 'attribute_exists(id)      ⑦
➥ AND version=:oldVersion
➥ AND #s IN (:stateCreated, :stateUploaded),
        'ExpressionAttributeNames': {'#s': 'state'},
        'ExpressionAttributeValues': {
          ':newState': {'S': 'uploaded'},
          ':oldVersion': {'N': image.version.toString()},
          ':newVersion': {'N': (image.version + 1).toString()},
          ':rawS3Key': {'S': rawS3Key},
          ':stateCreated': {'S': 'created'},
          ':stateUploaded': {'S': 'uploaded'}
        },
        'ReturnValues': 'ALL_NEW',
        'TableName': 'imagery-image'
      }, function(err, data) {
        if (err) { /* [...] */ } else {
          sqs.sendMessage({      ⑧
            'MessageBody': JSON.stringify({
              'imageId': image.id, 'desiredState': 'processed'
            }),
            'QueueUrl': process.env.ImageQueue,      ⑩
          }, function(err) {
            if (err) {
              throw err;
            } else {
              response.redirect('/#view=' + image.id);
              response.end();
            }
          });
        }
      });
    }
  });
}

app.post('/image/:id/upload', function(request, response) {      ⑪
  getImage(request.params.id, function(err, image) {
    if (err) { /* [...] */ } else {
      const form = new multiparty.Form();      ⑫
      form.on('part', function(part) {
        uploadImage(image, part, response);
      });
      form.parse(request);
    }
  });
});
}

```

- ① Creates a key for the S3 object
- ② Calls the S3 API to upload an object

- ③ The S3 bucket name is passed in as an environment variable (the bucket will be created later in the chapter).
- ④ The body is the uploaded stream of data.
- ⑤ Calls the DynamoDB API to update an object.
- ⑥ Updates the state, version, and raw S3 key
- ⑦ Updates only when item exists. Version equals the expected version, and state is one of those allowed.
- ⑧ Calls the SQS API to publish a message
- ⑨ Creates the message body containing the image's ID and the desired state
- ⑩ The queue URL is passed in as an environment variable.
- ⑪ Registers the route with Express
- ⑫ We are using the multiparty module to handle multi-part uploads.

The server side is finished. Next you'll continue to implement the processing part in the Imagery worker. After that, you can deploy the application.

16.3.3 Implementing a fault-tolerant worker to consume SQS messages

The Imagery worker processes images by applying a sepia filter asynchronously. The worker runs through the following step in an endless loop. It is worth noting, that multiple worker can run at the same time.

1. Poll the queue for new messages
2. Fetch process data from database
3. Download image from S3
4. Apply sepia filter to image
5. Upload modified image to S3
6. Update process state in database
7. Mark message as done by deleting from queue

SETTING UP THE WORKER

To get started you need some boilerplate code to load dependencies, initial AWS endpoints, and an endless loop to receive messages. Listing 16.5 explains the details.

Listing 16.5 Initializing the Imagery worker (worker/worker.js)

```

const AWS = require('aws-sdk');      ①
const assert = require('assert-plus');
const Jimp = require('jimp');
const fs = require('fs/promises');

const db = new AWS.DynamoDB({});    ②
const s3 = new AWS.S3({});
const sqs = new AWS.SQS({});

const states = {
  'processed': processed
};

async function processMessages() {  ③
  let data = await sqs.receiveMessage({
    QueueUrl: process.env.ImageQueue,
    MaxNumberOfMessages: 1
  }).promise();
  if (data.Messages && data.Messages.length > 0) {
    var task = JSON.parse(data.Messages[0].Body);
    var receiptHandle = data.Messages[0].ReceiptHandle;
    assert.string(task.imageId, 'imageId');  ⑤
    assert.string(task.desiredState, 'desiredState');
    let image = await getImage(task.imageId);  ⑥
    if (typeof states[task.desiredState] === 'function') {
      await states[task.desiredState](image);  ⑦
      await sqs.deleteMessage({
        QueueUrl: process.env.ImageQueue,
        ReceiptHandle: receiptHandle
      }).promise();
    } else {
      throw new Error('unsupported desiredState');
    }
  }
}

async function run() {
  while (true) {  ⑨
    try {
      await processMessages();
      await new Promise(resolve => setTimeout(resolve, 10000));  ⑩
    } catch (e) {  ⑪
      console.log('ERROR', e);
    }
  }
}

run();

```

- ① Loads Node.js modules (dependencies)
- ② Configure clients to interact with AWS services
- ③ Function reads messages from queue, processes them, and finally deletes the message from the queue
- ④ Read one message from queue, might return an empty result in case there are no messages in the queue
- ⑤ Make sure the message contains all required properties
- ⑥ Get process data from database

- ⑦ Triggers the state machine
- ⑧ If message was processed successfully, this deletes the message from the queue
- ⑨ A loop running endlessly
- ⑩ Sleep for 10 seconds
- ⑪ Catch all exceptions, ignore them, and try again

The Node.js module `jimp` is used to create sepia images. You'll wire that up next.

HANDLING SQS MESSAGES AND PROCESSING THE IMAGE

The SQS message to trigger the image processing is handled in the worker. Once a message is received, the worker starts to download the raw image from S3, applies the sepia filter, and uploads the processed image back to S3. After that, the process state in DynamoDB is modified. Figure 16.15 shows the steps.

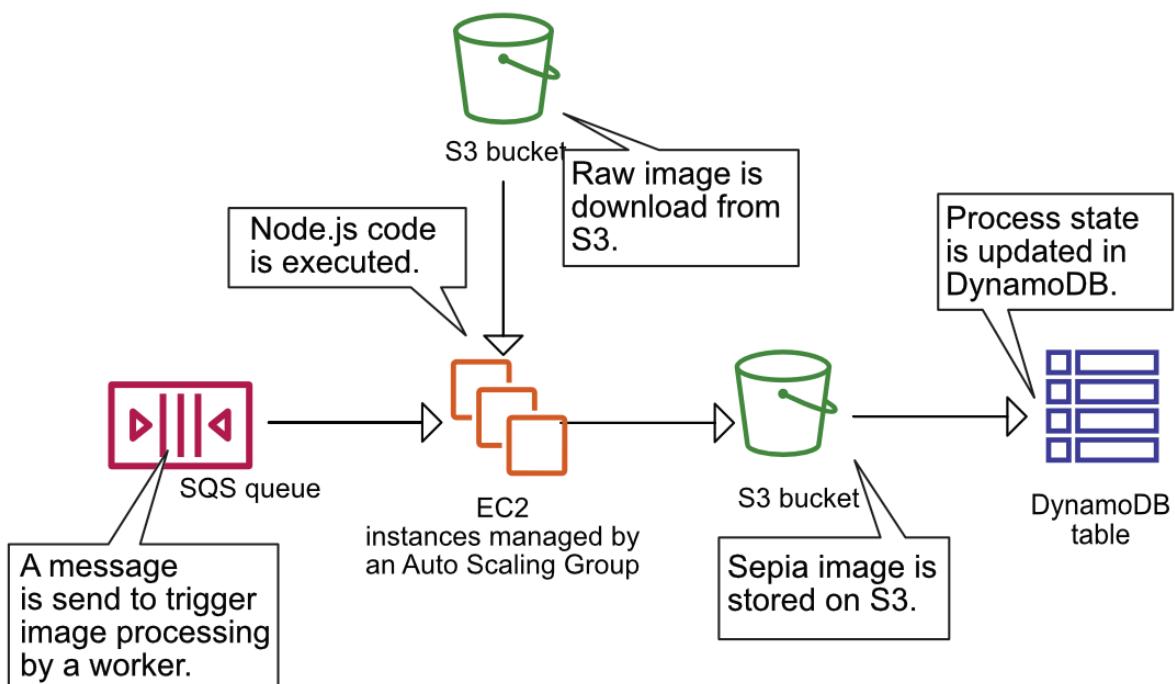


Figure 16.15 Processing a raw image to upload a sepia image to S3

Listing 16.6 shows the code to process an image.

Listing 16.6 Imagery worker: handles SQS messages (worker/worker.js)

```

async function processImage(image) {
  let processedS3Key = 'processed/' + image.id + '-' + Date.now() + '.png';
  let rawFile = './tmp_raw_' + image.id;
  let processedFile = './tmp_processed_' + image.id;
  let data = await s3.getObject({  
    ①  
    'Bucket': process.env.ImageBucket,  
    'Key': image.rawS3Key  
}).promise();
  await fs.writeFile(rawFile, data.Body, {'encoding': null});  
  ②
  let lenna = await Jimp.read(rawFile);  
  ③
  await lenna.sepia().write(processedFile);  
  ④
  await fs.unlink(rawFile);  
  ⑤
  let buf = await fs.readFile(processedFile, {'encoding': null});  
  ⑥
  await s3.putObject({  
    'Bucket': process.env.ImageBucket,  
    'Key': processedS3Key,  
    'ACL': 'public-read',  
    'Body': buf,  
    'ContentType': 'image/png'  
}).promise();
  await fs.unlink(processedFile);  
  ⑧
  return processedS3Key;
}

async function processed(image) {
  let processedS3Key = await processImage(image);
  await db.updateItem({  
    ⑨  
    'Key': {  
      'id': {  
        'S': image.id  
      }  
    },  
    'UpdateExpression':  
      'SET #s=:newState, version=:newVersion,  
    ↵ processedS3Key=:processedS3Key',  
      ⑩  
    'ConditionExpression':  
      'attribute_exists(id) AND version=:oldVersion  
    ↵ AND #s IN (:stateUploaded, :stateProcessed)',  
      ⑪  
    'ExpressionAttributeNames': {  
      '#s': 'state'  
    },  
    'ExpressionAttributeValues': {  
      ':newState': {'S': 'processed'},  
      ':oldVersion': {'N': image.version.toString()},  
      ':newVersion': {'N': (image.version + 1).toString()},  
      ':processedS3Key': {'S': processedS3Key},  
      ':stateUploaded': {'S': 'uploaded'},  
      ':stateProcessed': {'S': 'processed'}  
    },  
    'ReturnValues': 'ALL_NEW',  
    'TableName': 'imagery-image'  
  }).promise();
}

```

- ① Fetch original image from S3
- ② Write original image to a temporary folder on disk
- ③ Read file with image manipulation library
- ④ Apply sepia filter and write processed image to disk
- ⑤ Delete original image from temporary folder

- ⑥ Read processed image
- ⑦ Upload processed image to S3
- ⑧ Delete processed file from temporary folder
- ⑨ Update the database item by calling the updateItem operation
- ⑩ Updates the state, version, and processed S3 key
- ⑪ Updates only when an item exists, version equals the expected version, and state is one of those allowed.

The worker is ready to manipulate your images. The next step is to deploy all that code to AWS in a fault-tolerant way.

16.3.4 Deploying the application

As done before, you'll use CloudFormation to deploy the application. The infrastructure consists of the following building blocks.

- A S3 bucket for raw and processed images
- A DynamoDB table `imagery-image`
- A SQS queue and dead-letter queue
- An Application Load Balancer (ALB)
- Two auto-scaling groups to manage EC2 instances running the server and worker
- IAM roles for the server and worker instances

It takes quite a while to create that CloudFormation stack; that's why you should do so now. After you've created the stack, we'll look at the template. After that, the application should be ready to use.

To help you deploy Imagery, we created a CloudFormation template located at <http://s3.amazonaws.com/awsinaction-code3/chapter16/template.yaml>. Create a stack based on that template. The stack output `EndpointURL` returns the URL that you can access from your browser to use Imagery. Here's how to create the stack from the terminal:

```
$ aws cloudformation create-stack --stack-name imagery \
  --template-url https://s3.amazonaws.com/\
  awsinaction-code3/chapter16/template.yaml \
  --capabilities CAPABILITY_IAM
```

Next, let's have a look what is going on behind the scenes.

BUNDLING RUNTIME AND APPLICATION INTO A MACHINE IMAGE (AMI)

In chapter 4, we introduced the concept of immutable machines. The idea is to create an Amazon Machine Image (AMI) containing the runtime, all required libraries, as well as the application's code or binary. The AMI is then used to launch EC2 instances with everything pre-installed. To deliver a new version of your application, you would create a new image, launch new instances, and terminate the old instances. We used Packer by HashiCorp to build AMIs. Check out chapter 4, if you want to recap the details. All we want to show here, is the configuration file, we used to pre-build and share AMIs containing the imagery worker and server with you.

Listing 16.7 explains the configuration file we used to build the AMIs for you. Please note, you do not need to run Packer to build your own AMIs. We have done so already, and shared the AMIs publicly.

Find the Packer configuration file at `chapter16/imagery.pkr.hcl` in our source code repository at <https://github.com/AWSinAction/code3>.

Listing 16.7 The configuration for Packer to build an AMI containing the imagery app

```

packer {
    required_plugins {
        amazon = {
            version = ">= 0.0.2"
            source  = "github.com/hashicorp/amazon"
        }
    }
}

source "amazon-ebs" "imagery" {
    ami_name      = "awsinaction-imagery-{{timestamp}}"④
    tags = {
        Name = "awsinaction-imagery"
    }
    instance_type = "t2.micro"⑥
    region       = "us-east-1"⑦
    source_ami_filter {
        filters = {
            name          = "amzn2-ami-hvm-2.0.*-x86_64-gp2"
            root-device-type = "ebs"
            virtualization-type = "hvm"
        }
        most_recent = true
        owners      = ["137112412989"]
    }
    ssh_username = "ec2-user"⑨
    ami_groups  = ["all"]⑩
    ami_regions = [
        "us-east-1",
        # [...]
    ]
}

build {⑫
    name      = "awsinaction-imagery"⑬
    sources  = [
        "source.amazon-ebs.imagery"⑭
    ]

    provisioner "file" {
        source = "./"⑮
        destination = "/home/ec2-user/"⑯
    }

    provisioner "shell" {⑰
        inline = [
            "curl -sL https://rpm.nodesource.com/setup_14.x | sudo bash -",⑱
            "sudo yum update",
            "sudo yum install -y nodejs cairo-devel libjpeg-turbo-devel",⑲
            "cd server/ && npm install && cd -",⑳
            "cd worker/ && npm install && cd -"
        ]
    }
}

```

- ^① Initializes and configure Packer
- ^② Adds the plugin required to build AMIs
- ^③ Configures how Packer will create the AMI
- ^④ The name for the AMI created by Packer
- ^⑤ The tags for the AMI created by Packer

- ⑥ The instance type used by Packer when spinning up a virtual machine to build the AMI
- ⑦ The region used by Packer to create the AMI
- ⑧ The filter describes how to find the base AMI - the latest version of Amazon Linux 2 - to start from
- ⑨ The user name required to connect to the build instance via SSH
- ⑩ Allow anyone to access the AMI
- ⑪ Copy the AMI to all commercial regions
- ⑫ Configures the steps Packer executes while building the image
- ⑬ The name for the build
- ⑭ The sources for the build (references source from above)
- ⑮ Copies all files and folders from the current directory ...
- ⑯ ... to the home directory of the EC2 instance used to build the AMI
- ⑰ Executes a shell script on the EC2 instance used to build the AMI
- ⑱ Add repository for Node.js 14, the runtime for imagery server and worker
- ⑲ Install Node.js and libraries needed to manipulate images
- ⑳ Install Node.js packages for server and worker

Next, you will learn about how to deploy the infrastructure with the help of CloudFormation.

DEPLOYING S3, DYNAMODB, AND SQS

Listing [16.8](#) describes the VPC, S3 bucket, DynamoDB table, and SQS queue.

Listing 16.8 Imagery CloudFormation template: S3, DynamoDB, and SQS

```
---
AWSTemplateFormatVersion: '2010-09-09'
Description: 'AWS in Action: chapter 16'
Mappings:
  RegionMap: ①
    'us-east-1':
      AMI: 'ami-0ad3c79dfb359f1ba'
    # [...]
Resources:
  VPC: ②
    Type: 'AWS::EC2::VPC'
    Properties:
      CidrBlock: '172.31.0.0/16'
      EnableDnsHostnames: true
    # [...]
  Bucket: ③
    Type: 'AWS::S3::Bucket'
    Properties:
      BucketName: !Sub 'imagery-${{AWS::AccountId}}' ④
      WebsiteConfiguration:
        ErrorDocument: error.html
        IndexDocument: index.html
  Table: ⑤
    Type: 'AWS::DynamoDB::Table'
    Properties:
      AttributeDefinitions:
        - AttributeName: id ⑥
          AttributeType: S
      KeySchema:
        - AttributeName: id
          KeyType: HASH
      ProvisionedThroughput:
        ReadCapacityUnits: 1
        WriteCapacityUnits: 1
      TableName: 'imagery-image'
  SQSDLQueue: ⑦
    Type: 'AWS::SQS::Queue'
    Properties:
      QueueName: 'imagery-dlq'
  SQSQueue: ⑧
    Type: 'AWS::SQS::Queue'
    Properties:
      QueueName: imagery
      RedrivePolicy:
        deadLetterTargetArn: !Sub '${{SQSDLQueue.Arn}}' ⑨
        maxReceiveCount: 10
    # [...]
Outputs:
  EndpointURL: ⑩
    Value: !Sub 'http://${LoadBalancer.DNSName}'
    Description: Load Balancer URL
```

- ① The map contains key-value pairs mapping regions to AMIs built by us including the imagery server and worker.
- ② The CloudFormation template contains a typical public VPC configuration
- ③ A S3 bucket for uploaded and processed images, with web hosting enabled
- ④ The bucket name contains the account ID to make the name unique.
- ⑤ DynamoDB table containing the image processes

- ⑥ The id attribute is used as the partition key.
- ⑦ SQS queue that receives messages that can't be processed
- ⑧ SQS queue to trigger image processing
- ⑨ If a message is received more than 10 times, it's moved to the dead-letter queue.
- ⑩ Visit the output with your browser to use Imagery.

The concept of a *dead-letter queue (DLQ)* needs a short introduction here as well. If a single SQS message can't be processed, the message becomes visible again on the queue for other workers. This is called a *retry*. But if for some reason every retry fails (maybe you have a bug in your code), the message will reside in the queue forever and may waste a lot of resources because of all the retries. To avoid this, you can configure a dead-letter queue. If a message is retried more than a specific number of times, it's removed from the original queue and forwarded to the DLQ. The difference is that no worker listens for messages on the DLQ. But you should create a CloudWatch alarm that triggers if the DLQ contains more than zero messages, because you need to investigate this problem manually by looking at the message in the DLQ. Once the bug is fixed you can move the messages from the dead letter queue back to the original queue to process them again.

Now that the basic resources have been designed, let's move on to the more specific resources.

IAM ROLES FOR SERVER AND WORKER EC2 INSTANCES

Remember that it's important to only grant the privileges that are necessary. All server instances must be able to do the following:

- `sqs:SendMessage` to the SQS queue created in the template to trigger image processing
- `s3:PutObject` to the S3 bucket created in the template to upload a file to S3 (you can further limit writes to the `upload/` key prefix)
- `dynamodb:GetItem`, `dynamodb:PutItem`, and `dynamodb:UpdateItem` to the DynamoDB table created in the template

All worker instances must be able to do the following:

- `sqs:DeleteMessage`, and `sqs:ReceiveMessage` to the SQS queue created in the template
- `s3:PutObject` to the S3 bucket created in the template to upload a file to S3 (you can further limit writes to the `processed/` key prefix)
- `dynamodb:.GetItem` and `dynamodb:UpdateItem` to the DynamoDB table created in the template

Both, the role for servers and workers need to grant access for the AWS Systems Manager to enable acces via Session Manager.

- `ssmmessages:*`
- `ssm:UpdateInstanceInformation`
- `ec2messages:*`

If you don't feel comfortable with IAM roles, take a look at the book's code repository on GitHub at <https://github.com/AWSinAction/code3>. The template with IAM roles can be found in `/chapter16/template.yaml`.

Now it's time to deploy the server.

DEPLOYING THE SERVER WITH A LOAD BALANCER AND AN AUTO-SCALING GROUP

The imagery server allows user to upload images, monitor the processing, and show the results. An Application Load Balancer (ALB) acts as the entry point into the system. Behind the load balancer a fleet of servers running on EC2 instances answers incoming HTTP requests. An auto-scaling group ensures EC2 instances are up and running and replaces instances that fail the load balancers's health check.

Listing [16.9](#) shows how to create the a load balancer with the help of CloudFormation.

Listing 16.9 CloudFormation template: load balancer for imagery server

```

LoadBalancer: ①
  Type: 'AWS::ElasticLoadBalancingV2::LoadBalancer'
  Properties:
    Subnets:
      - Ref: SubnetA
      - Ref: SubnetB
    SecurityGroups:
      - !Ref LoadBalancerSecurityGroup
    Scheme: 'internet-facing'
  DependsOn: VPCGatewayAttachment
LoadBalancerListener: ②
  Type: 'AWS::ElasticLoadBalancingV2::Listener'
  Properties:
    DefaultActions:
      - Type: forward ③
        TargetGroupArn: !Ref LoadBalancerTargetGroup
    LoadBalancerArn: !Ref LoadBalancer
    Port: 80 ④
    Protocol: HTTP
LoadBalancerTargetGroup: ⑤
  Type: 'AWS::ElasticLoadBalancingV2::TargetGroup'
  Properties:
    HealthCheckIntervalSeconds: 5 ⑥
    HealthCheckPath: '/'
    HealthCheckPort: 8080
    HealthCheckProtocol: HTTP
    HealthCheckTimeoutSeconds: 3
    HealthyThresholdCount: 2
    UnhealthyThresholdCount: 2
    Matcher:
      HttpCode: '200,302'
    Port: 8080 ⑦
    Protocol: HTTP
    VpcId: !Ref VPC
LoadBalancerSecurityGroup: ⑧
  Type: 'AWS::EC2::SecurityGroup'
  Properties:
    GroupDescription: 'awsinaction-elb-sg'
    VpcId: !Ref VPC
    SecurityGroupIngress:
      - CidrIp: '0.0.0.0/0' ⑨
        FromPort: 80
        IpProtocol: tcp
        ToPort: 80

```

- ① The load balancer distributing incoming requests among a fleet of virtual machines
- ② Configures a listener for the load balancer
- ③ The HTTP listener forwards all requests to the default target group defined below
- ④ The listener will listen for HTTP requests on port 80/TCP
- ⑤ The default target group
- ⑥ The target group will check the health of registered EC2 instances by sending HTTP requests on port 8080/TCP
- ⑦ By default the target group will forward requests to port 8080/TCP of registered virtual machines
- ⑧ A security group for the load balancer

- ⑨ Allows incoming traffic on port 80/TCP from anywhere

Next, listing 16.10 shows how to create an auto scaling group to launch EC2 instances and register them at the load balancer.

Listing 16.10 CloudFormation template: auto-scaling group for imagery server

```

ServerSecurityGroup: ①
  Type: 'AWS::EC2::SecurityGroup'
  Properties:
    GroupDescription: 'imagery-worker'
    VpcId: !Ref VPC
    SecurityGroupIngress:
      - FromPort: 8080 ②
        IpProtocol: tcp
        SourceSecurityGroupId: !Ref LoadBalancerSecurityGroup
        ToPort: 8080
  ServerLaunchTemplate: ③
    Type: 'AWS::EC2::LaunchTemplate'
    Properties:
      LaunchTemplateData:
        IamInstanceProfile:
          Name: !Ref ServerInstanceProfile
        ImageId: !FindInMap [RegionMap, !Ref 'AWS::Region', AMI] ④
        Monitoring:
          Enabled: false
        InstanceType: 't2.micro' ⑤
        NetworkInterfaces:
          - AssociatePublicIpAddress: true
            DeviceIndex: 0
            Groups:
              - !Ref ServerSecurityGroup
        UserData: ⑦
          'Fn::Base64': !Sub |
            #!/bin/bash -ex
            trap '/opt/aws/bin/cfn-signal -e 1'
        ➔ --region ${AWS::Region} --stack ${AWS::StackName}
        ➔ --resource ServerAutoScalingGroup' ERR
          cd /home/ec2-user/server/
          sudo -u ec2-user ImageQueue=${SQSQueue} ImageBucket=${Bucket}
        ➔ nohup node server.js > server.log &
          /opt/aws/bin/cfn-signal -e $? --stack ${AWS::StackName}
        ➔ --resource ServerAutoScalingGroup --region ${AWS::Region}
  ServerAutoScalingGroup: ⑧
    Type: 'AWS::AutoScaling::AutoScalingGroup'
    Properties:
      LaunchTemplate: ⑨
        LaunchTemplateId: !Ref ServerLaunchTemplate
        Version: !GetAtt 'ServerLaunchTemplate.LatestVersionNumber'
      MinSize: 1 ⑩
      MaxSize: 2
      DesiredCapacity: 1
      TargetGroupARNs: ⑫
        - !Ref LoadBalancerTargetGroup
      HealthCheckGracePeriod: 120
      HealthCheckType: ELB ⑬
      VPCZoneIdentifier:
        - !Ref SubnetA
        - !Ref SubnetB
        # [...]
      DependsOn: VPCGatewayAttachment
      # [...]

```

- ① A security group for the EC2 instances running the server

- ② Allows incoming traffic on port 8080/TCP but only from the load balancer
- ③ The launch template used as a blueprint for spinning up EC2 instances
- ④ Looks up the AMI with the Imagery server pre-installed from the region map (see listing above)
- ⑤ Launch virtual machines of type `t2.micro` to run example under the free tier.
- ⑥ Configures a network interface (ENI) with a public IP address and the security group of the server
- ⑦ Each virtual machine will execute this script at the end of the boot process. The script starts the Node.js server.
- ⑧ Create an auto-scaling group which manages the virtual machines running the Imagery server
- ⑨ References the launch template
- ⑩ The auto-scaling group will spin up at least one and no more than two EC2 instances
- ⑪ The auto-scaling group will register and deregister virtual machines at the target group
- ⑫ The auto-scaling group will replace EC2 instances that fail the target group's health check
- ⑬ Spins up EC2 instances distributed among two subnets = AZs

That's all about the server. Next, you need to deploy the worker.

DEPLOYING THE WORKER WITH AN AUTO-SCALING GROUP

Deploying the worker works similar. But instead of a load balancer, the queue is used for decoupling. Please note, that we already explained how to create a SQS in listing [16.8](#). Therefore, all that's left is the auto-scaling group and a launch template. Listing [16.11](#) shows the details.

Listing 16.11 CloudFormation template: load balancer and auto-scaling group for imagery worker

```

WorkerLaunchTemplate: ①
  Type: 'AWS::EC2::LaunchTemplate'
  Properties:
    LaunchTemplateData:
      IamInstanceProfile: ②
        Name: !Ref WorkerInstanceProfile
      ImageId: !FindInMap [RegionMap, !Ref 'AWS::Region', AMI] ③
      Monitoring: ④
        Enabled: false
      InstanceType: 't2.micro' ⑤
      NetworkInterfaces: ⑥
        - AssociatePublicIpAddress: true
          DeviceIndex: 0
        Groups:
          - !Ref WorkerSecurityGroup
      UserData: ⑦
        'Fn::Base64': !Sub |
          #!/bin/bash -ex
          trap '/opt/aws/bin/cfn-signal -e 1 --region ${AWS::Region}'
        ➔ --stack ${AWS::StackName} --resource WorkerAutoScalingGroup' ERR
          cd /home/ec2-user/worker/
          sudo -u ec2-user ImageQueue=${SQSQueue} ImageBucket=${Bucket}
        ➔ nohup node worker.js > worker.log &
          /opt/aws/bin/cfn-signal -e $? --stack ${AWS::StackName}
        ➔ --resource WorkerAutoScalingGroup --region ${AWS::Region}
WorkerAutoScalingGroup: ⑧
  Type: 'AWS::AutoScaling::AutoScalingGroup'
  Properties:
    LaunchTemplate: ⑨
      LaunchTemplateId: !Ref WorkerLaunchTemplate
      Version: !GetAtt 'WorkerLaunchTemplate.LatestVersionNumber'
    MinSize: 1 ⑩
    MaxSize: 2
    DesiredCapacity: 1
    HealthCheckGracePeriod: 120
    HealthCheckType: EC2 ⑪
    VPCZoneIdentifier: ⑫
      - !Ref SubnetA
      - !Ref SubnetB
    Tags: ⑬
      - PropagateAtLaunch: true
        Value: 'imager-worker'
        Key: Name
    DependsOn: VPCGatewayAttachment
    # [...]

```

- ① The launch template used as a blueprint for spinning up EC2 instances
- ② Attach an IAM role to the EC2 instances to allow the worker to access SQS, S3, and DynamoDB
- ③ Looks up the AMI with the Imagery worker pre-installed from the region map (see listing above)
- ④ Disables detailed monitoring of EC2 instances to avoid costs
- ⑤ Launch virtual machines of type `t2.micro` to run example under the free tier.
- ⑥ Configures a network interface (ENI) with a public IP address and the security group of the worker

- ⑦ Each virtual machine will execute this script at the end of the boot process. The script starts the Node.js worker.
- ⑧ Create an auto-scaling group which manages the virtual machines running the Imagery worker
- ⑨ References the launch template
- ⑩ The auto-scaling group will spin up at least one and no more than two EC2 instances
- ⑪ The auto-scaling group will replace failed EC2 instances
- ⑫ Spins up EC2 instances distributed among two subnets = AZs
- ⑬ Adds a Name tag to each instance which will show up at the Management Console, for example

After all that YAML reading, the CloudFormation stack should be created. Verify the status of your stack:

```
$ aws cloudformation describe-stacks --stack-name imagery
{
  "Stacks": [
    {
      ...
      "Description": "AWS in Action: chapter 16",
      "Outputs": [
        {
          "Description": "Load Balancer URL",
          "OutputKey": "EndpointURL",
          "OutputValue": "http://....us-east-1.elb.amazonaws.com" ❶
        },
        {
          "StackName": "imagery",
          "StackStatus": "CREATE_COMPLETE" ❷
        }
      ]
    }
}
```

- ❶ Copy this output into your web browser.
- ❷ Wait until CREATE_COMPLETE is reached.

The EndpointURL output of the stack contains the URL to access the Imagery application. When you open Imagery in your web browser, you can upload an image as shown in figure 16.16.

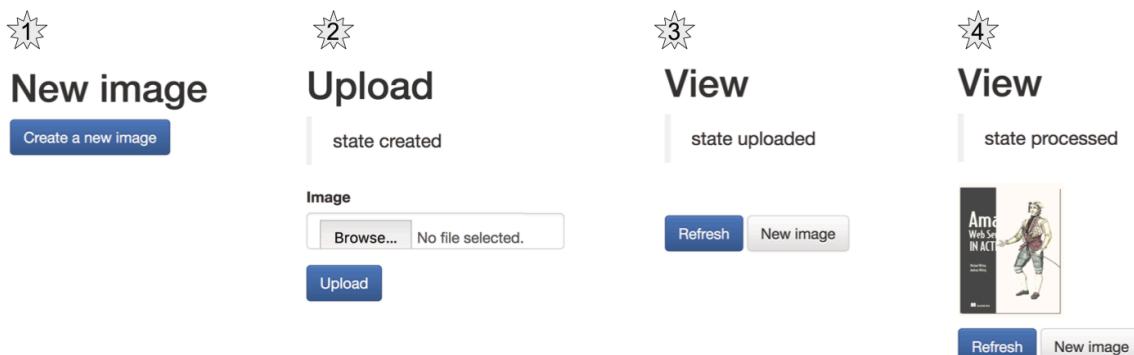


Figure 16.16 The Imagery application in action

Go ahead and upload some images and enjoy watching the images being processed.

SIDE BAR

Cleaning up



To get the name of the S3 bucket used by Imagery, run the following command in your terminal.

```
$ aws cloudformation describe-stack-resource --stack-name imagery \
  --logical-resource-id Bucket \
  --query "StackResourceDetail.PhysicalResourceId" \
  --output text
imagery-000000000000
```

Delete all the files in your S3 bucket `imagery-000000000000`. Don't forget to replace `$bucketname` with the output from the previous command.

```
$ aws s3 rm --recursive s3://$bucketname
```

Execute the following command to delete the CloudFormation stack:

```
$ aws cloudformation delete-stack --stack-name imagery
```

Stack deletion will take some time.

Congratulations, you have accomplished a big milestone: building a fault tolerant application on AWS. You are only one step away from the end game, which is scaling your application dynamically based on load.

16.4 Summary

- Fault tolerance means expecting that failures happen, and designing your systems in such a way that they can deal with failure.
- To create a fault-tolerant application, you can use idempotent actions to transfer from one state to the next.
- State shouldn't reside on the EC2 instance (a stateless server) as a prerequisite for fault-tolerance.
- AWS offers fault-tolerant services and gives you all the tools you need to create fault-tolerant systems. EC2 is one of the few services that isn't fault-tolerant out of the box.
- You can use multiple EC2 instances to eliminate the single point of failure. Redundant EC2 instances in different availability zones, started with an auto-scaling group, are the way to make EC2 fault-tolerant.

17

Scaling up and down: auto-scaling and CloudWatch

This chapter covers

- Creating an auto-scaling group with launch template
- Using auto-scaling to change the number of virtual machines
- Scaling a synchronous decoupled app behind a load balancer (ALB)
- Scaling an asynchronous decoupled app using a queue (SQS)

Suppose you're organizing a party to celebrate your birthday. How much food and drink do you need to buy? Calculating the right numbers for your shopping list is difficult:

- How many people will attend? You received several confirmations, but some guests will cancel at short notice or show up without letting you know in advance. So the number of guests is vague.
- How much will your guests eat and drink? Will it be a hot day, with everybody drinking a lot? Will your guests be hungry? You need to guess the demand for food and drink based on experiences from previous parties as well as weather, time of day, and other variables.

Solving the equation is a challenge because there are many unknowns. Behaving as a good host, you'll order more food and drink than needed so no guest will be hungry or thirsty for long. It may cost you more money than necessary, and you may end up wasting some of it, but this possible waste is the risk you must take to ensure you have enough for unexpected guests and circumstances.

Planning to meet future demands for your IT infrastructure is nearly impossible. To prevent a supply gap, you need to add extra capacity on top of the planned demand to prevent running short of resources.

Before the cloud, the same was true for our industry when planning the capacity of our IT

infrastructure. When procuring hardware for a data center, we always had to buy hardware based on the demands of the future. There were many uncertainties when making these decisions:

- How many users needed to be served by the infrastructure?
- How much storage would the users need?
- How much computing power would be required to handle their requests?

To avoid supply gaps, we had to order more or faster hardware than needed, causing unnecessary expenses.

On AWS, you can use services on demand. Planning capacity is less and less important. You can scale from one EC2 instance to thousands of EC2 instances. Storage can grow from gigabytes to petabytes. You can scale on demand, thus replacing capacity planning. The ability to scale on demand is called *elasticity* by AWS.

Public cloud providers like AWS can offer the needed capacity with a short waiting time. AWS serves more than a million customers, and at that scale it isn't a problem to provide you with 100 additional virtual machines within minutes if you need them suddenly. This allows you to address another problem: typical traffic patterns, as shown in figure 17.1. Think about the load on your infrastructure during the day versus at night, on a weekday versus the weekend, or before Christmas versus the rest of year. Wouldn't it be nice if you could add capacity when traffic grows and remove capacity when traffic shrinks? That's what this chapter is all about.

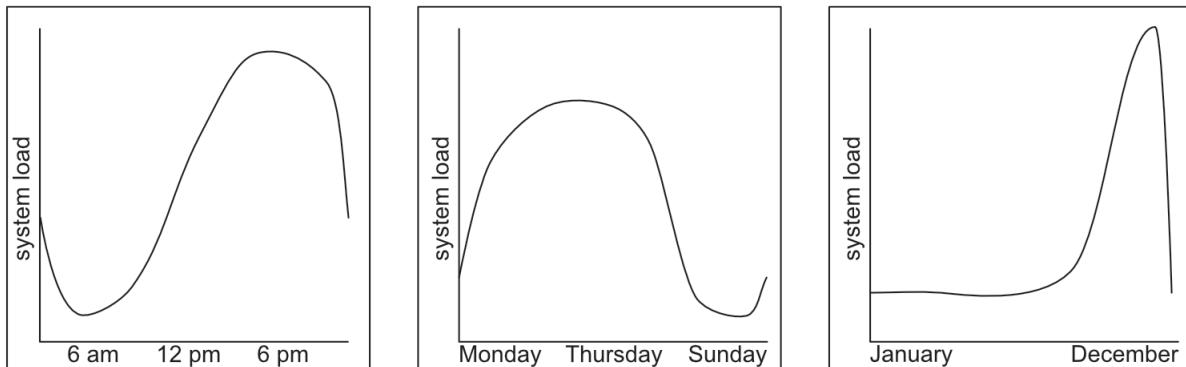


Figure 17.1 Typical traffic patterns for a web shop

Scaling the number of virtual machines is possible with *auto-scaling groups (ASG)* and *scaling policies* on AWS. Auto-scaling is part of the EC2 service and helps you to scale the number of EC2 instances you need to fulfill the current load of your system. We introduced auto-scaling groups in chapter 14 to ensure that a single virtual machine was running even if an outage of an entire data center occurred.

In this chapter, you'll learn how to manage a fleet of EC2 instances and how to adapt the size of the fleet depending on the current utilization of the infrastructure. To do so, you will use the concepts that you have learned about in chapters 14 and 15 already and enhance it with

automatic scaling.

- Using auto-scaling groups to launch multiple virtual machines of the same kind as you have been doing in chapters 14 and 15 already.
- Changing the number of virtual machines based on CPU load with the help of CloudWatch alarms, which is a new concept we are introducing in this chapter.
- Changing the number of virtual machines based on a schedule, to adapt to recurring traffic patterns, something you will learn about in this chapter.
- Using a load balancer as an entry point to the dynamic EC2 instance pool as you have been doing in chapter 15 already.
- Using a queue to decouple the jobs from the dynamic EC2 instance pool in a similar way than you learned about in chapter 15 already.

SIDE BAR
Examples are 100% covered by the Free Tier

The examples in this chapter are totally covered by the Free Tier. As long as you don't run the examples longer than a few days, you won't pay anything for it. Keep in mind that this applies only if you created a fresh AWS account for this book and there is nothing else going on in your AWS account. Try to complete the chapter within a few days, because you'll clean up your account at the end of the chapter.

There are two prerequisites for being able to scale your application horizontally, which means increasing and decreasing the number of virtual machines based on the current workload:

- The EC2 instances you want to scale need to be *stateless*. You can achieve stateless servers by storing data with the help of service like RDS (SQL database), DynamoDB (NoSQL database), EFS (network filesystem), or S3 (object store) instead of storing data on disks (instance store or EBS) that are only available to a single EC2 instance.
- An entry point to the dynamic EC2 instance pool is needed to be able to distribute the workload across multiple EC2 instances. EC2 instances can be decoupled synchronously with a load balancer, or asynchronously with a queue.

We introduced the concept of the stateless server in part 3 of this book and explained how to use decoupling in chapter 15. In this chapter you'll return to the concept of the stateless server and also work through an example of synchronous and asynchronous decoupling.

17.1 Managing a dynamic EC2 instance pool

Imagine that you need to provide a scalable infrastructure to run a web application, such as a blogging platform. You need to launch uniform virtual machines when the number of requests grows, and terminate virtual machines when the number of requests shrinks. To adapt to the current workload in an automated way, you need to be able to launch and terminate VMs automatically. Therefore, the configuration and deployment of the web application needs to be done during bootstrapping, without human interaction.

In this section, you will create an auto-scaling group. Next, you will learn how to change the number of EC2 instances launched by the auto-scaling group based on scheduled actions. Afterwards, you will learn how to scale based on a utilization metric provided by CloudWatch.

AWS offers a service to manage such a dynamic EC2 instance pool, called *auto-scaling groups*. Auto-scaling groups help you to:

- Dynamically adjust the number of virtual machines that are running
- Launch, configure, and deploy uniform virtual machines

The auto-scaling group grows and shrinks within the bounds you define. Defining a minimum of two virtual machines allows you to make sure at least two virtual machines are running in different availability zones to plan for failure. Conversely, defining a maximum number of virtual machines ensures you are not spending more money than you intended for your infrastructure.

As figure [17.2](#) shows, auto-scaling consists of three parts:

1. A launch template that defines the size, image, and configuration of virtual machines.
2. An auto-scaling group that specifies how many virtual machines need to be running based on the launch template.
3. Scaling plans that adjust the desired number of EC2 instances in the auto-scaling group based on a plan or dynamically.

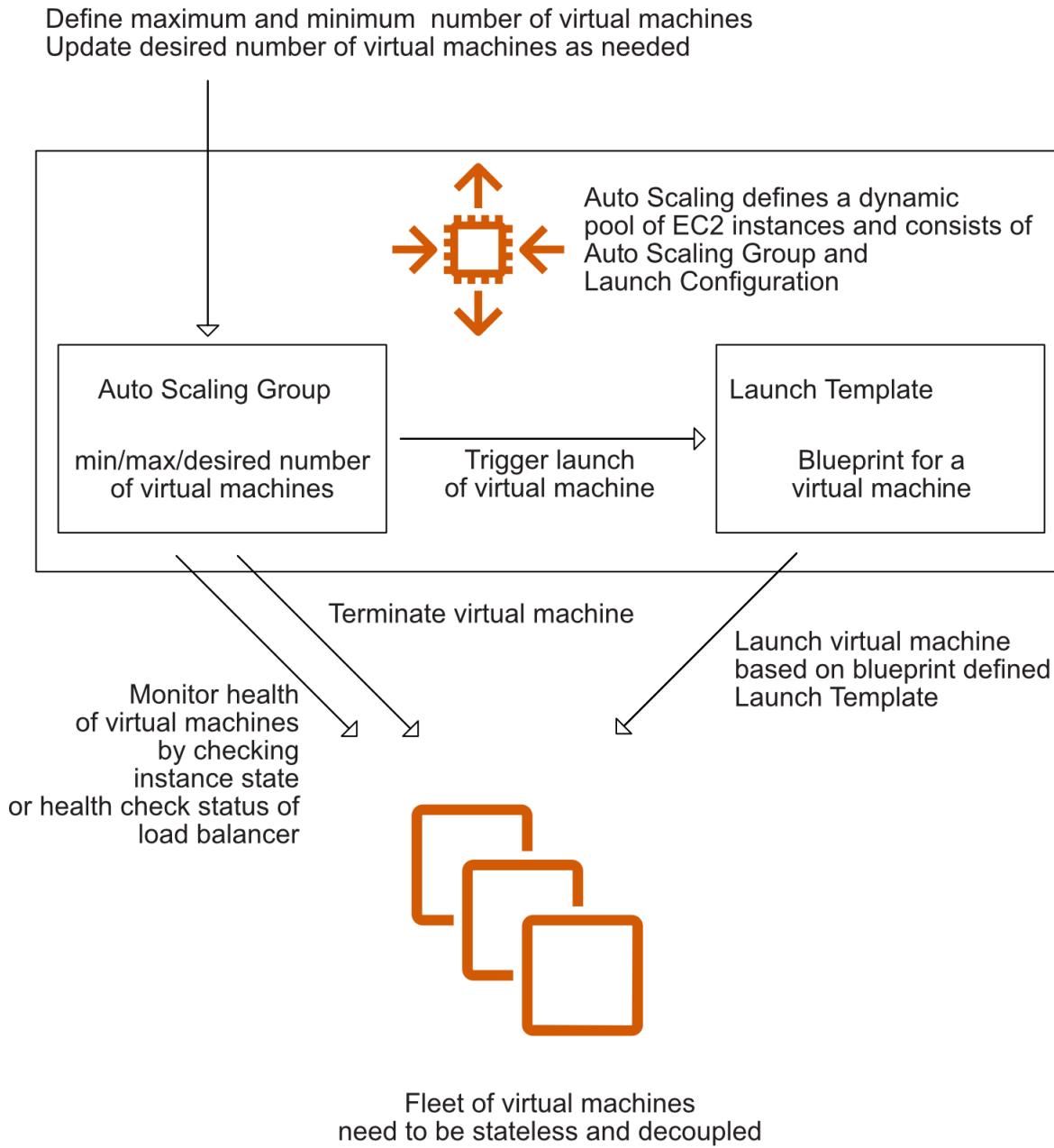


Figure 17.2 Auto-scaling consists of an ASG and a launch template, launching and terminating uniform virtual machines

If you want multiple EC2 instances to handle a workload, it's important to start identical virtual machines to build a homogeneous foundation. Use a launch template to define and configure new virtual machines. Table 17.1 shows the most important parameters for a launch template.

Table 17.1 Launch template parameters

Name	Description	Possible values
ImageId	Image from which to start a virtual machine	ID of Amazon Machine Image (AMI)
InstanceType	Size for new virtual machines	Instance type (such as t2.micro)
UserData	User data for the virtual machine used to execute a script during bootstrapping	BASE64-encoded String
NetworkInterfaces	Configures the network interfaces of the virtual machine. Most importantly, this parameter allows you to attach a public IP address to the instance.	List of network interface configurations.
IamInstanceProfile	Attaches an IAM instance profile linked to an IAM role	Name or Amazon Resource Name (ARN, an ID) of an IAM instance profile

After you create a launch template, you can create an auto-scaling group that references it. The auto-scaling group defines the maximum, minimum, and desired number of virtual machines. *Desired* means this number of EC2 instances should be running. If the current number of EC2 instances is below the desired number, the auto-scaling group will add EC2 instances. If the current number of EC2 instances is above the desired number, EC2 instances will be terminated. The desired capacity can be changed automatically based on load, or a schedule or manually. *Minimum* and *maximum* are the lower and upper limit for the number of virtual machines within the auto-scaling group.

The auto-scaling group also monitors whether EC2 instances are healthy and replaces broken instances. Table 17.2 shows the most important parameters for an auto-scaling group.

Table 17.2 Auto-scaling group (ASG) parameters

Name	Description	Possible values
DesiredCapacity	Desired number of healthy virtual machines	Integer
MaxSize	Maximum number of virtual machines, the upper scaling limit	Integer
MinSize	Minimum number of virtual machines, the lower scaling limit	Integer
HealthCheckType	How the auto-scaling group checks the health of virtual machines	EC2 (health of the instance) or ELB (health check of instance performed by a load balancer)
HealthCheckGracePeriod	Period for which the health check is paused after the launch of a new instance, to wait until the instance is fully bootstrapped	Number of seconds
LaunchTemplate	ID (<code>LaunchTemplateId</code>) and version of launch template used as a blueprint when spinning up virtual machines.	ID and version of launch template
TargetGroupARNs	The target groups of a load balancer, where auto-scaling registers new instances automatically	List of target group ARNs
VPCZoneIdentifier	List of subnets in which to launch EC2 instances in	List of subnet identifiers of a VPC

If you specify multiple subnets with the help of `VPCZoneIdentifier` for the auto-scaling group, EC2 instances will be evenly distributed among these subnets and thus among availability zones.

SIDE BAR**Don't forget to define health check grace period**

If you are using the ELB's health check for your auto-scaling group, make sure you specify a `HealthCheckGracePeriod` as well. Specify a health check grace period based on the time it takes from launching an EC2 instance until your application is running and passes the ELB's health check. For a simple web application, a health check period of 5 minutes is suitable.

Listing 17.1 shows how to set up such a dynamic EC2 instance pool with the help of a CloudFormation template.

Listing 17.1 Auto-scaling group and launch template for a web app

```
# [...]
LaunchTemplate:
  Type: 'AWS::EC2::LaunchTemplate'
  Properties:
    LaunchTemplateData:
      IamInstanceProfile:
        Name: !Ref InstanceProfile
      ImageId: 'ami-028f2b5ee08012131' ①
      InstanceType: 't2.micro' ②
      NetworkInterfaces:
        - AssociatePublicIpAddress: true ③
          DeviceIndex: 0
          Groups: ④
            - !Ref WebServerSecurityGroup
      UserData:
        'Fn::Base64': !Sub |
          #!/bin/bash -x
          yum -y install httpd
AutoScalingGroup:
  Type: 'AWS::AutoScaling::AutoScalingGroup'
  Properties:
    TargetGroupARNs: ⑥
    - !Ref LoadBalancerTargetGroup
    LaunchTemplate: ⑦
      LaunchTemplateId: !Ref LaunchTemplate
      Version: !GetAtt 'LaunchTemplate.LatestVersionNumber'
    MinSize: 2 ⑧
    MaxSize: 4 ⑨
    HealthCheckGracePeriod: 300 ⑩
    HealthCheckType: ELB ⑪
    VPCZoneIdentifier: ⑫
      - !Ref SubnetA
      - !Ref SubnetB
# [...]
```

- ① Image (AMI) from which to launch new virtual machines
- ② Instance type for new EC2 instances
- ③ Associates a public IP address with new virtual machines
- ④ Attach these security groups when launching new virtual machines.
- ⑤ Script executed during the bootstrap of virtual machines
- ⑥ Registers new virtual machines at the target group of the load balancer
- ⑦ References the launch template
- ⑧ Minimum number of EC2 instances
- ⑨ Maximum number of EC2 instances
- ⑩ Waits 300 seconds before terminating a new virtual machine because of a unsuccessful health check
- ⑪ Uses the health check from the ELB to check the health of the EC2 instances
- ⑫ Starts the virtual machines in these two subnets of the VPC

In summary, auto-scaling groups are a useful tool if you need to start multiple virtual machines

of the same kind across multiple availability zones. Additionally, an auto-scaling group replaces failed EC2 instances automatically.

17.2 Using metrics or schedules to trigger scaling

So far in this chapter, you've learned how to use an auto-scaling group and a launch template to manage virtual machines. With that in mind, you can change the desired capacity of the auto-scaling group manually, and new instances will be started or old instances will be terminated to reach the new desired capacity.

To provide a scalable infrastructure for a blogging platform, you need to increase and decrease the number of virtual machines in the pool automatically by adjusting the desired capacity of the auto-scaling group with scaling policies.

Many people surf the web during their lunch break, so you might need to add virtual machines every day between 11:00 a.m. and 1:00 p.m. You also need to adapt to unpredictable load patterns—for example, if articles hosted on your blogging platform are shared frequently through social networks.

Figure 17.3 illustrates two different ways to change the number of virtual machines:

- Defining a schedule. The timing would increase or decrease the number of virtual machines according to recurring load patterns (such as decreasing the number of virtual machines at night).
- Using a CloudWatch alarm. The alarm will trigger a scaling policy to increase or decrease the number of virtual machines based on a metric (such as CPU usage or number of requests on the load balancer).

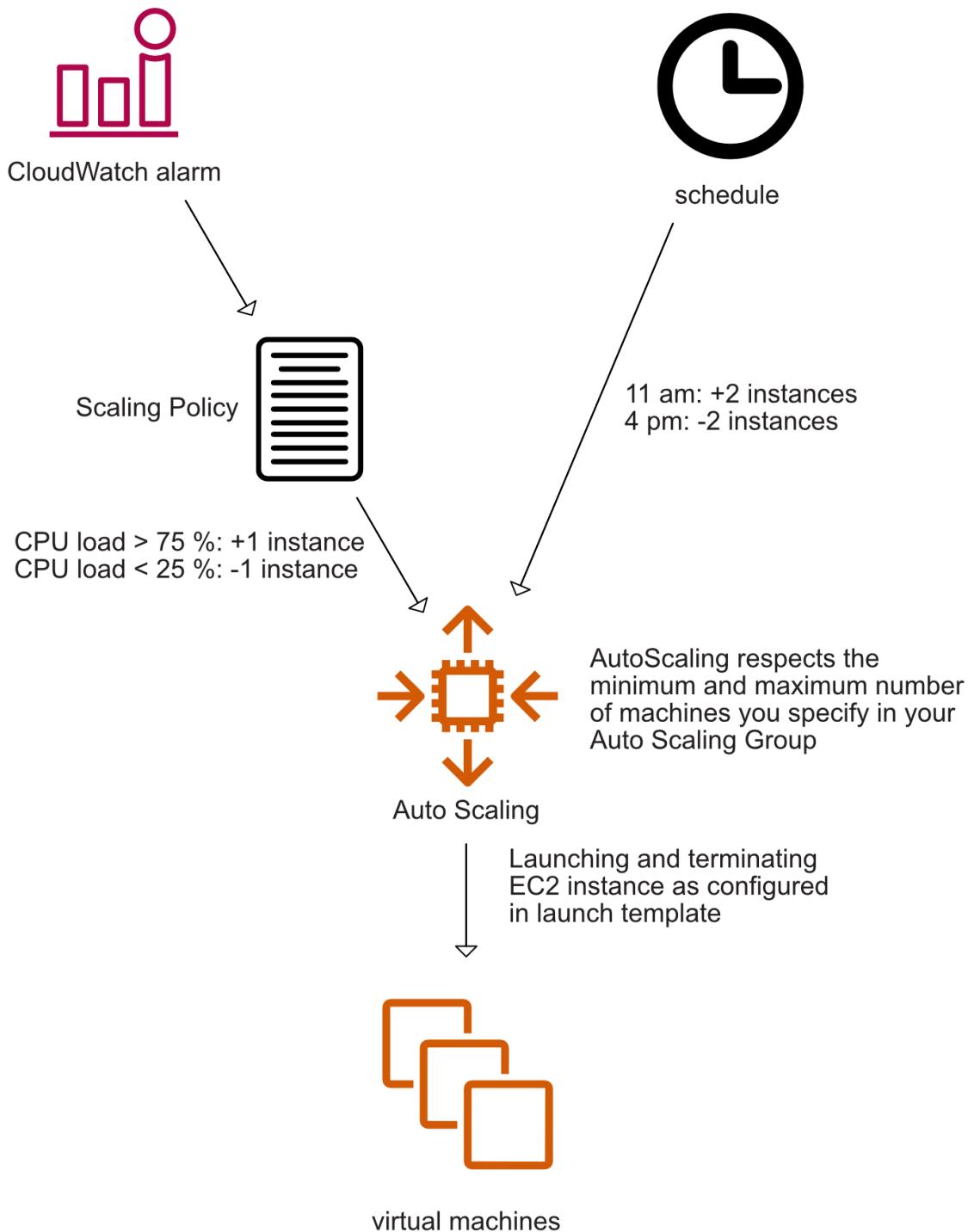


Figure 17.3 Triggering auto-scaling based on CloudWatch alarms or schedules

Scaling based on a schedule is less complex than scaling based on a CloudWatch metric, because it's difficult to find a metric to scale on reliably. On the other hand, scaling based on a schedule is less precise, as you have to over-provision your infrastructure to be able to handle unpredicted spikes in load.

17.2.1 Scaling based on a schedule

When operating a blogging platform, you might notice the following load patterns:

- *One-time actions*—Requests to your registration page increase heavily after you run a TV advertisement in the evening.
- *Recurring actions*—Many people seem to read articles during their lunch break, between 11:00 a.m. and 1:00 p.m.

Luckily, scheduled actions adjust your capacity with one-time or recurring actions. You can use different types of actions to react to both load pattern types.

Listing 17.2 shows a one-time scheduled action increasing the number of web servers at 12:00 UTC on Jan. 1, 2018. As usual, you'll find the code in the book's code repository on GitHub: <https://github.com/AWSinAction/code3>. The CloudFormation template for the WordPress example is located in `/chapter17/wordpress-schedule.yaml`.

Listing 17.2 Scheduling a one-time scaling action

```
OneTimeScheduledActionUp:
  Type: 'AWS::AutoScaling::ScheduledAction'          ①
  Properties:
    AutoScalingGroupName: !Ref AutoScalingGroup      ②
    DesiredCapacity: 4                                ③
    StartTime: '2025-01-01T12:00:00Z'                 ④
```

- ① Defining a scheduled action
- ② Name of the auto-scaling group
- ③ Set desired capacity to 4.
- ④ Change setting at 12:00 UTC on Jan. 1, 2025.

You can also schedule recurring scaling actions using cron syntax. The code example 17.3 shows how to use two scheduled actions to increase the desired capacity during business hours (08:00 to 20:00 UTC) every day.

Listing 17.3 Scheduling a recurring scaling action that runs at 20:00 UTC every day

```
RecurringScheduledActionUp:
  Type: 'AWS::AutoScaling::ScheduledAction'          ①
  Properties:
    AutoScalingGroupName: !Ref AutoScalingGroup      ②
    DesiredCapacity: 4                                ③
    Recurrence: '0 8 * * *'
RecurringScheduledActionDown:
  Type: 'AWS::AutoScaling::ScheduledAction'
  Properties:
    AutoScalingGroupName: !Ref AutoScalingGroup
    DesiredCapacity: 2                                ④
    Recurrence: '0 20 * * *'                         ⑤
```

- ① Defining a scheduled action
- ② Set desired capacity to 4.
- ③ Increase capacity at 08:00 UTC every day.
- ④ Set desired capacity to 2.
- ⑤ Decrease capacity at 20:00 UTC every day.

Recurrence is defined in Unix cron syntax format as shown here:

```
* * * * *
| | | |
| | | +-- day of week (0 - 6) (0 Sunday)
| | +--- month (1 - 12)
| +----- day of month (1 - 31)
| +----- hour (0 - 23)
+----- min (0 - 59)
```

We recommend using scheduled scaling actions whenever your infrastructure's capacity requirements are predictable—for example, an internal system used during work hours only, or a marketing action planned for a certain time.

17.2.2 Scaling based on CloudWatch metrics

Predicting the future is hard. Traffic will increase or decrease beyond known patterns from time to time. For example, if an article published on your blogging platform is heavily shared through social media, you need to be able to react to unplanned load changes and scale the number of EC2 instances.

You can adapt the number of EC2 instances to handle the current workload using CloudWatch alarms and scaling policies. CloudWatch helps monitor virtual machines and other services on AWS. Typically, services publish usage metrics to CloudWatch, helping you to evaluate the available capacity.

There are three types of scaling policies:

1. *Step scaling* allows more advanced scaling, as multiple scaling adjustments are supported, depending on how much the threshold you set has been exceeded.
2. *Target tracking* frees you from defining scaling steps and thresholds. You need only to define a target (such as CPU utilization of 70%) and the number of EC2 instances is adjusted accordingly.
3. *Predictive scaling* uses machine learning to predict load. It works best for cyclical traffic and recurring workload patterns (see "Predictive scaling for Amazon EC2 Auto Scaling" at <https://docs.aws.amazon.com/autoscaling/ec2/userguide/ec2-auto-scaling-predictive-scaling.html> to learn more).
4. *Simple scaling* is a legacy option which was replaced with *Step Scaling*.

All types of scaling policies use metrics and alarms to scale the number of EC2 instances based

on the current workload. As shown in figure 17.4, the virtual machines publish metrics to CloudWatch constantly. A CloudWatch alarm monitors one of these metrics and triggers a scaling action if the defined threshold is reached. The scaling policy then increases or decreases the desired capacity of the auto-scaling group.

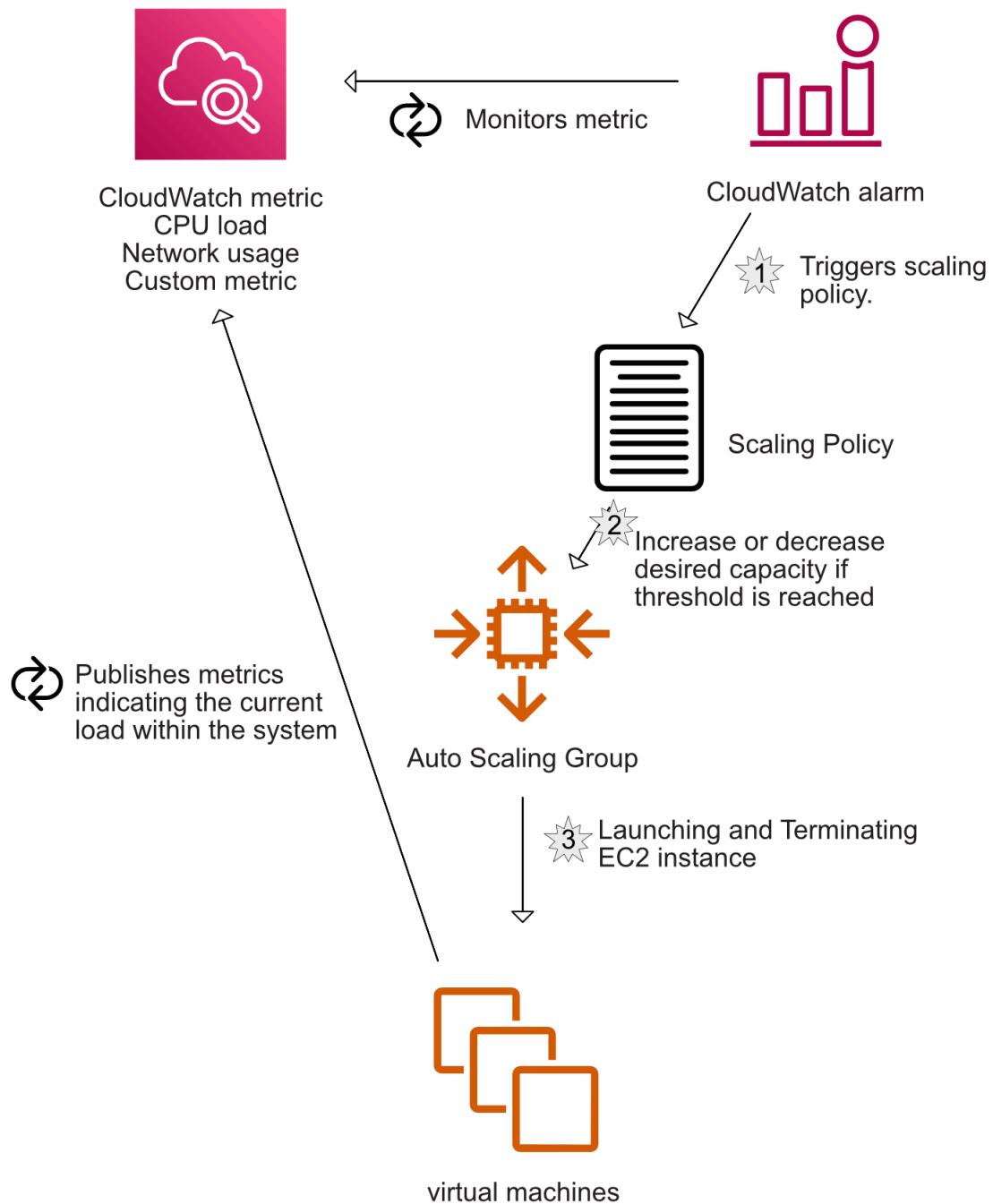


Figure 17.4 Triggering auto-scaling based on a CloudWatch metric and alarm

An EC2 instance publishes several metrics to CloudWatch by default: CPU, network, and disk utilization are the most important. Unfortunately, there is currently no metric for a virtual

machine's memory usage. You can use these metrics to scale the number of VMs if a bottleneck is reached. For example, you can add EC2 instances if the CPU is working to capacity.

The following parameters describe a CloudWatch metric:

- **Namespace**—Defines the source of the metric (such as AWS/EC2)
- **Dimensions**—Defines the scope of the metric (such as all virtual machines belonging to an auto-scaling group)
- **MetricName**—Unique name of the metric (such as `CPUUtilization`)

CloudWatch alarms are based on CloudWatch metrics. Table 17.3 explains the alarm parameters in detail.

Table 17.3 Parameters for a CloudWatch alarm that triggers scaling based on CPU utilization of all virtual machines belonging to an ASG

Context	Name	Description	Possible values
Condition	Statistic	Statistical function applied to a metric	Average, Sum, Minimum, Maximum, SampleCount
Condition	Period	Defines a time-based slice of values from a metric	Seconds (multiple of 60)
Condition	EvaluationPeriods	Number of periods to evaluate when checking for an alarm	Integer
Condition	Threshold	Threshold for an alarm	Number
Condition	ComparisonOperator	Operator to compare the threshold against the result from a statistical function	GreaterThanOrEqualToThreshold, GreaterThanThreshold, LessThanThreshold, LessThanOrEqualToThreshold
Metric	Namespace	Source of the metric	AWS/EC2 for metrics from the EC2 service
Metric	Dimensions	Scope of the metric	Depends on the metric; references the ASG for an aggregated metric over all associated EC2 instances
Metric	MetricName	Name of the metric	For example, <code>CPUUtilization</code>
Action	AlarmActions	Actions to trigger if the threshold is reached	Reference to the scaling policy

You can define alarms on many different metrics. You'll find an overview of all namespaces, dimensions, and metrics that AWS offers at <http://mng.bz/8E0X>. For example, you could scale based on the load balancer's metric counting the number of requests per target, or the networking throughput of your EC2 instances. You can also publish custom metrics—for example, metrics directly from your application like thread pool usage, processing times, or user sessions.

SIDE BAR**Scaling based on CPU load with VMs that offer burstable performance**

Some virtual machines, such as instance family t₂ and t₃, offer burstable performance. These virtual machines offer a baseline CPU performance and can burst performance for a short time based on credits. If all credits are spent, the instance operates at the baseline. For a t_{2.micro} instance, baseline performance is 10% of the performance of the underlying physical CPU.

Using virtual machines with burstable performance can help you react to load spikes. You save credits in times of low load, and spend credits to burst performance in times of high load. But scaling the number of virtual machines with burstable performance based on CPU load is tricky, because your scaling strategy must take into account whether your instances have enough credits to burst performance. Consider searching for another metric to scale (such as number of sessions) or using an instance type without burstable performance.

You've now learned how to use auto-scaling to adapt the number of virtual machines to the workload. It's time to bring this into action.

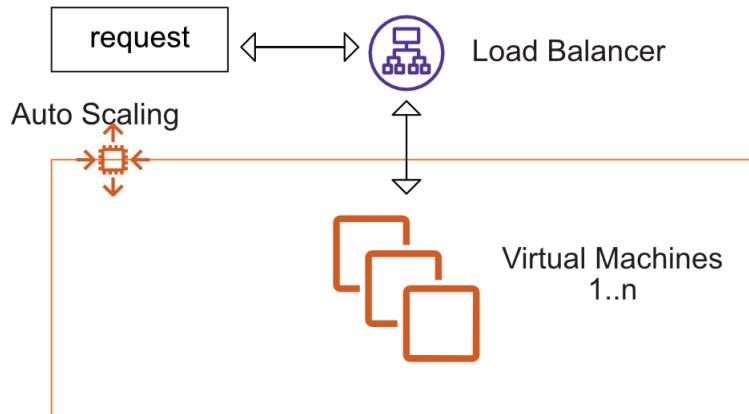
17.3 Decouple your dynamic EC2 instance pool

If you need to scale the number of virtual machines running your blogging platform based on demand, auto-scaling groups can help you provide the right number of uniform virtual machines, and scaling schedules or CloudWatch alarms can increase or decrease the desired number of EC2 instances automatically. But how can users reach the EC2 instances in the pool to browse the articles you're hosting? Where should the HTTP request be routed?

Chapter 15 introduced the concept of decoupling: synchronous decoupling with ELB, and asynchronous decoupling with SQS. If you want to use auto-scaling to grow and shrink the number of virtual machines, you need to decouple your EC2 instances from the clients, because the interface that's reachable from outside the system needs to stay the same no matter how many EC2 instances are working behind the scenes.

Figure 17.5 shows how to build a scalable system based on synchronous or asynchronous decoupling. A load balancer is acting as the entry point for synchronous decoupling, by distributing requests among a fleet of virtual machines. A message queue is used as the entry point for asynchronous requests. Messages from producers are stored in the queue. The virtual machines then poll the queue and process the messages asynchronously.

Synchronous decoupling



Asynchronous decoupling

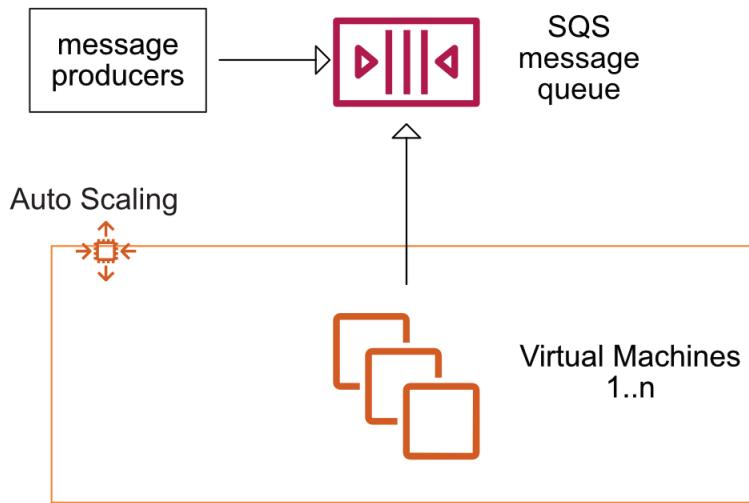


Figure 17.5 Decoupling allows you to scale the number of virtual machines dynamically.

Decoupled and scalable applications require stateless servers. A stateless server stores any shared data remotely in a database or storage system. The following two examples implement the concept of a stateless server:

- *WordPress blog*—Decoupled with ELB, scaled with auto-scaling and CloudWatch based on CPU utilization, and data outsourced to a MySQL database (RDS) and a network filesystem (EFS).
- *URL2PNG taking screenshots of URLs*—Decoupled with a queue (SQS), scaled with auto-scaling and CloudWatch based on queue length, and data outsourced to a NoSQL database (DynamoDB) and an object store (S3).

17.3.1 Scaling a dynamic EC2 instance pool synchronously decoupled by a load balancer

Answering HTTP(S) requests is a synchronous task. If a user wants to use your web application, the web server has to answer the corresponding requests immediately. When using a dynamic EC2 instance pool to run a web application, it's common to use a load balancer to decouple the EC2 instances from user requests. The load balancer forwards HTTP(S) requests to multiple EC2 instances, acting as a single entry point to the dynamic EC2 instance pool.

Suppose your company has a corporate blog for publishing announcements and interacting with the community. You're responsible for hosting the blog. The marketing department complains about slow page speed and even timeouts in the evening, when traffic reaches its daily peak. You want to use the elasticity of AWS by scaling the number of EC2 instances based on the current workload.

Your company uses the popular blogging platform WordPress for its corporate blog. Chapters 2 and 11 introduced a WordPress setup based on EC2 instances and RDS (MySQL database). In this final chapter of the book, we'd like to complete the example by adding the ability to scale.

Figure 17.6 shows the final, extended WordPress example. The following services are used for this highly available scaling architecture:

- EC2 instances running Apache to serve WordPress, a PHP application
- RDS offering a MySQL database that's highly available through Multi-AZ deployment
- EFS storing PHP, HTML, and CSS files as well as user uploads such as images and videos
- ELB to synchronously decouple the web servers from visitors
- Auto-scaling and CloudWatch to scale the number of EC2 instances based on the current CPU load of all running virtual machines

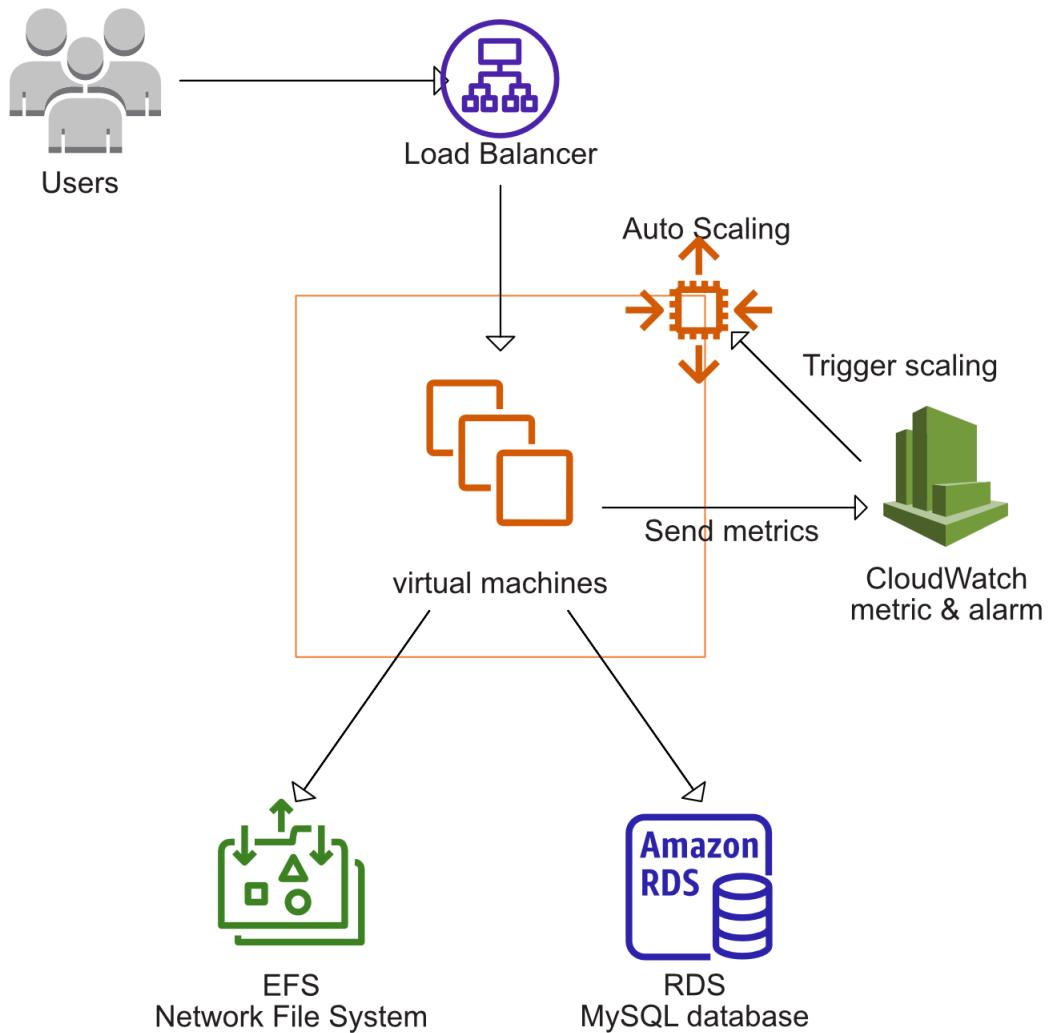


Figure 17.6 Auto-scaling web servers running WordPress, storing data on RDS and EFS, decoupled with a load balancer scaling based on load

As usual, you'll find the code in the book's code repository on GitHub: <https://github.com/AWSinAction/code3>. The CloudFormation template for the WordPress example is located in `/chapter17/wordpress.yaml`.

Execute the following command to create a CloudFormation stack that spins up the scalable WordPress setup. Replace `$Password` with your own password consisting of 8 to 30 letters and digits.

```
$ aws cloudformation create-stack --stack-name wordpress \
  --template-url https://s3.amazonaws.com/\
  awsinaction-code3/chapter17/wordpress.yaml --parameters \
  "ParameterKey=WordpressAdminPassword,ParameterValue=$Password" \
  --capabilities CAPABILITY_IAM
```

It will take up to 15 minutes for the stack to be created. This is a perfect time to grab some coffee or tea. Log in to the AWS Management Console, and navigate to the AWS CloudFormation service to monitor the process of the CloudFormation stack named `wordpress`. You have time to look through the most important parts of the CloudFormation template, shown in the following two listings [17.4](#), [17.5](#) and [17.6](#).

Create a blueprint to launch EC2 instances, also known as launch template as illustrated in listing [17.4](#) first.

Listing 17.4 Creating a scalable, HA WordPress setup (part 1)

```
LaunchTemplate:
  Type: 'AWS::EC2::LaunchTemplate' ①
  Metadata: # [...]
  Properties:
    LaunchTemplateData:
      BlockDeviceMappings: ③
      - DeviceName: '/dev/xvda'
        Ebs:
          Encrypted: true ④
          VolumeType: gp3 ⑤
      IamInstanceProfile: ⑥
        Name: !Ref InstanceProfile
      ImageId: !FindInMap [RegionMap, !Ref 'AWS::Region', AMI] ⑦
      Monitoring: ⑧
        Enabled: false
      InstanceType: 't2.micro' ⑨
      MetadataOptions: ⑩
        HttpTokens: required
      NetworkInterfaces: ⑪
      - AssociatePublicIpAddress: true ⑫
        DeviceIndex: 0
        Groups:
        - !Ref WebServerSecurityGroup ⑬
      UserData: # [...]
  ⑭
```

- ① Creates a launch template for auto-scaling
- ② Contains the configuration for the EC2 instance applied during bootstrapping
- ③ Configures the EBS volume attached to each instance
- ④ Enables encryption-at-rest for the EBS volume
- ⑤ Chooses volume type `gp3` for the volume
- ⑥ Configures the IAM instance profile for the virtual machines. Allowing the machines to authenticate and authorize for AWS services
- ⑦ Select the image from a map with AMIs organized by region
- ⑧ Disables detailed monitoring of the EC2 instances to reduce costs. Enable for production workloads
- ⑨ Configures the instance type
- ⑩ Enables IMDSv2, a security related configuration
- ⑪ Defines the network interface for the virtual machine

- ⑫ Enables a public IP address for the virtual machine
- ⑬ A list of security groups that should be attached to the EC2 instance
- ⑭ The user data contains a script to install and configure WordPress automatically

Second, the auto scaling group shown in listing 17.5 launches EC2 instances based on the launch template.

Listing 17.5 Creating a scalable, HA WordPress setup (part 2)

```
AutoScalingGroup:
  Type: 'AWS::AutoScaling::AutoScalingGroup'    ①
  DependsOn: ②
    - EFSMountTargetA
    - EFSMountTargetB
  Properties:
    TargetGroupARNs: ③
      - !Ref LoadBalancerTargetGroup
    LaunchTemplate: ④
      LaunchTemplateId: !Ref LaunchTemplate
      Version: !GetAtt 'LaunchTemplate.LatestVersionNumber'
    MinSize: 2 ⑤
    MaxSize: 4 ⑥
    HealthCheckGracePeriod: 300 ⑦
    HealthCheckType: ELB ⑧
    VPCZoneIdentifier: ⑨
      - !Ref SubnetA
      - !Ref SubnetB
    Tags:
      - PropagateAtLaunch: true
        Value: wordpress
        Key: Name
```

- ① Create an auto-scaling group
- ② As the EC2 instances require access to the EFS file system, wait until CloudFormation created the mount targets
- ③ Register and deregister virtual machines at the target group of the load balancer
- ④ References the latest version of the launch template
- ⑤ Launches at least two machines and ensures that at least two virtual machines are running, one for each of the two AZs for high availability
- ⑥ Launches no more than four machines
- ⑦ Wait five minutes to allow the EC2 instance and web server to start before evaluating the health check
- ⑧ Uses the ELB health check to monitor the health of the virtual machines
- ⑨ Launches VMs into two different subnets in two different AZs for high availability
- ⑩ Adds a tag including a name for all VMs launched by the ASG

You will learn how to create CloudWatch alarms for scaling in the next example. For now, we are using a target tracking scaling policy that creates CloudWatch alarms automatically in the

background. A target tracking scaling policy works like the thermostat in your home: you define the target and the thermostat constantly adjusts the heating power to reach the target.

Predefined metric specifications for the use with target tracking are:

- `ASGAverageCPUUtilization` to scale based on the average CPU utilization among all instances within an auto-scaling group.
- `ALBRequestCountPerTarget` to scale based on the number of requests forwarded from the Application Load Balancer (ALB) to a target.
- `ASGAverageNetworkIn` and `ASGAverageNetworkOut` to scale based on the average number of bytes received or sent.

In some cases, scaling based on CPU utilization, request count per target, or network throughput does not work. For example, you might have another bottleneck you need to scale on, such as disk I/O. Any CloudWatch metric can be used for target tracking as well. There is only one requirement: adding or removing instances must affect the metric proportionally. For example, request latency is not a valid metric for target tracking, as adjusting the number of instances does not affect the request latency directly.

Listing 17.6 Creating a scalable, HA WordPress setup (part 3)

```
ScalingPolicy:
  Type: 'AWS::AutoScaling::ScalingPolicy'          ①
  Properties:
    AutoScalingGroupName: !Ref AutoScalingGroup      ②
    PolicyType: TargetTrackingScaling                ③
    TargetTrackingConfiguration:
      PredefinedMetricSpecification:                 ④
        PredefinedMetricType: ASGAverageCPUUtilization ⑤
      TargetValue: 70                                ⑥
    EstimatedInstanceWarmup: 60                      ⑦
  
```

- ① Creates a scaling policy
- ② Adjusts the desired capacity of the auto-scaling group.
- ③ Creates a scaling policy tracking a specified target
- ④ Configures the target tracking
- ⑤ Uses a predefined scaling metric
- ⑥ Average CPU utilization across all EC2 instances of the ASG
- ⑦ Defines the target at 70% CPU utilization
- ⑧ Exclude newly launched EC2 instances from CPU metric for 60 seconds to avoid scaling on load caused due to the bootstrapping of the VM and your application.

Follow these steps after the CloudFormation stack reaches the state `CREATE_COMPLETE` to create a new blog post containing an image:

1. Select the CloudFormation stack `wordpress`, and switch to the Outputs tab.

2. Open the link shown for key URL with a modern web browser.
3. Search for the Log In link in the navigation bar, and click it.
4. Log in with username admin and the password you specified when creating the stack with the CLI.
5. Click Posts in the menu on the left.
6. Click Add New.
7. Type in a title and text, and upload an image to your post.
8. Click Publish.
9. Go back to the blog by clicking on the View Post link.

Now you're ready to scale. We've prepared a load test that will send 500,000 requests to the WordPress setup within a few minutes. Don't worry about costs: the usage is covered by the Free Tier. After three minutes, new virtual machines will be launched to handle the load. The load test takes 10 minutes. Another 15 minutes later, the additional VMs will disappear. Watching this is fun; you shouldn't miss it.

NOTE

If you plan to do a big load test, consider the AWS Acceptable Use Policy at <https://aws.amazon.com/aup> and ask for permission before you begin (see also <https://aws.amazon.com/security/penetration-testing>).

SIDE BAR**Simple HTTP load test**

We're using a tool called Apache Bench to perform a load test of the WordPress setup. The tool is part of the `httpd-tools` package available from the Amazon Linux package repositories.

Apache Bench is a basic benchmarking tool. You can send a specified number of HTTP requests by using a specified number of threads. We're using the following command for the load test, to send 500,000 requests to the load balancer using 15 threads. The load test is limited to 600 seconds and we're using a connection timeout of 120 seconds. Replace `$UrlLoadBalancer` with the URL of the load balancer:

```
$ ab -n 500000 -c 15 -t 300 -s 120 -r $UrlLoadBalancer
```

Update the CloudFormation stack with the following command to start the load test:

```
$ aws cloudformation update-stack --stack-name wordpress \
  --template-url https://s3.amazonaws.com/ \
  --awsaction-code3/chapter17/wordpress-loadtest.yaml --parameters \
  --ParameterKey=WordpressAdminPassword,UsePreviousValue=true \
  --capabilities CAPABILITY_IAM
```

Watch for the following things to happen, using the AWS Management Console:

1. Open the CloudWatch service, and click Alarms on the left.
2. When the load test starts, the alarm called

TargetTracking-wordpress-AutoScalingGroup--**AlarmHigh**- will reach the ALARM state after about 10 minutes.

3. Open the EC2 service, and list all EC2 instances. Watch for two additional instances to launch. At the end, you'll see five instances total (four web servers and the EC2 instance running the load test).
4. Go back to the CloudWatch service, and wait until the alarm named TargetTracking-wordpress-AutoScalingGroup--**AlarmLow**- reaches the ALARM state.
5. Open the EC2 service, and list all EC2 instances. Watch for the two additional instances to disappear. At the end, you'll see three instances total (two web servers and the EC2 instance running the load test).

The entire process will take about 30 minutes.

You've watched auto-scaling in action: your WordPress setup can now adapt to the current workload. The problem with pages loading slowly or even timeouts in the evening is solved.

SIDE BAR Cleaning up



Execute the following commands to delete all resources corresponding to the WordPress setup:

```
$ aws cloudformation delete-stack --stack-name wordpress
```

17.3.2 Scaling a dynamic EC2 instances pool asynchronously decoupled by a queue

Imagine that you're developing a social bookmark service where users can save and share their links. Offering a preview that shows the website being linked to is an important feature. But the conversion from URL to PNG is causing high load during the evening, when most users add new bookmarks to your service. Because of that, customers are dissatisfied with your application's slow response times.

You will learn how to dynamically scale a fleet of EC2 instances to asynchronously generate screenshots of URLs in the following example. Doing so allows you to guarantee low response times at any time because the load-intensive workload is isolated into background jobs.

Decoupling a dynamic EC2 instance pool asynchronously offers an advantage if you want to scale based on workload: because requests don't need to be answered immediately, you can put requests into a queue and scale the number of EC2 instances based on the length of the queue. This gives you an accurate metric to scale, and no requests will be lost during a load peak because they're stored in a queue.

To handle the peak load in the evening, you want to use auto-scaling. To do so, you need to decouple the creation of a new bookmark and the process of generating a preview of the website. Chapter 12 introduced an application called URL2PNG that transforms a URL into a PNG image. Figure 17.7 shows the architecture, which consists of an SQS queue for asynchronous decoupling as well as S3 for storing generated images. Creating a bookmark will trigger the following process:

1. A message is sent to an SQS queue containing the URL and the unique ID of the new bookmark.
2. EC2 instances running a Node.js application poll the SQS queue.
3. The Node.js application loads the URL and creates a screenshot.
4. The screenshot is uploaded to an S3 bucket, and the object key is set to the unique ID.
5. Users can download the screenshot directly from S3 using the unique ID.

A CloudWatch alarm is used to monitor the length of the SQS queue. If the length of the queue reaches five, an additional virtual machine is started to handle the workload. When the queue length goes below five, another CloudWatch alarm decreases the desired capacity of the auto-scaling group.

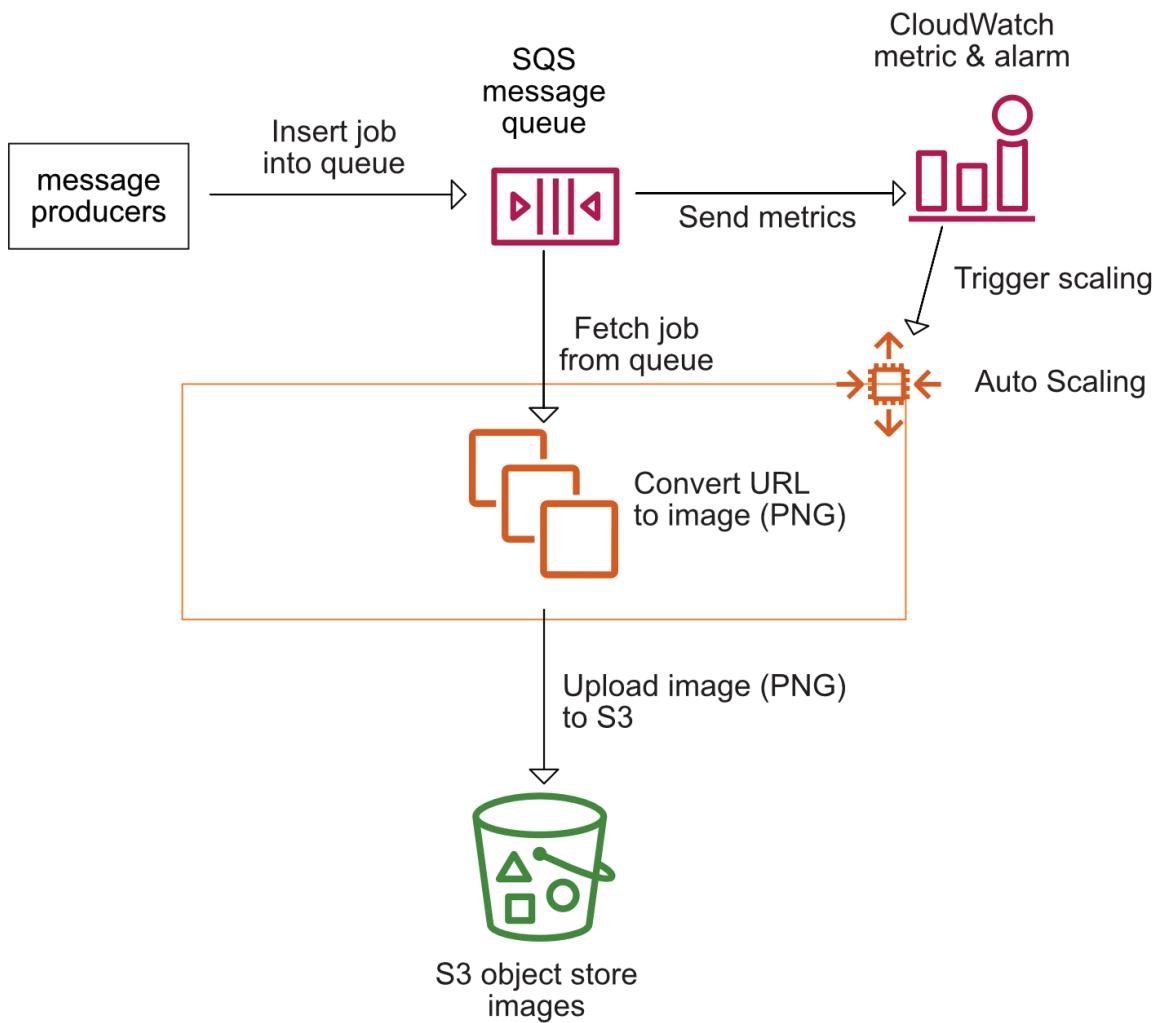


Figure 17.7 Auto-scaling virtual machines that convert URLs into images, decoupled by an SQS queue

The code is in the book's code repository on GitHub at <https://github.com/AWSinAction/code3>. The CloudFormation template for the URL2PNG example is located at chapter17/url2png.yaml.

Execute the following command to create a CloudFormation stack that spins up the URL2PNG application.

```
$ aws cloudformation create-stack --stack-name url2png \
  --template-url https://s3.amazonaws.com/\
  awsinaction-code3/chapter17/url2png.yaml \
  --capabilities CAPABILITY_IAM
```

It will take up to five minutes for the stack to be created. Log in to the AWS Management Console, and navigate to the AWS CloudFormation service to monitor the process of the CloudFormation stack named url2png.

We're using the length of the SQS queue to scale the number of EC2 instances. As the number of

messages in the queue does not correlate with the number of EC2 instances processing messages from the queue, it is not possible to use a target tracking policy. Therefore, you will use a step scaling policy in this scenario as illustrated in listing 17.7.

Listing 17.7 Monitoring the length of the SQS queue

```
# [...]
HighQueueAlarm:
  Type: 'AWS::CloudWatch::Alarm'
  Properties:
    EvaluationPeriods: 1          ①
    Statistic: Sum               ②
    Threshold: 5                ③
    AlarmDescription: 'Alarm if queue length is higher than 5.'
    Period: 300                 ④
    AlarmActions:
      - !Ref ScalingUpPolicy    ⑤
    Namespace: 'AWS/SQS'         ⑥
    Dimensions:                  ⑦
      - Name: QueueName
        Value: !Sub '${SQSQueue.QueueName}'
    ComparisonOperator: GreaterThanThreshold   ⑧
    MetricName: ApproximateNumberOfMessagesVisible ⑨
# [...]
```

- ① Number of time periods to evaluate when checking for an alarm
- ② Sums up all values in a period
- ③ Alarms if the threshold of 5 is reached
- ④ Uses a period of 300 seconds because SQS metrics are published every 5 minutes
- ⑤ Increases the number of desired instances by 1 through the scaling policy
- ⑥ The metric is published by the SQS service.
- ⑦ The queue, referenced by name, is used as the dimension of the metric.
- ⑧ Alarms if the sum of the values within the period is greater than the threshold of 5
- ⑨ The metric contains an approximate number of messages pending in the queue.

The CloudWatch alarm triggers a scaling policy. The scaling policy shown in 17.8 defines how to scale. To keep things simple, we are using a step scaling policy with only a single step. Add additional steps if you want to react to a threshold breach in a more fine-grained way.

Listing 17.8 A step scaling policy which adds one more instance to an ASG

```
# [...]
ScalingUpPolicy:
  Type: 'AWS::AutoScaling::ScalingPolicy'          ①
  Properties:
    AdjustmentType: 'ChangeInCapacity'            ②
    AutoScalingGroupName: !Ref AutoScalingGroup     ③
    PolicyType: 'StepScaling'                      ④
    MetricAggregationType: 'Average'              ⑤
    EstimatedInstanceWarmup: 60                    ⑥
    StepAdjustments:
      - MetricIntervalLowerBound: 0                ⑦
        ScalingAdjustment: 1                      ⑧
# [...]
```

- ① Creates a scaling policy
- ② The scaling policy increases the capacity by an absolute number.
- ③ Attaches the scaling policy to the auto-scaling group
- ④ Creates a scaling policy of type step scaling
- ⑤ The aggregation type used when evaluating the steps, based on the metric defined within the CloudWatch alarm that triggers the scaling policy
- ⑥ The metrics of a newly launched instance are ignored for 60 seconds while it boots up.
- ⑦ Defines the scaling steps. We use a single step in this example.
- ⑧ The scaling step is valid from the alarms threshold to infinity.
- ⑨ Increase the desired capacity of the ASG by 1.

To scale down the number of instances when the queue is empty, a CloudWatch alarm and scaling policy with the opposite values needs to be defined.

You're ready to scale. We've prepared a load test that will quickly generate 250 messages for the URL2PNG application. A virtual machine will be launched to process jobs from the SQS queue. After a few minutes, when the load test is finished, the additional virtual machine will disappear.

Update the CloudFormation stack with the following command to start the load test:

```
$ aws cloudformation update-stack --stack-name url2png \
  --template-url https://s3.amazonaws.com/\
  awsinaction-code3/chapter17/url2png-loadtest.yaml \
  --capabilities CAPABILITY_IAM
```

Watch for the following things to happen, with the help of the AWS Management Console:

1. Open the CloudWatch service, and click Alarms at left.
2. When the load test starts, the alarm called `url2png-HighQueueAlarm-*` will reach the ALARM state after a few minutes.

3. Open the EC2 service, and list all EC2 instances. Watch for an additional instance to launch. At the end, you'll see three instances total (two workers and the EC2 instance running the load test).
4. Go back to the CloudWatch service, and wait until the alarm named `url2png-LowQueueAlarm-*` reaches the `ALARM` state.
5. Open the EC2 service, and list all EC2 instances. Watch for the additional instance to disappear. At the end, you'll see two instances total (one worker and the EC2 instance running the load test).

The entire process will take about 20 minutes.

You've watched auto-scaling in action. The URL2PNG application can now adapt to the current workload, and the problem with slowly generated screenshots has been solved.

SIDE BAR Cleaning up



Execute the following commands to delete all resources corresponding to the `url2png` example.

```
$ URL2PNG_BUCKET=$(aws cloudformation describe-stacks --stack-name url2png \
  --query "Stacks[0].Outputs[?OutputKey=='BucketName'].OutputValue" \
  --output text

$ aws s3 rm s3://${URL2PNG_BUCKET} --recursive

$ aws cloudformation delete-stack --stack-name url2png
```

Whenever distributing an application among multiple EC2 instances, you should use an auto-scaling group. Doing so allows you to spin up identical instances with ease. You get the most out of the possibilities of the cloud when scaling the number of instances based on a schedule or a metric depending on the load pattern.

17.4 Summary

- You can use auto-scaling to launch multiple identical virtual machines by using a launch template and an auto-scaling group.
- EC2, SQS, and other services publish metrics to CloudWatch (CPU utilization, queue length, and so on).
- CloudWatch alarms can change the desired capacity of an auto-scaling group. This allows you to increase the number of virtual machines based on CPU utilization or other metrics.
- Virtual machines need to be stateless if you want to scale them according to your current workload.
- To distribute load among multiple virtual machines, synchronous decoupling with the help of a load balancer or asynchronous decoupling with a message queue is necessary.

That's it! You have mastered the end game: scaling your infrastructure dynamically. Kudos, you have learned about and experienced the most important aspects of Amazon Web Services. We wish you all the best for moving your first production workload to the cloud.

18

Building modern architectures for the cloud: ECS and Fargate

This chapter covers

- Deploying a web server with App Runner, the simplest way to run containers on AWS.
- Comparing Elastic Container Service (ECS) and Elastic Kubernetes Service (EKS).
- An introduction into ECS: cluster, task definition, task, and service.
- Running containers with Fargate without the need of managing virtual machines.
- Building a modern architecture based on ALB, ECS, Fargate, and S3.

When working with our consulting clients, we handle two types of projects:

- Brownfield projects where the goal is to migrate workloads from on-premises to the cloud. Sooner or later these clients also ask for ways to modernize their legacy systems.
- Greenfield projects where the goal is to develop a solution from scratch with the latest technology available in the cloud.

Both types of projects are interesting and challenging. This chapter introduces a modern architecture, which you could use to modernize a legacy system as well as when building something from scratch. In recent years, there has hardly been a technology that has spread as rapidly as containers. In our experience, containers fit well for both brownfield and greenfield projects. You will learn how to make use of containers, to deploy your workloads on AWS in this chapter.

We want to focus on the cutting edge aspects of deploying containers on AWS. Therefore, we skip the details on how to create container images or start a container with Docker. We recommend Docker in Action (<https://www.manning.com/books/docker-in-action-second-edition> in case you want to learn about the fundamentals of Docker and containers.

SIDE BAR Examples not covered by Free Tier

The examples in this chapter are not covered by the Free Tier. Deploying the examples to AWS will cost you less than \$1 per day. You will find information on how to delete all resources at the end of each example or at the end of the chapter. Therefore, we recommend to complete the chapter within a few days.

SIDE BAR Chapter requirements

This chapter assumes, that you have a basic understanding of the following components:

- Running software in containers (*Docker in Action, 2nd Edition*, <https://livebook.manning.com/book/docker-in-action-second-edition/chapter-2/>)
- Storing data on Simple Storage Service (chapter 8)
- Distributing requests with Elastic Load Balancing (chapter 15)

On top of that, the example included in this chapter makes intensive use of the following:

- Automating cloud infrastructure with CloudFormation (chapter 5)

18.1 Why should you consider containers instead of virtual machines?

First of all, containers and virtual machines are similar concepts. This means you can apply your knowledge gained from previous chapters to the world of containers. As shown in figure 18.1 both approaches start with an image to spin up a virtual machine or container. Of course, there are differences between both technologies, but we will not discuss them here. As a mental model, it helps to think of containers as lightweight virtual machines.



Figure 18.1 From a high level view virtual machines and containers are similar concepts.

How often do you hear "... but it works on my machine" when talking to developers? It is not easy to create an environment providing the libraries, frameworks and runtime environments required by an application. Since 2013, Docker has made the concept of containers popular. As in logistics, a container in software development is a standardized unit that can be easily moved and delivered. In our experience, this simplifies the development process significantly. Especially, when aiming for continuous deployment, which means shipping every change to test or production systems automatically.

In theory, you spin up a container based on the same image on your local machine, an on-premises server, as well as in the cloud. Boundaries only exist between UNIX and Windows, as well as Intel/AMD and ARM processors. In contrast, it is much more complicated to launch an Amazon Machine Image (AMI) on your local machine.

Also, containers increase portability. In our opinion, it is much easier to move a containerized workload from on-premises to the cloud, or to another cloud provider. But beware of the marketing promises by many vendors, it is still a lot of work to integrate your system with the target infrastructure.

We have guided several organizations in the adoption of containers. In doing so, we have observed that containers promote an important competency: building and running immutable servers. An immutable server, is a server that you do not change once it is launched from an image. But what if you need to rollout a change? Create a new image and replace the old servers

with servers launched from the new image. In theory, you could do the same thing with EC2 instances as well, and we highly recommend you do so. But as you are typically not able to log into a running container to make changes, so following the immutable server approach is your only option. The keyword here is `Dockerfile`, a configuration file containing everything that is needed to build a container image.

18.2 Comparing different options to run containers on AWS

Hopefully we've convinced you that containers on AWS bring many benefits. Next, let's answer the question of how best to deploy containers on AWS.

To impress you, let's start with a simple option: AWS App Runner. Type the following command into your terminal to launch containers running a simple web server from a container image.

SIDE BAR

AWS CLI

AWS CLI not working on your machine? Go to chapter 4 to learn how to install and configure the command-line interface.

Listing 18.1 Creating an App Runner service

```
aws apprunner create-service \
  ↪ --service-name simple \
  ↪ --source-configuration '{"ImageRepository": \
  ↪ {"ImageIdentifier": "public.ecr.aws/s5r5alt5/simple:latest", \
  ↪ "ImageRepositoryType": "ECR_PUBLIC"}}' \
  ↪ ① ② ③ ④
```

- ① Creating an App Runner service which will spin up containers.
- ② Define a name for the service.
- ③ Configure the source of the container image.
- ④ Choose public or private container registry hosted by AWS.

It will take about 5 minutes until a simple web server is up and running. Use the following command to get the status and URL of your service. Open the URL in your browser. On a side note, App Runner even supports custom domains, in case that's a crucial feature to you.

Listing 18.2 Fetching information about App Runner services

```
$ aws apprunner list-services
{
    "ServiceSummaryList": [
        {
            "ServiceName": "simple",
            "ServiceId": "5e7ffd09c13d4d6189e99bb51fc0f230",
            "ServiceArn": "arn:aws:apprunner:us-east-1:111111111111:
                ↪service/simple/...", ❶
            "ServiceUrl": "bxjsdpnnaz.us-east-1.awssapprunner.com", ❷
            "CreatedAt": "2022-01-07T20:26:48+01:00",
            "UpdatedAt": "2022-01-07T20:26:48+01:00",
            "Status": "RUNNING" ❸
        }
    ]
}
```

- ❶ The ARN of the service, needed to delete the service later.
- ❷ Open this URL in your browser.
- ❸ Wait until the status reaches RUNNING.

App Runner is a Platform-as-a-Service (PaaS) offering for container workloads. You provide a container image bundling a web application and App Runner takes care of everything else as illustrated in figure [18.2](#).

- Runs and monitors containers.
- Distributes requests among running containers.
- Scales the number of containers based on load.

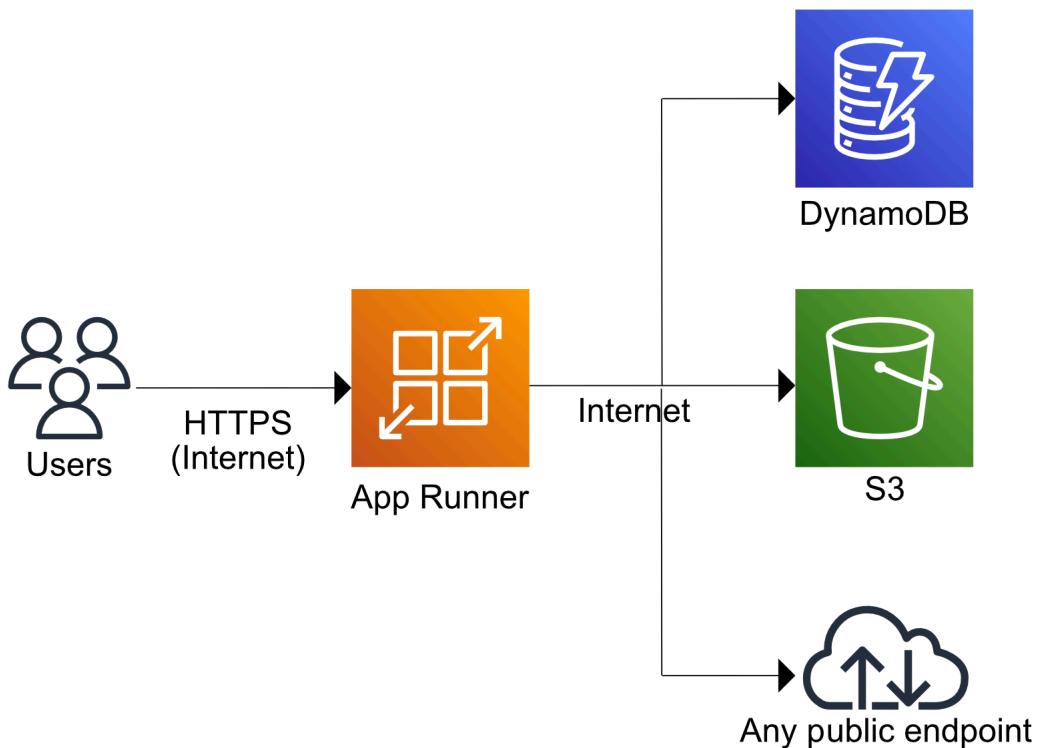


Figure 18.2 App Runner provides a simple way to host containerized web applications

The clue is that you are only paying for memory but not CPU resources during times a running container does not process any requests. Let's look at pricing with an example. Imagine a web application with minimal resource requirements only used from 9am to 5 pm, which is 8 hours per day. The minimal configuration on App Runner is 1 vCPU and 2 GB memory.

- Active hours (= hours in which requests are processed)
 - 1 vCPU: $\$0.064 * 8 * 30 = \15.36 per month
 - 2 GB memory: $2 * \$0.007 * 8 * 30 = \3.36 per month
- Inactive hours (= hours in which no requests are processed)
 - 2 GB memory: $2 * \$0.007 * 16 * 30 = \6.72 per month

In total, that's \$25,44 per month for the smallest configuration supported by App Runner. See "AWS App Runner Pricing" at <https://aws.amazon.com/apprunner/pricing/> for more details.

By the way, don't forget to delete your App Runner service to avoid unexpected costs. Replace `$ServiceArn` with the ARN you noted down after creating the service.

Listing 18.3 Fetching information about App Runner services

```
$ aws apprunner delete-service \
  --service-arn $ServiceArn
```

That was fun, wasn't it? But simplicity comes with limitations. Here are a few reasons, why App

Runner might not be a good fit to deploy your application.

1. **Unlimited network access** anyone can reach your service running on App Runner via Internet. There is no network firewall or WAF (web application firewall) option available. Also, your application has access to the whole Internet.
2. **Missing private network connectivity** your service cannot reach any resources running in your VPC. For example, you cannot connect to an RDS database instance running in your VPC. Only public services - S3 and DynamoDB for example - are accessible from App Runner.

It is also worth noting, that App Runner does not come with an SLA yet. Also, comparing costs between the different options is tricky, as different dimensions are used for billing. Roughly speaking, App Runner should be cheap for small workloads with few requests, but rather expensive for large workloads with many requests.

That's why we will introduce two other ways to deploy containers on AWS next. The two main services to manage containers on AWS are: Elastic Container Service (ECS) and Elastic Kubernetes Services (EKS).

NOTE

What is Kubernetes?

Kubernetes (K8s) is an open-source container orchestration system. Originally, Google designed Kubernetes, but nowadays, the Cloud Native Computing Foundation maintains the project. Kubernetes runs on your local machine, on-premises, and most cloud providers offer a fully-managed service.

The discussion about which of the two services is better is often very heated and reminiscent of the discussions about the editors vim and emacs. When viewed unemotionally, the functional scope of ECS and EKS is very similar. The both handle:

- Monitoring and replacing failed containers.
- Deploying new versions of your containers.
- Scaling the number of containers to adapt to load.

Of course, we would also like to highlight the differences, which we have summarized in the following table.

Table 18.1 Launch configuration parameters

Category	ECS	EKS
Portability	ECS is available on AWS. ECS Anywhere is an extension to utilize ECS for on-premises workloads. Other cloud providers do not support ECS.	EKS is available on AWS. For on-premises workloads, there is EKS Anywhere which is supported by AWS but requires VMware vSphere and the option to deploy and manage Kubernetes yourself. Also, most other cloud providers come with a Kubernetes offering.
License	Proprietary service but free of charge.	Open-source license (Apache License 2.0).
Ecosystem	Works very well together with many AWS services (e.g., ALB, IAM, VPC, ...).	Comes with a vibrant open-source ecosystem (e.g., Prometheus, Helm, ...). Integration with AWS services exist, but are not always mature.
Costs	A cluster is free of charge. Of course, you pay for the compute infrastructure.	AWS charges about \$72 per month for each cluster. Also, AWS recommends not to deploy workloads that require isolation to the same cluster. On top of that, you are paying for the compute infrastructure.

We observe that Kubernetes is very popular especially but not only among developers. Even though we are software developers ourselves, we prefer ECS for most workloads. The most important arguments for us are: monthly costs per cluster and integration with other AWS services. On top of that, CloudFormation comes with full support for ECS.

Next, you will learn about the basic concepts behind ECS.

18.3 The ECS basics: cluster, service, task, and task definition

When working with ECS, you need to create a cluster first. A cluster is a logical group for all the components we discuss next. It is fine to create multiple clusters to isolate workloads from each other. For example, we typically create a different clusters for test and production environments. The cluster itself is free of charge and by default you can create up to 10,000 clusters, which you probably do not need by the way.

To run a container on ECS you need to create a task definition. The task definition includes all the informations that are required to run a container. See figure [18.3](#) for more details.

1. The container image URL
2. Provisioned baseline and limit for CPU
3. Provisioned baseline and limit for memory
4. Environment variables
5. Network configuration

Please note, that a task definition might describe one or multiple containers.

Next, you are ready to create a task. To do so, you need to specify the cluster as well as the task definition. After you created the task, ECS will try to run containers as specified. It is important to note, that all containers defined in a task definition will run on the same host. That's important, in case you have multiple containers that need to share local resources, the local network for example.

Figure 18.3 shows how to run tasks based on a task definition.

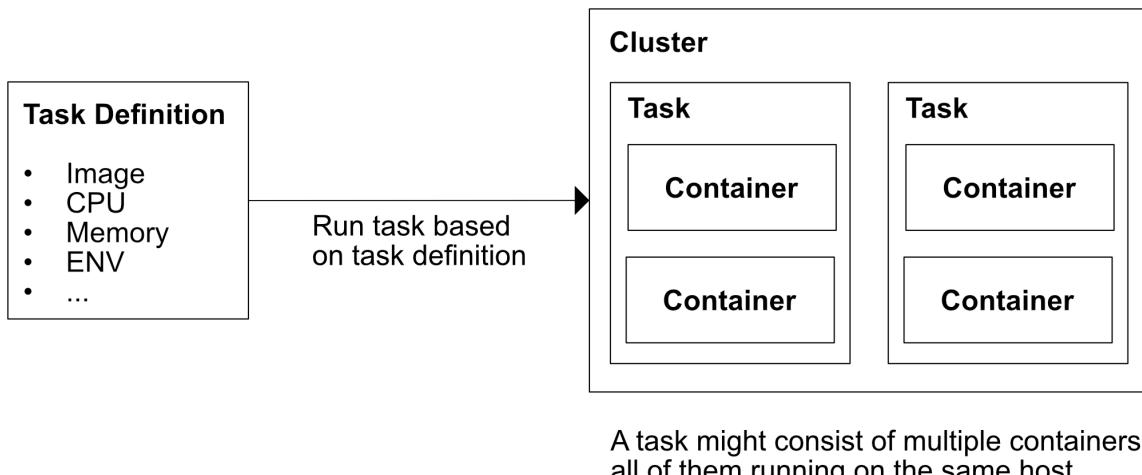


Figure 18.3 A task definition defines all the details needed to create a task which consists of one or multiple containers.

Luckily, you can but do not have to create tasks manually. Suppose you want to deploy a web server on ECS. In this case, you need to ensure that at least two containers of the same kind are running around the clock to spread the workload among two availability zones. In case of high load, even more containers should be started for a short time. You need an ECS service for that.

Think of an ECS service similar to an Auto Scaling Group. An ECS service, as shown in figure 18.4, performs the following tasks.

- Runs multiple tasks of the same kind.
- Scales the number of tasks based on load.
- Monitors and replaces failed tasks.
- Spreads tasks across availability zones.
- Orchestrates rolling updates.

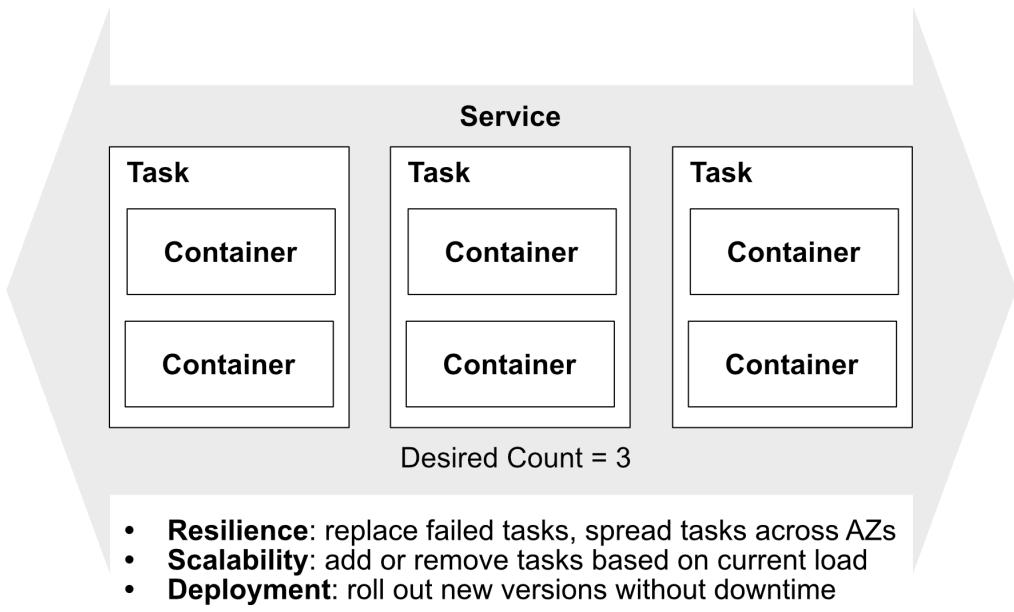


Figure 18.4 An ECS service manages multiple tasks of the same kind

Equipped with the knowledge of the most important components for ECS, we move on.

18.4 AWS Fargate: running containers without managing a cluster of virtual machines

Let's take a little trip down AWS history lane. ECS was generally available since 2015. At the beginning, ECS was adding another layer to our infrastructures. With ECS you not only had to manage, maintain and scale containers, but also the underlying EC2 instances. This was increasing complexity significantly.

In November 2017, AWS introduced an important service: AWS Fargate. As shown in figure 18.5, Fargate provides a fully-managed container infrastructure allowing you to spin up containers in a similar way to launching EC2 instances. This was a game changer! Since then, we deployed our workloads with ECS and Fargate whenever possible. And we advise you to do the same.

By the way, Fargate is not only available for ECS but for EKS as well. Also, Fargate offers Amazon Linux 2 and Microsoft Windows 2019 Server Full and Core editions as platform for your containers.

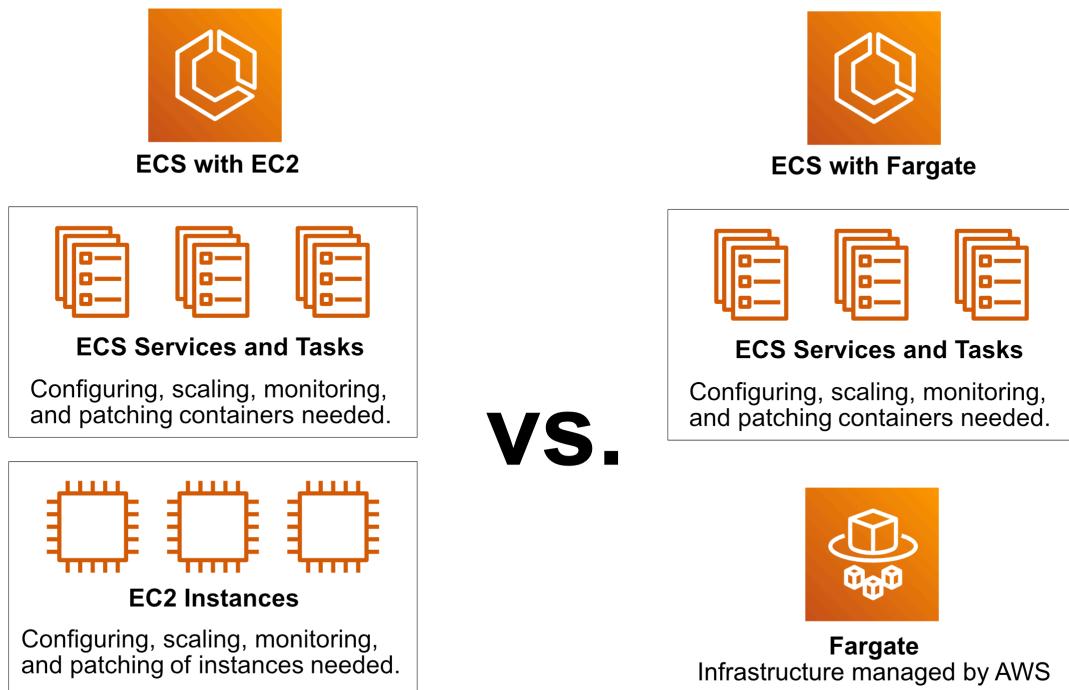


Figure 18.5 With Fargate you do not need to manage a fleet of EC2 instances anymore to deploy containers with ECS

With EC2 instances, you choose an instance type which specifies the available resources like CPU and memory. In contrast, Fargate requires you to configure the provisioned CPU and memory capacity per task. The following table shows the available options.

Table 18.2 Provisioning CPU and Memory for Fargate

CPU	Memory
0.25 vCPU	0.5 GB, 1 GB, or 2 GB
0.5 vCPU	Min 1 GB, Max 4 GB, 1 GB increments
1 vCPU	Min 2 GB, Max 8 GB, 1 GB increments
2 vCPU	Min 4 GB, Max 16 GB, 1 GB increments
4 vCPU	Min 8 GB, Max 30 GB, 1 GB increments

Fargate is billed for every second a task is running, from downloading the container image until the task terminates. What does a Fargate task with 1 vCPU and 4 GB memory cost per month? It depends on the region and architecture (Linux/X86, Linux/ARM, Windows/X86). Let's do the math for Linux/ARM in `us-east-1`.

- 1 vCPU: $\$0.04048 * 24 * 30 = \$29,15$ per month
- 4 GB memory: $4 * \$0.004445 * 24 * 30 = \$12,80$ per month

In total, that's \$41.95 per month for a Fargate task with 1 vCPU and 2 GB memory. See "AWS Fargate Pricing" at <https://aws.amazon.com/fargate/pricing/> for more details.

When comparing the costs for CPU and memory, it is noticeable that EC2 is cheaper compared to Fargate. For example, a `m6g.medium` instance with 1 vCPU and 4 GB memory costs \$27.72 per month. But when scaling EC2 instances for ECS yourself fragmentation and overprovisioning will add up as well. Besides that, the additional complexity will consume working time. In our opinion, Fargate is worth it in most scenarios.

It is important to mention, that Fargate comes with a few limitations. Most applications are not affected by those limitations, but you should double check before starting with Fargate. Here comes a list of the most important - but not all - limitations. See "Amazon ECS on AWS Fargate" at https://docs.aws.amazon.com/AmazonECS/latest/developerguide/AWS_Fargate.html for more details.

- A maximum of 4 vCPU and 30 GB memory.
- Container cannot run in privileged mode.
- Missing GPU support.
- Attaching EBS volumes is not supported.

Now it's finally time to see ECS in action.

18.5 Walking through a cloud-native architecture: ECS, Fargate, and S3

We take notes all the time. When we're on the phone with a customer, when we're thinking through a new chapter for a book, when we're looking at a new AWS service in detail. Do you do it similarly? Imagine you want to host your notes in the cloud. In this example, you will deploy **notea**, a privacy-first, open-source note-taking application to AWS. **notea** is a typical modern web application which uses React for the user interface and Next.js for the backend. All data is stored on S3.

The cloud-native architecture as shown in figure [18.6](#) consists of the following building blocks:

- The Application Load Balancer (ALB) distributes incoming requests among all running containers.
- An ECS service spins up containers and scales based on CPU load.
- Fargate provides the underlying compute capacity.
- The application stores all data on S3.

You may have noticed that the concepts from the previous chapters can be transferred to a modern architecture based on ECS easily.

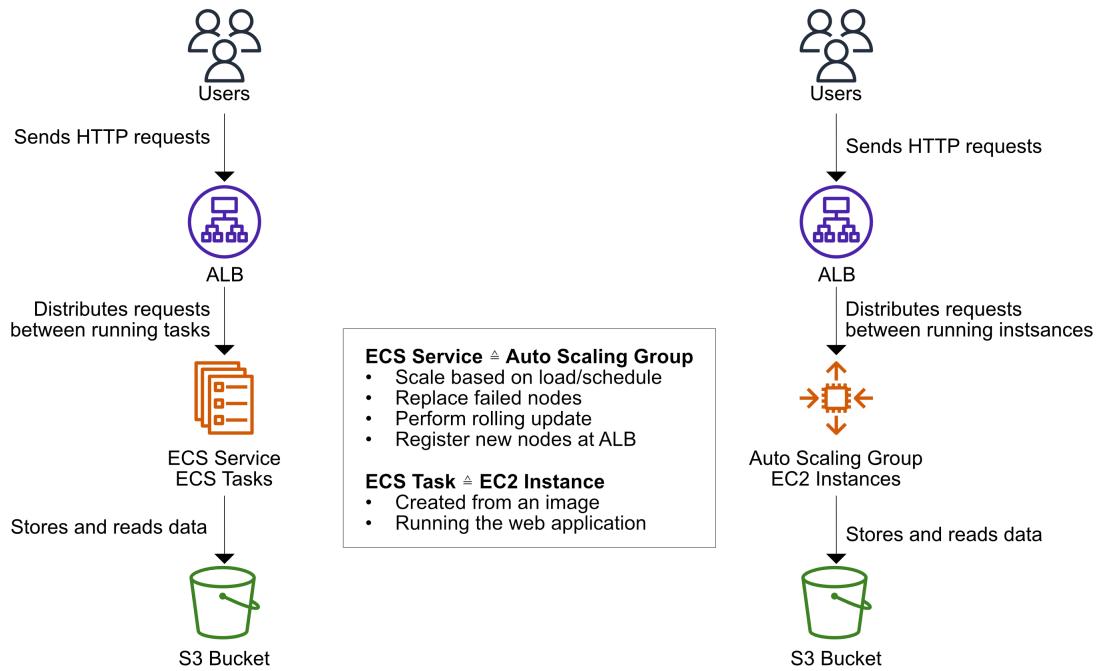


Figure 18.6 ECS is for containers, what an auto-scaling group is for EC2 instances

As usual, you'll find the code in the book's code repository on GitHub: <https://github.com/AWSinAction/code3>. The CloudFormation template for the notea example is located in `/chapter18/notea.yaml`.

Execute the following command to create a CloudFormation stack that spins up notea. Don't forget to replace `$ApplicationId` with a unique character sequence (e.g., your name abbreviation) and `$Password` with a password for protecting your notes. Please note, your password will be transmitted unencrypted over HTTP, so you should use a throwaway password that you are not using anywhere else.

```
$ aws cloudformation create-stack --stack-name notea \
  --template-url https://s3.amazonaws.com/\
  awsinaction-code3/chapter18/notea.yaml --parameters \
  "ParameterKey=ApplicationId,ParameterValue=$ApplicationId" \
  "ParameterKey=Password,ParameterValue=$Password" \
  --capabilities CAPABILITY_IAM
```

It will take about five minutes until your note-taking app is up and running. Use the following command to wait until the stack was created successfully and fetch the URL to open in your browser.

```
$ aws cloudformation wait stack-create-complete \
  --stack-name notea && aws cloudformation describe-stacks \
  --stack-name notea --query "Stacks[0].Outputs[0].OutputValue" \
  --output text
```

Congratulations, you launched a modern web application with ECS and Fargate. Happy note

taking!

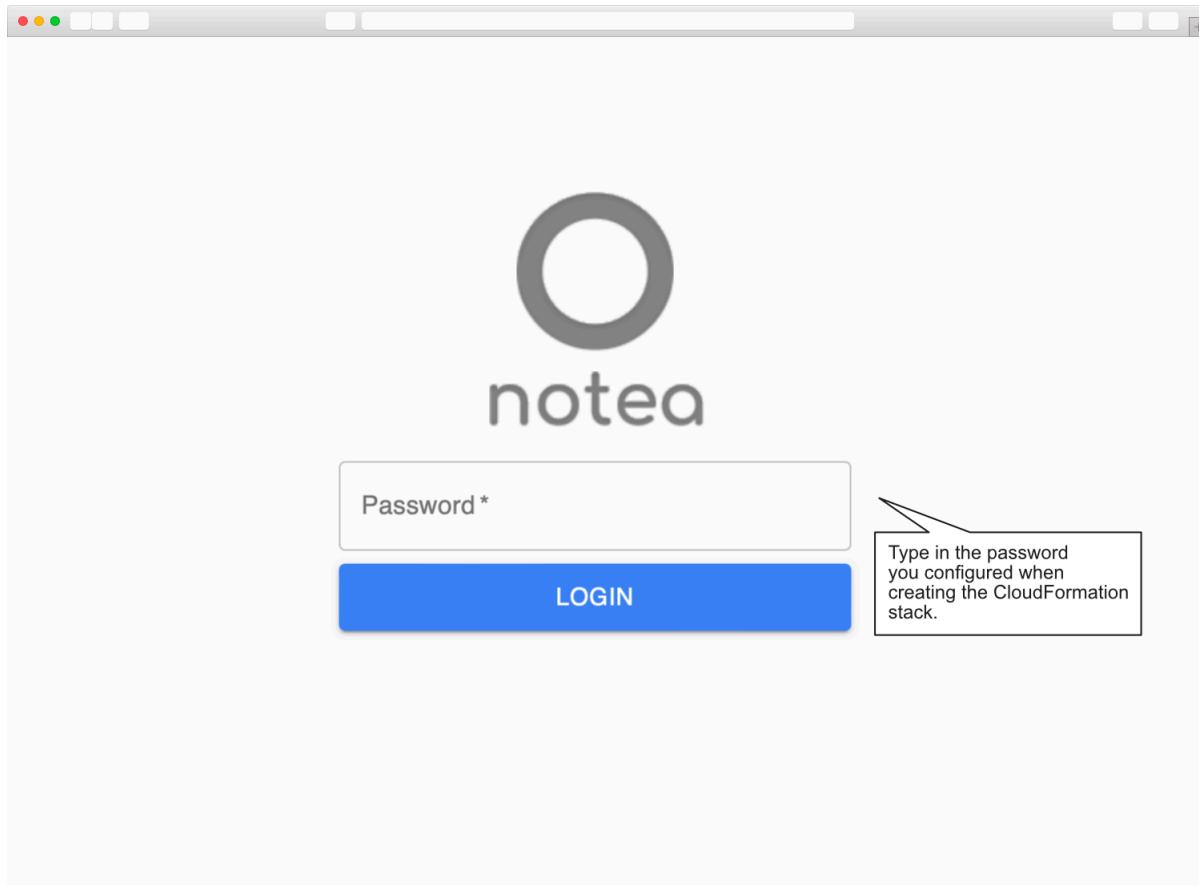


Figure 18.7 notea is up and running!

Next, we highly recommend to open the AWS Management Console and go to the ECS service to explore the cluster, the service, the tasks, and tasks definition. Use <https://console.aws.amazon.com/ecs/> to jump right into the ECS service.

What we like about ECS is that we can deploy all components with CloudFormation. Therefore, let's dive into the code. First, you need to create a task definition. For a better understanding, figure [18.8](#) shows the differnt configuration parts of the task definition.

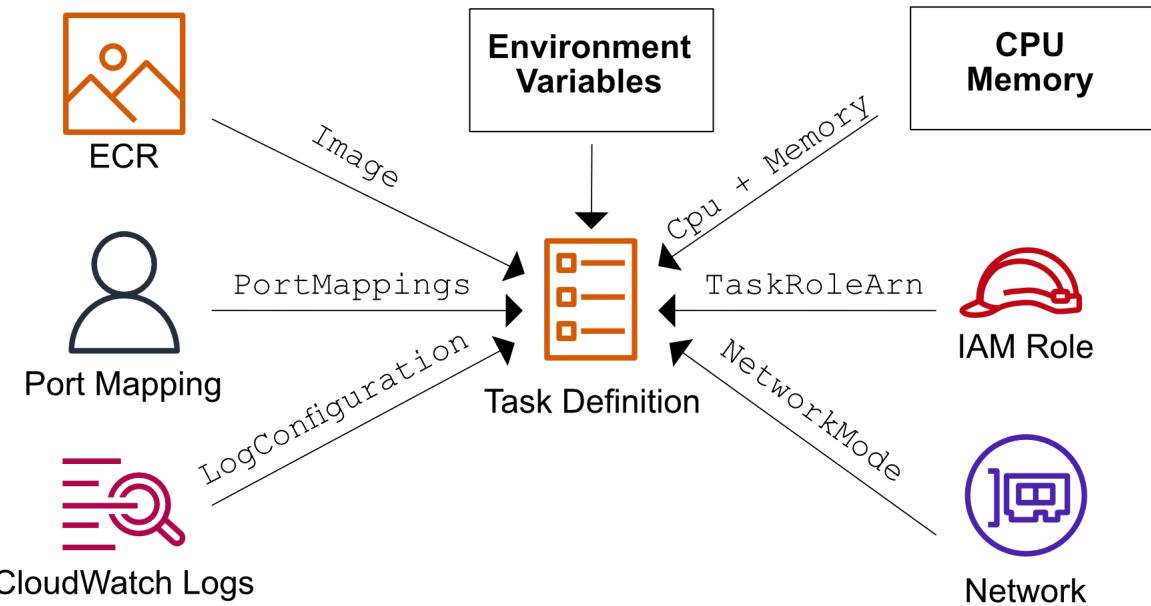


Figure 18.8 Configuring a task definition with CloudFormation

The code in listing 18.4 shows the details.

Listing 18.4 Configuring a task definition

```

TaskDefinition:
  Type: 'AWS::ECS::TaskDefinition'
  Properties:
    ContainerDefinitions: ①
      - Name: app ②
        Image: 'public.ecr.aws/s5r5alt5/notea:latest' ③
      PortMappings:
        - ContainerPort: 3000 ④
          Protocol: tcp
        Essential: true
      LogConfiguration: ⑤
        LogDriver: awslogs
      Options:
        'awslogs-region': !Ref 'AWS::Region'
        'awslogs-group': !Ref LogGroup
        'awslogs-stream-prefix': app
    Environment: ⑥
      - Name: 'PASSWORD'
        Value: !Ref Password
      - Name: 'STORE_REGION'
        Value: !Ref 'AWS::Region'
      - Name: 'STORE_BUCKET'
        Value: !Ref Bucket
      - Name: COOKIE_SECURE
        Value: 'false'
    Cpu: 512 ⑦
    ExecutionRoleArn: !GetAtt 'TaskExecutionRole.Arn' ⑧
    Family: !Ref 'AWS::StackName'
    Memory: 1024 ⑨
    NetworkMode: awsvpc ⑩
    RequiresCompatibilities: [FARGATE] ⑪
    TaskRoleArn: !GetAtt 'TaskRole.Arn' ⑫
  
```

- ① Remember that a task definition describes one or multiple containers? In this example, there is only one container called `app`.
- ② We will reference the container named `app` later.
- ③ The URL points to a publicly hosted container image bundling the `notea` app.
- ④ The container starts a server on port 3000.
- ⑤ The log configuration tells the container to ship logs to CloudWatch, which is the default for ECS and Fargate.
- ⑥ The `notea` container expects a few environment variables for configuration. Those environment variables are configured here.
- ⑦ This tells Fargate to provision 0.5 vCPUs for our task.
- ⑧ The IAM role is used by Fargate to fetch container images, ship logs, and similar tasks.
- ⑨ This tells Fargate to assign 1024 MB memory to our task.
- ⑩ Fargate only supports the networking mode `awsvpc` which will attach an Elastic Network Interface (ENI) to each task. You learned about the ENI in chapter 3 already.
- ⑪ Specifies that the task definition should be used with Fargate only.
- ⑫ The IAM role used by the application to access S3.

The task definition configures two IAM roles for the tasks. An IAM role is required to authenticate and authorize when accessing any AWS services. The IAM role defined by `ExecutionRoleArn` is not very interesting, the role grants Fargate access to basic services for downloading container images or publishing logs. However, the IAM role `TaskRoleArn` is very important as it grants the containers access to AWS services. In our example, `notea` requires read and write access to S3. And that's exactly what the IAM role in listing [18.5](#) is all about.

Listing 18.5 The task role grants the container access to list, read, write, and delete objects of an S3 bucket

```

TaskRole:
  Type: 'AWS::IAM::Role'
  Properties:
    AssumeRolePolicyDocument:
      Statement:
        - Effect: Allow
          Principal:
            Service: 'ecs-tasks.amazonaws.com'    ①
          Action: 'sts:AssumeRole'
    Policies:
      - PolicyName: S3AccessPolicy
        PolicyDocument:
          Statement:
            - Effect: Allow
              Action:
                - 's3:GetObject'    ②
                - 's3:PutObject'    ③
                - 's3:DeleteObject' ④
              Resource: !Sub '${Bucket.Arn}/*'    ⑤
            - Effect: Allow
              Action:
                - 's3>ListBucket'    ⑥
              Resource: !Sub '${Bucket.Arn}'

```

- ① Grants ECS tasks access to the role.
- ② Authorize role to read data from S3.
- ③ Authorize role to write data to S3.
- ④ Authorize role to delete data from S3.
- ⑤ Read and write access only granted to the S3 bucket created for notes only.
- ⑥ Allows listing all the objects in the bucket.

Next, you need to create an ECS service that launches tasks and with them containers. The most important configuration details shown in listing [18.6](#) are.

- `DesiredCount` defines the number of tasks the service will launch. The `DesiredCount` will be changed by auto-scaling later.
- `LoadBalancers` the service registers and deregisters tasks at the ALB out-of-the-box with this configuration.

Listing 18.6 Create an ECS service to spin up tasks running the web app

```

Service:
  DependsOn: LoadBalancerHttpListenerRule
  Type: 'AWS::ECS::Service'
  Properties:
    Cluster: !Ref 'Cluster'      ①
    CapacityProviderStrategy:
      - Base: 0
        CapacityProvider: 'FARGATE'  ②
        Weight: 1
    DeploymentConfiguration:
      MaximumPercent: 200  ③
      MinimumHealthyPercent: 100  ④
      DeploymentCircuitBreaker:
        Enable: true
        Rollback: true
      DesiredCount: 2  ⑥
      HealthCheckGracePeriodSeconds: 30  ⑦
    LoadBalancers:
      - ContainerName: 'app'  ⑨
        ContainerPort: 3000  ⑩
        TargetGroupArn: !Ref TargetGroup  ⑪
    NetworkConfiguration:
      AwsVpcConfiguration:
        AssignPublicIp: 'ENABLED'  ⑫
        SecurityGroups:
          - !Ref ServiceSecurityGroup  ⑬
        Subnets: [!Ref SubnetA, !Ref SubnetB]  ⑭
    PlatformVersion: '1.4.0'  ⑮
  TaskDefinition: !Ref TaskDefinition

```

- ① A service belongs to a cluster.
- ② Run tasks on Fargate. Alternatively, you could switch to FARGATE_SPOT to reduce costs similar to EC2 spot instances as discussed in chapter 3.
- ③ During a deployment ECS is allowed to double the number of tasks.
- ④ During a deployment ECS ensures that the number of running containers does not decrease.
- ⑤ By enabling the deployment circuit breaker, you ensure that ECS will not try forever to deploy a broken version.
- ⑥ ECS will run or stop tasks to make sure two tasks are up and running.
- ⑦ When a new task starts, ECS will wait for 30 seconds for the task to pass the health check. You need to increase this period for applications starting slow.
- ⑧ The ECS service registers and deregisters tasks at the load balancer.
- ⑨ To be more precise, a specific container gets registered at the load balancer.
- ⑩ The application is listening on port 3000.
- ⑪ The target group of the load balancer to register or deregister tasks.
- ⑫ When deploying to a public subnet, assigning public IP addresses is required to ensure outbound connectivity.
- ⑬ Each task comes with its own ENI. The security group defined here is used to filter traffic.

- ⑯ A list of subnets to start tasks in. You should use at least two different subnets and a desired count greater than 2 to achieve high availability.
- ⑰ From time to time, AWS releases a new Fargate platform with additional features. We highly recommend specifying a platform version instead of using LATEST to avoid issues in production.

Being able to scale workloads is one of the superpowers of cloud computing. Of course, our container-based infrastructure should be able to scale-out and scale-in based on load as well. Listing 18.7 shows how to configure auto-scaling. In the example, we are using a target tracking scaling policy. The trick is, that we only need to define the target value for the CPU utilization. The Application Auto Scaling service will take care of the rest and will increase or decrease the desired count of the ECS service automatically.

Listing 18.7 Configuring auto-scaling based on CPU utilization for the ECS service

```
ScalableTarget:
  Type: AWS::ApplicationAutoScaling::ScalableTarget
  Properties:
    MaxCapacity: '4'      ①
    MinCapacity: '2'       ②
    RoleARN: !GetAtt 'ScalableTargetRole.Arn'   ③
    ServiceNamespace: ecs ④
    ScalableDimension: 'ecs:service:DesiredCount' ⑤
    ResourceId: !Sub     ⑥
      - 'service/${Cluster}/${Service}'
      - Cluster: !Ref Cluster
        Service: !GetAtt 'Service.Name'
CPUScalingPolicy:
  Type: AWS::ApplicationAutoScaling::ScalingPolicy
  Properties:
    PolicyType: TargetTrackingScaling ⑦
    PolicyName: !Sub 'awsinaction-notea-${ApplicationID}'
    ScalingTargetId: !Ref ScalableTarget ⑧
    TargetTrackingScalingPolicyConfiguration:
      TargetValue: 50.0 ⑨
      ScaleInCooldown: 180 ⑩
      ScaleOutCooldown: 60 ⑪
    PredefinedMetricSpecification:
      PredefinedMetricType: ECSServiceAverageCPUUtilization ⑫
```

- ① The upper limit for scaling tasks.
- ② The lower limit for scaling tasks.
- ③ The IAM role is required to grant Application Auto Scaling access to CloudWatch metrics and ECS.
- ④ Application Auto Scaling supports all kinds of services. You want to scale ECS in this example.
- ⑤ Scale by increasing or decreasing the desired count of the ECS service.
- ⑥ This references the ECS service in the ECS cluster created above.
- ⑦ A simple way to scale ECS services is by using the target tracking scaling which requires minimal configuration.

- ⑧ This references the scalable target resource from above.
- ⑨ The target is to keep the CPU utilization at 50%. You might want to increase that to 70-80% in real-world scenarios.
- ⑩ After terminating tasks wait 3 minutes before re-evaluating the situation.
- ⑪ After starting tasks wait one minute before re-evaluating the situation.
- ⑫ In this example, we scale based on CPU utilization.
`ECSServiceAverageMemoryUtilization` is another predefined metric.

That's it, you have learned how to deploy a modern web application on ECS and Fargate.

Don't forget to delete the CloudFormation stack and all data on S3. Replace `$ApplicationId` with a unique character sequence you chose when creating the stack.

```
$ aws s3 rm s3://awsinaction-notea-$ApplicationID --recursive
$ aws cloudformation delete-stack --stack-name notea
$ aws cloudformation wait stack-delete-complete \
  --stack-name notea
```

What a ride. You have come a long way from AWS basics, to advanced cloud architecture principles, and with modern containerized architectures. Now only one thing remains to be said: Go build!

18.6 Summary

- App Runner is the simplest way to run containers on AWS. However, to achieve simplicity App Runner comes with limitations. For example, the containers aren't running in your VPC.
- The Elastic Container Service (ECS) and the Elastic Kubernetes Service (EKS) are both orchestrating container clusters. We recommend ECS for most use cases because of costs per cluster, integration into all parts of AWS, and CloudFormation support.
- With Fargate, you no longer have to maintain EC2 instances to run your containers. Instead, AWS provides a fully-managed compute layer for containers.
- The main components of ECS are: cluster, task definition, task, and service.
- The concepts from EC2-based architectures apply to container-based architectures as well. For example, an ECS service is the equivalent of an Auto Scaling Group.

Notes

1. AWS Customer Success, aws.amazon.com/solutions/case-studies/.
2. Learn more about Jenkins by reading its documentation at <http://wiki.jenkins-ci.org/display/JENKINS/Use+Jenkins>.